

Starting a New REST API project? A Performance Benchmark of Frameworks and Execution Environments

Sergio Di Meglio¹, Luigi Libero Lucio Starace¹ and Sergio Di Martino¹

¹Dept. of Electrical Engineering and Information Technology, Università degli Studi di Napoli Federico II, Naples, Italy

Abstract

REST APIs have become widely adopted in the software industry, finding extensive usage for business-critical purposes such as data exchange, mobile app development, and microservice architectures. Such popularity has led to a proliferation of dedicated frameworks, making it challenging for developers and organizations to choose which one to use for developing resource-efficient solutions. Adding to the complexity is the possibility of adopting also different execution environments, such as GraalVM, that offer advantages such as faster startup times, lower memory footprint, and polyglot capabilities.

Some prior works have investigated the performance of various frameworks for REST APIs. Still, these studies often consider simplistic scenarios with a single endpoint, which fail to capture the complexity and diversity of real-world REST API applications. Furthermore, the impact of different execution environments on performance was often overlooked. Consequently, there remains a significant knowledge gap in comprehensively assessing the combined influence of frameworks and execution environments on the performance of REST APIs. This study aims to move a first step towards bridging that gap, by conducting a thorough performance benchmark that encompasses real-world REST APIs and considers also the effects of different execution environments.

More in detail, the study focuses on two of the most popular programming language and framework combinations for REST APIs, namely JavaScript with the Express framework and Java with the Spring framework. As for the execution environment, we consider both mainstream execution environments for JavaScript and Java (Node and OpenJDK, respectively), and GraalVM, which can execute both Java and JavaScript software. The benchmarking process involves conducting realistic load and stress tests using state-of-the-art tools. Results reveal significant differences in performance across the considered combinations, providing insights that could support developers and system architects in making more informed decisions on the technologies to use for their REST API projects.

Keywords

Performance Benchmark, Rest API, Load Testing, Stress Testing, GraalVM

1. Introduction

The REST (REpresentational State Transfer) paradigm has become a prominent architectural style for designing distributed applications, providing a standardized approach for building scalable and interoperable systems leveraging the widely-used HTTP protocol [1, 2]. REST

IWSM-MENSURA 2023, September 14–15, 2023, Rome, Italy

✉ sergio.dimeglio@unina.it (S. Di Meglio); luigiliberolucio.starace@unina.it (L. L. L. Starace); sergio.dimartino@unina.it (S. Di Martino)

🌐 <https://www.linkedin.com/in/sergio-di-meglio> (S. Di Meglio); <https://luistar.github.io/> (L. L. L. Starace)

>ID 0000-0001-7945-9014 (L. L. L. Starace); 0000-0002-1019-9004 (S. Di Martino)

 © 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

APIs have gained wide adoption in the software industry, enabling critical functionalities such as data exchange, mobile app development, and microservice-based architectures [3]. Such popularity has led to the proliferation of dedicated frameworks, designed to streamline the process of designing, implementing and managing REST APIs.

With the proliferation of dedicated frameworks, developers and organizations face the challenge of selecting the most appropriate technologies for effectively developing high-performance REST APIs [4, 5]. This decision is influenced by various factors, including project requirements, the availability of resources and expertise within the development team, the ease of use of the framework and the availability of libraries and support. Performance aspects and resource efficiency also play a critical role in the decision process. Indeed, selecting a high-performance framework can significantly improve user satisfaction thanks to reduced latencies and response times. These non-functional properties are as crucial as functional correctness in today's highly competitive web and mobile application market [6, 7]. Moreover, using more resource-effective frameworks can also have a significant impact on deployment costs in pay-as-you-go public cloud services [8], and lead to lower energy consumption in general, which is increasingly important also from sustainability and *green-computing* perspectives. The decision-making scenario is further complicated by the possibility of running the applications using emerging and optimized execution environments, such as GraalVM [9], which promise faster startup times, lower memory footprints, and polyglot capabilities.

To support practitioners in these decisions, a number of works in the literature have focused on comparing programming languages and frameworks with controlled benchmarks. Most works, however, focus on programming languages in a general-purpose context, comparing different implementations of the same algorithms in terms of lines of code and running times [10, 11]. More recently, some studies began investigating the performance impact of adopting emerging execution environments such as GraalVM [12], but these works generally focus on a single programming language, and in specific contexts (e.g.: running unit tests). Some recent works have also compared different REST API frameworks written in different programming languages, evaluating their performance under different workloads [4, 5]. These works, however, relied on simplistic REST API implementations, which can hardly be considered representative of the complexity of real-world applications, limiting their generalizability. Moreover, to the best of our knowledge, no study has investigated the combined effect of different frameworks, written in different programming languages, and different execution environments in the context of REST APIs.

The goal of this paper is to move a first step towards filling that gap, by investigating the combined impact on the performance of realistic REST APIs of two of the most popular frameworks, and of different execution environments. To this end, we leveraged as subject REST APIs two different implementations (one using Java with the Spring Boot framework, and the other using JavaScript with the Express framework) of a prototype application provided by an industrial partner, and configured them to run both on a mainstream execution environment (OpenJDK for Java, Node for JavaScript) and on GraalVM. We deployed each REST API in a controlled environment and systematically analyzed their performance by subjecting them to automated performance tests. Results show that there exist significant differences in terms of performance and stability among the different considered configurations of frameworks and execution environments and provide useful insights based on which developers and organizations

can make more informed decisions.

The paper is organized as follows. Section 2 provides preliminary notions on the REST paradigm and dedicated frameworks, on the GraalVM runtime, as well as on performance testing practices, which are widely used to assess the performance of REST APIs. In Section 3, we present an overview of the state of the art on comparison of programming languages, frameworks, and execution environments, with a particular focus on the performance of REST API frameworks. Section 4 provides a detailed description of the benchmarking process we employed in our study, by describing the subject system, the considered combinations of frameworks and execution environments, the performance testing workloads we used to assess performance as well as the metrics we used to quantify it. In Section 5, we present and discuss the results of our benchmarks, while in Section 6, we discuss some threats to the validity of our findings. Last, in Section 7, we give some closing remarks and discuss future works.

2. Background

This section provides a brief overview of some preliminary concepts on which this paper is based. More in detail, we start by describing REST APIs and by presenting the most popular frameworks. Then, we describe the GraalVM polyglot runtime environment, hinting at the characteristics that make it attractive in the REST API domain. Last, we provide some preliminary notions on performance testing practices, which we use in our study to benchmark each framework/execution environment combination.

2.1. The REST Architectural Style

REST (REpresentational State Transfer) is a client/server architectural style designed to promote simplicity, scalability, and interoperability between different systems communicating over a network using the HTTP protocol [1]. At its core, REST is based on the concept of resources, which can be entities or objects that need to be accessed or manipulated over a network. Each resource is uniquely identified by its Uniform Resource Identifier (URI).

REST APIs expose these resources and allow clients to perform various operations on them, such as retrieving data, creating new resources, updating existing ones, or deleting them. The operations are typically carried out by means of HTTP requests to the appropriate URIs, using standard methods, such as GET, POST, PUT, and DELETE, which semantically align with the basic actions of retrieving, creating, updating, and deleting resources, respectively [13, 14].

Another key aspect of the REST style is *statelessness*, meaning that the server does not need to store any client-specific session information (*state*) between requests. Each request from the client must contain all the necessary information for the server to understand and process it. This statelessness makes REST APIs highly scalable and enables easy distribution and load balancing across multiple servers [15, 2, 16].

Due to its simplicity, scalability, and uniform, standardized interface, the REST architectural style has emerged as a prevalent choice in the software industry for developing robust and interoperable web services [3]. According to a recent survey [17], REST is by far the most widely-adopted architectural style for APIs, with an adoption rate exceeding 90%.

2.1.1. REST API frameworks

REST API frameworks provide developers with pre-built tools, libraries, conventions and abstractions to simplify the process of designing, implementing, and managing RESTful APIs. When starting a new project from scratch, choosing the most suitable framework is crucial, as it can significantly impact time-to-market, quality, consistency, and performance. Nowadays, a wide range of REST API frameworks is available, written in different programming languages, which can make the selection process quite challenging. The popularity of a programming language/framework plays a significant role in the decision process. Indeed, more popular programming languages tend to have a larger developer community and extensive documentation support. A recent survey involving 850 professional programmers from over 100 different countries revealed that, in 2022, JavaScript was the most widely used programming language to develop REST API, with a 48% adoption rate, and Java also accounted for a significant 19% of the share [18]. Within JavaScript, Express is by far the most popular backend framework, followed by other frameworks such as Nest or Fastify [19]. As for Java, Spring Boot is the most widely-adopted framework in the context of REST API development [20].

2.2. GraalVM

GraalVM is an emerging execution environment offering several advantages for running applications. It is a polyglot runtime that supports multiple programming languages, including Java, JavaScript, Python, Ruby, and more [21]. GraalVM efficiently handles polyglot applications with minimal performance loss, enabling developers to choose the most suitable language for their problem-solving needs [9]. It leverages both Just-In-Time (JIT) compilation and Ahead-Of-Time (AOT) compilation. The AOT compiler precompiles the code, creating a self-contained executable known as a native image. This approach reduces the workload at runtime as libraries and dependencies are already provided and loaded in advance. The JIT compiler in GraalVM dynamically compiles the code during runtime, constantly examining and optimizing it [12, 22]. Additionally, GraalVM can lead to a smaller memory footprint compared to traditional virtual machines, which can be beneficial, especially in resource-constrained environments. These features make GraalVM an attractive choice for developers seeking performance optimization and language flexibility in their applications, and its adoption has been observed in prominent companies like Disney+, Facebook, and Twitter [21, 12]. Nonetheless, there is a lack of literature addressing the extent of their usage in the REST API domain and the resulting improvements.

2.3. Performance Testing

Performance testing is the set of activities aimed at assessing the performance and correctness of the behaviours of a system under varying load levels. The objective is to quantify the performance of the system and identify potential load-related problems [23]. In the domain of REST APIs, as well as in web domains in general, a load is typically represented by sequences of requests generated by concurrent users, each performing different use cases over a specific time period [24]. For example, performance tests can simulate load peaks that may occur during the Christmas season for an e-store, helping assess whether the user experience remains satisfactory.

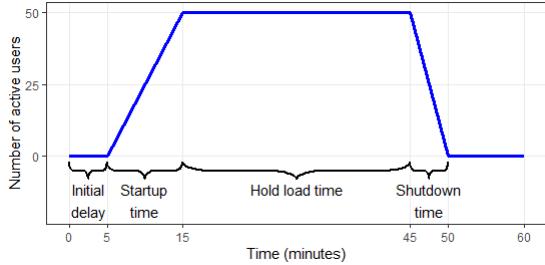


Figure 1: Example of an user group specification, from [26]

A load is typically defined as a set of different groups of users, each of which represents concurrent users that execute the same user session (sequence of interactions with the system) [25, 26]. Furthermore, to allow for the definition of more realistic loads, the behaviour of each user group can be customized by varying the following parameters:

- **Initial delay:** the time between the start of the test and the activation of the user group.
- **Startup time:** the time required to go from zero to the specified number of users.
- **Hold load time:** the time for which all activated users will remain active.
- **Shutdown time:** the time within which all activated users stop making requests.

Performance testing includes the broad set of activities aimed at evaluating that the system behaves as expected, under different loads [23, 27]. Based on the specific goals, it is possible to identify different types of performance testing such as *load testing* and *stress testing*.

Load testing is a type of performance testing that aims to ensure that the system under test in production performs as expected under expected realistic loads [28]. The main goal is to generate a load that is as realistic as what the system will find once it is released into production.

Stress testing focuses on putting a system under intense conditions in order to evaluate the ability of a system or component to handle peak loads exceeding the anticipated load limits. Stress testing is also used to assess the system's ability to handle low resource availability such as access to memory or servers [23, 27].

3. Related Works

Performance is a critical aspect of software development, as it directly impacts user experience, system responsiveness, and overall efficiency. Consequently, a good deal of studies in the literature have focused on investigating the performance of different programming languages, to support developers and organizations in making informed decisions on programming language selection based on their specific requirements and constraints. These studies aim to assess how various programming languages perform in terms of execution speed, memory usage, scalability, ease of use, and other relevant metrics. By comparing the performance characteristics of different languages. These performance studies often involve running benchmark tests, implementing algorithms or tasks in different programming languages, and measuring their execution times, resource consumption, or the number of lines of code that were required for the implementation.

For example, Prechelt et al. [11] compared seven different programming languages, namely C, C++, Java, Perl, Python, Rexx, and Tcl. They implemented the same application in each language, allowing for the evaluation of program length, programming effort, runtime effectiveness, memory consumption, and reliability. The results showed that scripting languages like Python create more concise programs but lack the speed and robustness of compiled and strongly typed languages like C. Similarly, Nanz et al. [10] compared software from the *Rosetta Code* repository, which includes around 700 programs of varying sizes written in different programming languages. In that study, the authors evaluated similar metrics as the study by Prechelet et al., including executable size, execution time, memory usage, and propensity to fail.

In the Web domain, Lei et al. [29] investigated the performance achievable by using three well-known web technology stacks, namely JavaScript with Node, PHP, and Python. They conducted objective systematic tests (benchmarks) involving *ApacheBench*, a stress testing tool that generates synthetic workloads, sending approximately 10,000 requests to a simple benchmark website offering three basic functionalities. Their study suggested that JavaScript achieved the best performance in terms of response times among the considered alternatives. Still, the study investigated generic web applications and is not specific to REST APIs. Moreover, the considered scenarios are simplistic and hardly representative of real-world applications. Lastly, the benchmarked technologies are nowadays obsolete (i.e., Node 0.10 which was used in the study reached end-of-life in 2016), and much more recent versions are available. More recently, benchmark studies focusing on REST APIs were conducted by Kemer and Samli [4] and by Abbade et al. [5]. These studies compared the performance of different types of REST APIs developed with various programming languages.

Kemer and Samli [4] compared the performance of REST APIs built in C#, Java, Go, Node.js, and Python. To this end, they designed and implemented a set of applications in the simplest form possible, without additional tasks or dependencies. The performance comparison was conducted by carrying out load and stress tests using the *K6* and *Locust* open-source tools. The results showed that Python had unstable request handling with the slowest response times. Go, C#, and Java performed the best, while Node.js had similar performance with slightly slower average response times.

Similarly, Abbade et al. [5] compared the performance of a simple POST route endpoint containing an authorization header and a JSON payload to be inserted into a *MongoDB* database. They considered REST API applications developed using Java with Spring Boot, JavaScript with the Express framework, and Python with Flask. Load testing was conducted using the *Locust* tool, and results showed that JavaScript achieved the best performance, while Java proved to be the most stable in terms of performance over time. Python had the worst results in terms of response times and the number of managed and failed requests.

All these previous studies based their comparisons on REST API applications with a single route, which can hardly be considered representative of the complexity of modern applications, limiting the generalizability of the studies. Moreover, the comparisons were limited to different REST APIs written in different languages (Java, JavaScript, Python, Go, etc.), and did not take into account the potential impact of varying the underlying execution environment. Indeed, in recent years, polyglot and optimized execution environments, such as GraalVM, are becoming increasingly popular. While several studies have explored how such runtime environments can improve execution time or memory footprint compared to mainstream solutions such as

OpenJDK and OracleJDK in specific domains such as unit test execution [12], their performance impact in the domain of REST APIs remains, to the best of our knowledge, unexplored.

This work aims at advancing the state of the art by investigating the combined influence of different modern and largely adopted programming languages/frameworks and runtime environments on the performance achievable by realistic REST APIs.

4. Performance Benchmark

This section describes the performance benchmark we conducted to compare the performance of two of the most popular REST API frameworks, namely Spring Boot (Java) and Express (JavaScript). The comparison does not only take into account the frameworks but rather also considers different execution environments. In particular, in addition to the mainstream Java and JavaScript environments (i.e., OpenJDK and Node), we also consider GraalVM.

In the remainder of this section, we present the performance benchmark we conducted by describing in detail the subject REST API, the considered combinations of frameworks and execution environments, the performance testing activities we conducted to assess the performance of each combination, the considered metrics and the adopted procedure.

4.1. The subject REST API

The REST API employed for the experiment is an industrial prototype developed by one of our industrial partners. The API allows users to create accounts and log in. Club owners can register their clubs and add information about facilities available, such as tennis courts or football fields. Users can search for available facilities, make reservations for specific dates and times, and also assign a satisfaction rating to each reservation. The API ensures secure access and protects user data through authentication and authorization mechanisms. The endpoints of the REST API and the supported HTTP methods, along with a brief description of the provided functionalities, are reported in Table 1.

4.2. Considered Frameworks and Execution Environments

The benchmark REST API was implemented twice, utilizing different frameworks for each implementation. The first implementation was developed using the Spring Boot framework (version 3.0.0), leveraging the power of Java as the programming language. The second implementation, on the other hand, leveraged the Express framework (version 4.16.4), which is the most popular JavaScript back-end framework. Both the implementations rely on the same PostgreSQL (version 15.1) relational database instance for data persistence.

As for the execution environments, we consider both mainstream Java and JavaScript execution environments (OpenJDK 17.0.2 and Node 19.5.0, respectively), and GraalVM 22.3.2, which can execute both Java and JavaScript software. Thus, we consider four combinations of programming framework and execution environment, with each framework being benchmarked with both the mainstream runtime environment for its programming language, and with GraalVM.

Table 1

List of endpoints and supported HTTP methods provided by the REST API.

Method	Path	Description
<i>POST</i>	/login	authenticates users and generates an access token
<i>GET</i>	/sports-facilities	retrieves a list of sport facilities
<i>GET</i>	/sports-facilities/{id}	retrieves the sport facility with the given id
<i>POST</i>	/sports-facilities/{id}/sports-field	creates a sports field for the given facility
<i>GET</i>	/sports-fields	retrieves a list of sports fields
<i>GET</i>	/sports-fields/{id}	retrieves a sport field by its id
<i>POST</i>	/users	creates a new user
<i>GET</i>	/users/{id}	retrieves a user by id
<i>GET</i>	/reservations	retrieves a list of sports fields reservations
<i>POST</i>	/reservations	creates a reservation request for a sports field
<i>GET</i>	/reservations/{id}	retrieves the reservation with the provided id
<i>PUT</i>	/reservations/{id}/status	updates the status of the given reservation
<i>GET</i>	/reservations/{id}/rating	retrieves the rating for a given reservation
<i>POST</i>	/reservations/{id}/rating	create a rating for the given reservation

Table 2

User groups used for load testing.

Group name	User behaviour description
Create facilities	After logging in, users retrieve the list of sport clubs and add a new facility to one of them.
Reserve a facility	After signing up and logging in, users request the list of available facilities and make a reservation for one of them.
Retrieve facilities	After logging in, users retrieve a list of facilities that are currently available.
Rate reservations	After signing up and logging in, users create a reservation for a sport facility and subsequently provide a rating for the experience.
Accept reservations	After signing up and logging in, users create a reservation for a facility, and the club owner accepts the reservation.

4.3. Performance Testing

To compare the performance of each configuration, we conducted performance tests using state-of-the-art tools. In particular, we carried out both load and stress testing activities, as also done in benchmark studies in the web domain [4, 5].

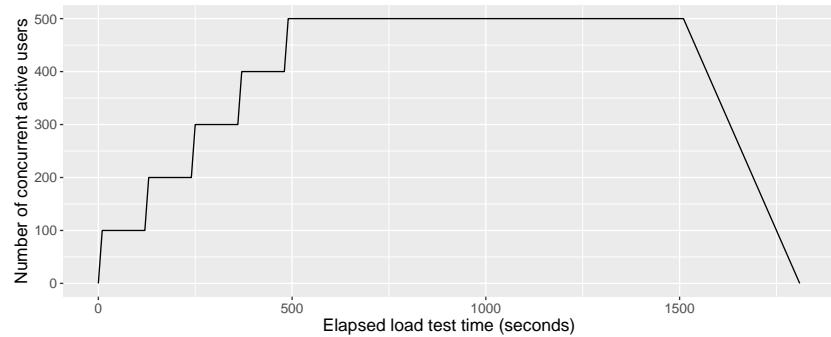
4.3.1. Load Testing

The main purpose of load testing is to assess system performance under realistic load conditions. To this end, we manually designed a workload consisting of five distinct user groups, each representing a specific sequence of realistic user interactions. A brief description of the considered user groups is provided in Table 2. The activity levels of each user group during load testing, i.e., how many users are actively performing that sequence of actions at any given time,

Table 3

Characterization of a single user group within load testing.

Start Thread Count	Initial Delay (secs)	Startup Time (secs)	Hold Load Time (secs)	Shutdown Time (secs)
20	0	10	1500	300
20	120	10	1380	300
20	240	10	1260	300
20	360	10	1140	300
20	480	10	1020	300

**Figure 2:** Number of active users over time in the load testing activity.

is formalized in Table 3 and described as follows. The load test starts with 20 users performing the sequence of actions in each user group. Subsequently, the load incrementally increases every two minutes, by adding 20 additional users until reaching 100 concurrent users for each user group. The workload lasts for approximately 30 minutes, with a peak usage period lasting 17 minutes, during which 500 active users exhibit one of the five user-group behaviours. The distribution of active users over time in the considered workload is also visualized in Figure 2.

4.3.2. Stress Testing

As for the stress testing activity, the system is subjected to increased loads or adverse conditions beyond its normal operational limits. The goal, in this case, is to identify the breaking point or threshold (intended as the maximum number of concurrent users) at which the system fails to perform adequately. For this activity, we designed a workload consisting of the same user behaviours leveraged for load testing, but with a way steeper increase in the number of concurrent users over time.

In particular, the workload starts with 100 concurrent users performing each of the considered behaviours, and the number of users in each group is increased by 100 users every minute. The test continues until one of the following failing conditions is met. Either (1) the average response time exceeds 30 seconds for a continuous duration of 60 seconds, or (2) the response error rate surpasses 50% for a continuous duration of 10 seconds.

4.4. Procedure and Metrics

The REST API involved in this study was implemented twice, using the two considered frameworks, by practitioners working with our industrial partner, supervised by one of the authors of this paper. To simplify the deployment of the different implementations of the REST API in each considered execution environment, we set up a different Docker container for each of the four considered combinations of framework and execution environment. The PostgreSQL instance required by the REST API to manage data persistence has also been deployed in a separate container. The orchestration of the REST API containers and the PostgreSQL container has been managed by leveraging the Docker Compose tool. In our experimental setup, we adopted a systematic approach to evaluate the performance of each configuration. To ensure accurate and reliable results, we deployed each configuration individually on a dedicated Linux server with an AMD Ryzen 9 5900X 12-Core processor, 64GB RAM, and a 1TB SSD. This approach allowed us to isolate the effects of programming languages, frameworks, and execution environments on the performance of the REST APIs. As for the Java execution environments, we configured both OpenJDK and GraalVM to use at most 16GB of heap space.

To generate the workloads for load and stress testing, we leveraged Apache JMeter, a widely-used, state-of-the-art tool for performance testing [30, 26]. We implemented the load and stress testing workloads described in the previous section as JMeter scripts and ran each of them on every considered combination of framework and execution environment. To ensure no performance interference, the workloads were executed on a different server, featuring an Intel(R) Core(TM) i9-9940X CPU, 64 GB RAM, and a 1TB SSD, connected to the main server running the REST API via a local high-speed network. To account for fluctuations due to the interference of external factors and to ensure accuracy and reliability, we performed five repetitions of the load and stress testing for each of the considered framework/execution environment configurations, computing, for each metric, the average across the repetitions.

For each execution of the load and stress tests, we leveraged the capabilities offered by the JMeter tool to collect detailed data on the way each generated request was handled. In particular, we collected, for each request to the REST API that was generated by the tool, the elapsed response time (i.e., the time elapsed from just before sending the request to just after the response has been completely received), and the HTTP status of the response, indicating whether the response was handled correctly or some error occurred. Starting from these raw data, we computed additional metrics that have also been used in several other empirical works aimed at comparing the performance of REST API frameworks [5, 4].

More in detail, in the context of load testing, we computed:

- the overall **sample count**, i.e., the number of requests that were processed during the execution of the load test. REST APIs which perform better manage more requests in the same amount of time.
- the **error rate**, i.e., the percentage of requests that were not handled correctly by the REST API, which resulted in an error response.
- **Mean and median elapsed response times**.
- **Throughput**, i.e., the number of requests being processed per second.
- **Application Performance Index (Apdex)**, a well-known metric to quantify user satisfaction with the performance of software applications [31]. Apdex can assume values

between 0 and 1, with 0 indicating complete dissatisfaction, and 1 representing complete satisfaction. This metric is computed based on two thresholds: the tolerance threshold T and the frustration threshold F . These thresholds represent, respectively, the maximum *ideal* response time, and the maximum *acceptable* response time. Requests that are handled within the tolerance threshold T , are considered *satisfied*, while requests that took more than T , but less than the frustration threshold F are considered *tolerated*. Given these definitions, the Apdex metric for a load test is defined as follows:

$$Apdex = \frac{Num. Satisfied + Num. Tolerated/2}{Num. of Requests}$$

In this study, the tolerance threshold T and frustration threshold F used to calculate the Apdex metric are set to 1 second and 10 seconds, respectively, according to the guidelines presented by Nielsen in [32].

As for stress testing, we considered the **time-to-failure** metric [33], commonly used to evaluate performance under stress. It measures the time a system or component takes to reach a failure state under the applied stress load. REST APIs that can handle increasing workloads more gracefully (i.e., that scale better) will last longer under heavy workloads.

5. Benchmark Results and Discussion

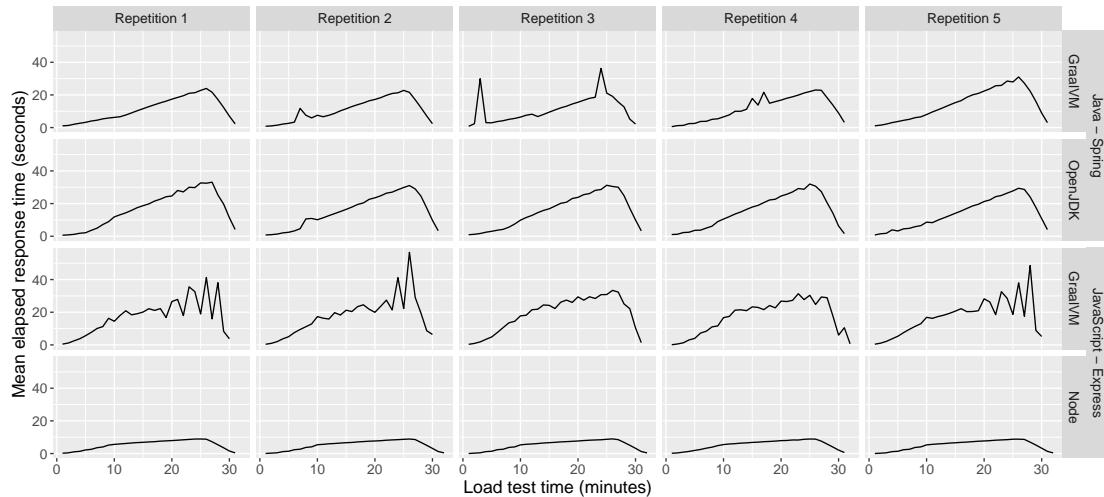
The results of the load testing benchmark are reported in Table 4, which lists, for each considered combination of framework and execution environment, the sample count, the error rate, average and median elapsed response times (in seconds), throughput and Apdex metric. Overall, the Express and Node configuration significantly outperforms all the other configurations, managing to process more than double the number of requests over the same time. This is also witnessed by the remarkably lower average and median elapsed response times, on a higher throughput and in a better user experience, as confirmed by the Apdex score. Moreover, despite serving the highest number of requests with the shortest response times and with the best user experience, the Express and Node configuration also exhibits the smallest error rate among the considered configurations, with remarkably no request resulting in an error, proving to be also the most reliable configuration.

Considering the impact of the execution environments, the results highlight that different trends seem to exist, depending on the considered programming language. When dealing with JavaScript code and with the Express framework, using GraalVM leads to significantly worse performance w.r.t. the Node runtime. Indeed, the Express/GraalVM configuration achieves the worst performance among all the considered configurations, in terms of sample count, response times, and throughput, with a remarkably high 4% error rate. When dealing with Java code and with the Spring Boot framework, on the other hand, switching to GraalVM can improve performance w.r.t. the mainstream OpenJDK runtime, leading to serving 12% more requests with 10% faster average response time, and improvements in throughput and Apdex as well. These improvements come at the cost of a slightly higher error rate (0,5% with GraalVM, 0,2% with OpenJDK), and higher median response times.

Table 4

Results of the load testing benchmark.

Configuration		Sample Count	Errors (%)	Average Resp. Time	Median Resp. Time	Throughput	APDEX
Java - Spring	GraalVM	80477	0,5	8,9	6,5	44,5	0,36
	OpenJDK	71729	0,2	9,9	5,9	39,7	0,34
JS - Express	GraalVM	60648	4,0	14,4	10,6	33,7	0,39
	Node	176047	0,0	4,9	4,7	97,3	0,55

**Figure 3:** Elapsed response times over time during load testing, for each repetition of the experiments.

The results of the load testing benchmark are also visualized in Figure 3, which shows, for each configuration and repetition of the benchmark, the average response times over time computed at a 1-minute granularity during the execution of the load test. As can be seen from the figure, Express with Node.js has the most stable behaviour in each configuration and the best response times (never exceeding 10 seconds) compared to the other configurations. Spring Boot with OpenJDK also shows stable behaviour in every configuration but with a higher response time than Express. GraalVM, on the other hand, both when used with Express and with Spring Boot, results in unstable behaviour with sudden spikes in response times.

The distribution of response times for each REST API endpoint involved in the load testing activities is depicted with box plots in Figure 4. The figure highlights that there seems to be no particular difference between the response times distributions for each endpoint in each configuration, with the only exception being the configuration with Express and GraalVM, for which the endpoints involving POST requests are associated with longer response times. Moreover, the greater variability of GraalVM configurations is confirmed by the presence of a larger number of outliers.

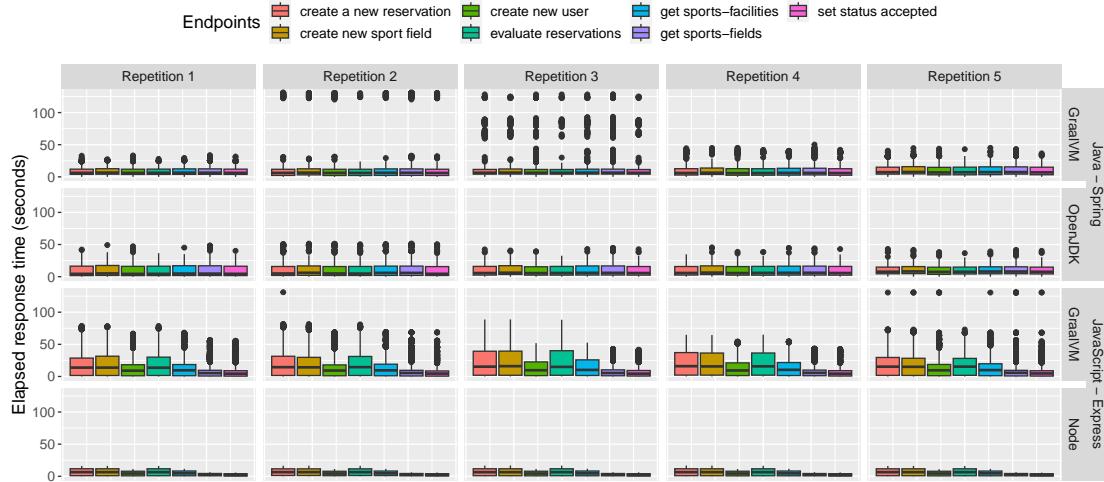


Figure 4: Elapsed response times distribution during load testing, for each endpoint and for each repetition of the experiment.

Table 5

Time to failure (in seconds) in stress testing (higher is better).

	Java (Spring)		JavaScript (Express)	
	GraalVM	OpenJDK	GraalVM	Node
Repetition 1	136	128	120	190
Repetition 2	79	129	107	190
Repetition 3	85	130	114	191
Repetition 4	86	132	114	193
Repetition 5	85	95	112	191
Average	94,2	122,8	113,4	191

Load testing results: Express with Node.js outperforms all other combinations of frameworks and execution environments, both in terms of response times and stability thereof during the load tests. The other combinations achieve roughly comparable performance, with those leveraging GraalVM generally exhibiting a greater variability in response times.

As for the stress testing benchmark, time-to-failure results are reported in Table 5. In all the repetitions and for all the considered framework/environment configurations, the stress tests failed because of the violation that concerned the average response time (see Section 4.3.2). These results show that the Express/Node configuration is also the most effective in handling the stress workload, with remarkably higher time-to-failure than all other configurations. Indeed, the average time-to-failure of the Express/Node configuration amounts to 191 seconds, which is 56% higher than the second-best configuration, namely Java/OpenJDK, which failed the stress test after 123 seconds on average. In this context, the adoption of GraalVM leads to worse

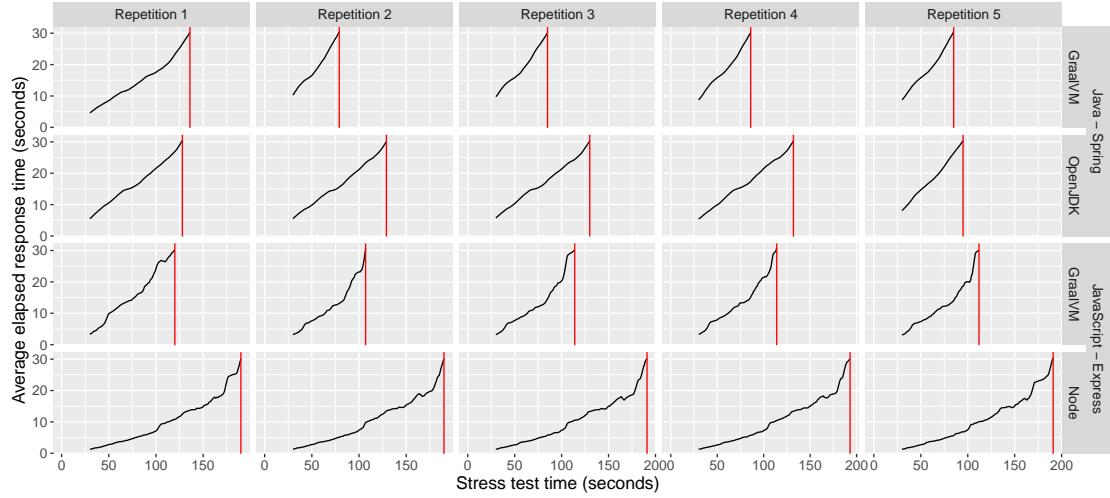


Figure 5: Elapsed response times over time during stress testing. Test failure is highlighted with a red bar.

performance in both Java and JavaScript applications, with the Java/GraalVM configuration being the one that performs the worst, failing on average after only 94 seconds. In Figure 5, we depict the average elapsed response time, computed in a 30-second sliding window, of the REST API under different configurations during the stress tests. The plots interrupt when a failure occurs, which is to say when the average response times in the 30-second window exceeds 30 seconds. The failing time is represented by the red vertical bar. These plots show that the Express/Node configuration is not only the most effective one in terms of overall time-to-failure but also seems to exhibit a slightly sub-linear increase in response times as the number of concurrent users increases, as opposed to the other configurations which exhibit a more linear (and steep) worsening in response times as the load increases.

Stress testing results: Express with Node.js proved to be the most effective configuration for handling the stress workload, with a 56% higher time-to-failure than the second-best configuration, i.e., Java Spring with OpenJDK. In the stress testing setting, the adoption of GraalVM leads to a worsening in performance with both Spring and Express.

6. Threats to Validity

In this section, we discuss some threats that could have affected the results of the benchmarks and their generalisation, according to the guidelines proposed by Wohlin et al. in [34].

6.1. Threats to Internal Validity

Threats to internal validity are factors that can undermine the causal conclusions or relationships established within a study, challenging the ability to draw accurate inferences about cause-

and-effect relationships. In this benchmark study, we ensured a fair comparison between the different configurations of frameworks and execution environments by deploying each of them, in isolation, on the same Linux server, with the same environment. By leveraging the Docker tool, we ensured that each repetition of the benchmarks started from the exact same initial configuration. Moreover, to limit the influence of external factors, we ensured that no other unnecessary service was running on the server, and that the REST API did not depend on external services, whose network latency could influence the benchmark results. As for the metrics used for capturing the dependent variables, they were computed automatically by the JMeter tool, which is recognized as a state-of-the-art and reliable performance testing solution [26].

An additional threat to the internal validity of our study lies in the possible presence of errors in the different implementations of the considered REST API. Indeed, some bugs could have been introduced when implementing the REST API using one of the considered frameworks, and this could affect the benchmark results. To mitigate this threat, each implementation of the REST API was subjected to the same automated functional test suite, that both implementations passed. Moreover, two of the authors of this paper manually reviewed the source code of the implementations on an endpoint-by-endpoint basis, to ensure that no bugs were introduced.

6.2. Threats to External Validity

These threats concern the generalizability of the findings to real-world scenarios beyond the specific context examined. One possible threat is posed by the selection of the REST API used in the study. Indeed, despite being an industrial prototype managing multiple resources with several endpoints and supporting all the operations that are typically present in REST APIs, including the presence of authentication and authorization mechanisms, the considered application might not be representative of every industrial REST API project. To mitigate this threat, future works should include additional REST APIs, possibly representing different kinds of industrial projects (e.g.: I/O intensive APIs, computation intensive ones, etc.).

An additional threat to the generalizability of the findings lies in the synthetic workloads employed to conduct load and stress testing. Indeed, despite our efforts in designing realistic workloads, the synthetic sequence of requests generated by performance testing tools may not be able to fully capture the complexities and variability encountered in real-world environments.

Lastly, the findings of this benchmarking study may be time-sensitive, as performance characteristics of programming languages, frameworks, and execution environments can evolve over time. New releases, updates, or optimizations in the evaluated technologies could impact the results and limit the study's long-term validity.

7. Conclusions

The REST paradigm has gained popularity as an architectural style for designing scalable and interoperable distributed systems. REST APIs have become widely adopted in the software industry, facilitating critical functionalities such as data exchange and mobile app development. This increased adoption has led to the development of dedicated frameworks aimed at simplifying the design and implementation of REST APIs.

However, the abundance of available frameworks presents a challenge for developers and organizations when selecting the most suitable technologies for building REST APIs. Performance and resource efficiency are among the key factors influencing the decision-making process. Indeed, choosing a high-performance framework can enhance user satisfaction by reducing latencies and response times, and opting for resource-efficient frameworks can contribute to cost savings in cloud services and promote sustainability efforts in terms of energy consumption. Adding to the complexity, the possibility of adopting emerging execution environments like GraalVM could lead to enticing benefits such as faster startup times, reduced memory footprints, and polyglot capabilities, further complicating the decision-making process.

Existing literature has explored the performance of programming languages and frameworks through controlled benchmarks. However, most studies focus on general-purpose contexts or single programming languages, lacking a comprehensive investigation of combined effects. Moreover, previous works evaluating REST API frameworks have often used simplistic implementations that do not reflect the complexity of real-world applications, limiting their applicability. To bridge this gap, this paper investigated the combined impact of different frameworks written in different programming languages and executed in different environments on the performance of realistic REST APIs. Results showed that significant differences in performance exist among the considered configurations, revealing that the Express JavaScript framework with the Node runtime outperformed all the other configurations in terms of stability and performance. The adoption of GraalVM, while proving to be pejorative for the JavaScript REST API, led to some benefits in performance for the REST API implemented with Java and the Spring Boot framework, but these improvements came at the cost of reduced stability and sudden spikes in response times.

In future works, we plan to extend our study by considering a larger number of REST API frameworks, including Python ones such as FastAPI or Django, which are very popular. We also plan to consider also additional factors, beyond performance, influencing the effectiveness of a framework. In particular, we plan to take into account ease of use and the effort required to implement the API. Future research could also focus on the resource efficiency of different frameworks, monitoring memory and CPU usage as well as energy consumption, and possibly compare the hosting costs in public cloud providers necessary to guarantee the same service levels using different frameworks and execution environments.

Data Availability Statement

The subject REST API implementations, the Docker environments to run them in the different execution environments, as well as the JMeter scripts to run the load and stress tests, are available in the replication package at [35]. The replication package also includes the outputs of each execution of the load and stress tests on each configuration, as well as all the code necessary to compute the considered metrics, and data analytics scripts we used to analyze the raw data and produce plots and results presented in this paper.

References

- [1] I. O. Suzanti, N. Fitriani, A. Jauhari, A. Khozaimi, Rest api implementation on android based monitoring application, in: *Journal of Physics: Conference Series*, volume 1569, IOP Publishing, 2020, p. 022088.
- [2] L. Zhang, L. Hang, W. Jin, D. Kim, Interoperable multi-blockchain platform based on integrated rest apis for reliable tourism management, *Electronics* 10 (2021) 2990.
- [3] S. Kotstein, J. Bogner, Which restful api design rules are important and how do they improve software quality? a delphi study with industry experts, in: *Service-Oriented Computing: 15th Symposium and Summer School, SummerSOC 2021, Virtual Event, September 13–17, 2021, Proceedings 15*, Springer, 2021, pp. 154–173.
- [4] E. Kemer, R. Samli, Performance comparison of scalable rest application programming interfaces in different platforms, *Computer Standards & Interfaces* 66 (2019) 103355.
- [5] L. R. Abbade, M. A. da Cruz, J. J. Rodrigues, P. Lorenz, R. A. Rabelo, J. Al-Muhtadi, Performance comparison of programming languages for internet of things middleware, *Transactions on Emerging Telecommunications Technologies* 31 (2020) e3891.
- [6] A. Corazza, S. Di Martino, A. Peron, L. L. L. Starace, Web application testing: Using tree kernels to detect near-duplicate states in automated model inference, in: *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–6.
- [7] S. Di Martino, A. R. Fasolino, L. L. L. Starace, P. Tramontana, Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing, *Software Testing, Verification and Reliability* 31 (2021) e1754.
- [8] V. Casola, A. De Benedictis, S. Di Martino, N. Mazzocca, L. L. L. Starace, Security-aware deployment optimization of cloud–edge systems in industrial iot, *IEEE Internet of Things Journal* 8 (2020) 12724–12733.
- [9] M. Šipek, B. Mihaljević, A. Radovan, Exploring aspects of polyglot high-performance virtual machine graalvm, in: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 1671–1676. doi:[10.23919/MIPRO.2019.8756917](https://doi.org/10.23919/MIPRO.2019.8756917).
- [10] S. Nanz, C. A. Furia, A comparative study of programming languages in rosetta code, in: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, 2015, pp. 778–788. doi:[10.1109/ICSE.2015.90](https://doi.org/10.1109/ICSE.2015.90).
- [11] L. Prechelt, An empirical comparison of seven programming languages, *Computer* 33 (2000) 23–29.
- [12] F. Fong, M. Raed, Performance comparison of graalvm, oracle jdk and openjdk for optimization of test suite execution time, 2021.
- [13] K. Mohamed, D. Wijesekera, Performance analysis of web services on mobile devices, *Procedia Computer Science* 10 (2012) 744–751.
- [14] F. Halili, E. Ramadani, et al., Web services: a comparison of soap and rest services, *Modern Applied Science* 12 (2018) 175.
- [15] R. Ramanathan, T. Korte, Software service architecture to access weather data using restful web services, in: *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 2014, pp. 1–8. doi:[10.1109/ICCCNT.2014.6963122](https://doi.org/10.1109/ICCCNT.2014.6963122).

- [16] M. Biehl, RESTful Api Design, volume 3, API-University Press, 2016.
- [17] J. Simpson, 20 impressive api economy statistics: Nordic apis, 2022. URL: <https://nordicapis.com/20-impressive-api-economy-statistics/>.
- [18] RapidAPI, State of APIs 2022: Rapid developer survey results, <https://stateofapis.com/>, 2023. Online; accessed 2023-06-23.
- [19] S. Greif, E. Burel, State of JavaScript 2022, 2022. URL: <https://2022.stateofjs.com/>.
- [20] JetBrains Inc., 2022 Developer Ecosystem - Java, 2022. URL: <https://www.jetbrains.com/lp/devcosystem-2022/java/>.
- [21] GraalVM, Online documentation, <https://www.graalvm.org/latest/docs/introduction/>, 2023. Online; accessed 2023-06-23.
- [22] J. A. Romero-Ventura, U. Juárez-Martínez, A. Centeno-Téllez, Polyglot programming with graalvm applied to bioinformatics for dna sequence analysis, in: New Perspectives in Software Engineering: Proceedings of the 10th International Conference on Software Process Improvement (CIMPS 2021) 10, Springer, 2022, pp. 163–173.
- [23] K. Yorkston, K. Yorkston, Performance testing tasks, Performance Testing: An ISTQB Certified Tester Foundation Level Specialist Certification Review (2021) 195–354.
- [24] M. Shams, D. Krishnamurthy, B. Far, A model-based approach for testing the performance of web applications, in: Proceedings of the 3rd international workshop on Software quality assurance, 2006, pp. 54–61.
- [25] Apache JMeter, Ultimate thread group in jmeter, <https://jmeter-plugins.org/wiki/UltimateThreadGroup/>, 2023. Online; accessed 2023-06-23.
- [26] E. Battista, S. DI Martino, S. Di Meglio, F. Scippacercola, L. L. L. Starace, E2E-loader: A framework to support performance testing of web applications, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), 2023, pp. 351–361. doi:[10.1109/ICST57152.2023.00040](https://doi.org/10.1109/ICST57152.2023.00040).
- [27] D. A. Menascé, Load testing of web sites, IEEE internet computing 6 (2002) 70–74.
- [28] Z. M. Jiang, A. E. Hassan, A survey on load testing of large-scale software systems, IEEE Transactions on Software Engineering 41 (2015) 1091–1118.
- [29] K. Lei, Y. Ma, Z. Tan, Performance comparison and evaluation of web development technologies in PHP, Python, and Node.js, in: 2014 IEEE 17th International Conference on Computational Science and Engineering, 2014, pp. 661–668. doi:[10.1109/CSE.2014.142](https://doi.org/10.1109/CSE.2014.142).
- [30] Apache JMeter, Apache JMeter, <https://jmeter.apache.org/>, 2023. Online; accessed 2023-06-23.
- [31] P. Sevcik, Defining the application performance index, Business Communications Review 20 (2005) 8–10.
- [32] J. Nielsen, Usability engineering, Morgan Kaufmann, 1994.
- [33] K. Qiu, Z. Zheng, K. S. Trivedi, B. Yin, Stress testing with influencing factors to accelerate data race software failures, IEEE Transactions on Reliability 69 (2020) 3–21. doi:[10.1109/TR.2019.2895052](https://doi.org/10.1109/TR.2019.2895052).
- [34] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.
- [35] S. Di Meglio, L. L. L. Starace, S. Di Martino, Replication Package for the paper titled “Starting a new REST API project? A performance benchmark of frameworks and execution environments”, 2022. URL: <https://zenodo.org/record/8059181>. doi:[0.5281/zenodo.805918](https://doi.org/10.5281/zenodo.805918).