IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

Research Article

Performance Comparison of RESTful Web APIs using a Test Suite: .NET vs. Java Spring Boot

[1]Antonio GODINHO, [2]Jose ROSADO, [3]Filipe SA and [4]Filipe CARDOSO

[1]Polytechnic Institute of Coimbra, Technology and Management School of Oliveira do Hospital,
Rua General Santos Costa, 3400-124 Oliveira do Hospital, Portugal
and Research Center for Arts and Communication (CIAC), Santarém Polytechnic University
Complexo Andaluz, Apartado 279, 2001 -904 Santarém, Portugal
[2]Polytechnic Institute of Coimbra, Coimbra Institute of Engineering
Rua Pedro Nunes - Quinta da Nora, 3030-199 Coimbra, Portugal
and INESC Coimbra - Instituto de Engenharia de Sistemas e Computadores de Coimbra
Rua Sílvio Lima, Pólo II,3030-790 Coimbra, Portugal
[3]Polytechnic Institute of Coimbra, Coimbra Institute of Engineering
Rua Pedro Nunes - Quinta da Nora, 3030-199 Coimbra, Portugal
[4]Santarém Management School, Santarém Polytechnic University
Complexo Andaluz, Apartado 279, 2001 -904 Santarém, Portugal
and INESC Coimbra - Instituto de Engenharia de Sistemas e Computadores de Coimbra
Rua Sílvio Lima, Pólo II,3030-790 Coimbra, Portugal

Correspondence should be addressed to: Antonio GODINHO; agodinho@iscac.pt

Received date:21 February 2024; Accepted date:19 June 2024; published date: 27 August 2024

Academic Editor: António Trigo

Abstract

Modern web development methods use full-stack to split front-end (client-side) and back-end (server-side) components. Front-end technologies involve what the user sees and interacts with, and back-end technologies involve the server-side logic, databases, and server configuration. Both sections can be technologically independent, yet there's a need for a communications protocol. In modern web development, web APIs enable applications to interact with external services and exchange data, allowing the back-end to communicate with multiple and different front-ends. The landscape of software development, especially in web platforms, is in a constant state of technological advancement. Selecting the right technology for building a Web API requires a comparative analysis to make informed decisions. Performance testing of a web API involves evaluating various performance characteristics, such as response time, reliability, scalability, and resource utilization under different scenarios. However, many testing frameworks focus on specific components or HTTP methods rather than considering the entire technology stack, potentially leading to inaccurate performance assessments. In this study, two web APIs were developed—one using .NET and the other employing Java Spring Boot. Both APIs use the same database engine and the same database to manipulate identical datasets. By utilizing a test scenario and toolset, real-world conditions can be simulated, allowing for the evaluation and visualization of the results of each test to facilitate performance comparison.

Keywords: Web API, Performance tests, Full-stack development, .NET, Java

Cite this Article as: Antonio GODINHO, Jose ROSADO, Filipe SA and Filipe CARDOSO (2024)," Performance Comparison of RESTful Web APIs using a Test Suite: .NET vs. Java Spring Boot ", Journal of Software & Systems Development, Vol. 2024 (2024), Article ID 478010, https://doi.org/10.5171/2024.478010

Introduction

Web development typically involves both client-side and server-side, referred to as full-stack web development (Lee, Jin and International Society for Computers and Their Applications, 2019). Full-stack development has seen significant growth in recent years as the need for web development has increased with the growth of the Internet and e-commerce. With the rise of the cloud, micro-services architecture, and the need to create and maintain complex web applications, full-stack developers have become increasingly in demand.

One of the main drivers of this growth is the increasing popularity of JavaScript, which is used for both front-end and back-end development, but especially on the front-end side. JavaScript has evolved over the years, and now it has an extended set of tools for back-end and front-end development, such as Node.js, Angular, React, and Vue.js, that allows developers to use the same language for both the front-end and back-end, making full-stack development more accessible. Web APIs (Application Programming Interfaces) play a crucial role in full-stack development as they allow the different components of a web application to communicate with each other. Complex applications can be divided into teams for the back-end and front-end, working independently and allowing different technologies to be used separately. This approach allows the split of the developing challenges into smaller and simpler tasks, reducing the complexity and potential for coding bugs.

An API is an interface with functions, tools, and protocols to integrate application software and services. Web API is an API that can be accessed via the web using the HTTP/HTTPS protocols. It allows requesting systems to access and manipulate web resources using a uniform and predefined set of rules. Interaction in REST-based systems happens through the Internet's Hypertext Transfer Protocol (HTTP) (Fielding, 2000).

A Web API allows the front-end of a web application to interact with the back-end by making requests to specific endpoints and receiving data in response. This separation allows the front-end to display dynamic data, such as user information or a shopping cart's contents, and perform operations, such as submitting a form or making a payment. The back-end is often written using technologies such as Java, Python, or Node.js and also uses web development frameworks.

In 2020, a survey from the developer nation showed that nearly 90% of developers use APIs, proving that the emergence of APIs has been a critical factor behind the developer ecosystem boom in the past few years (Voskoglou, 2020).

With an increasing number of programming languages, many with similar components and coding styles, performance should play a role in choosing a language/framework. The proper way to do this evaluation is to develop two different Web APIs using various technologies that use the same database and display the same output.

RESTful Web API

A RESTful Web API is a web-based architectural style for creating web services. REST (Representational State Transfer) is a set of principles that govern how data are exchanged between clients and servers over HTTP protocol. RESTful APIs are designed to be simple and scalable, making it easy for developers to create web applications that can communicate with each other over the Internet.

RESTful APIs are based on resources identified by unique URIs (Uniform Resource Identifiers). These resources can be manipulated using standard HTTP methods such as GET, POST, PUT, and DELETE. One of the critical features of RESTful APIs is that they are stateless. Each request the client sends contains all the information needed to complete the request. The server does not maintain any client-specific information between requests, making it easier to scale the API and handle many requests. One of the critical characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616 (Rodriguez, 2008).

The HTTP standard defines eight different kinds of messages. These four are the most commonly used: GET - Get a representation of this resource. DELETE - Destroy this resource. POST - Create a new resource based on the given presentation underneath this one. PUT - Replace this resource's state with the one described in the given representation (*RESTful Web APIs [Book]*, 2013).

These methods or verbs are often interpreted as the standard CRUD operations - Create, Read, Update, and Delete - as depicted in Figure 1.

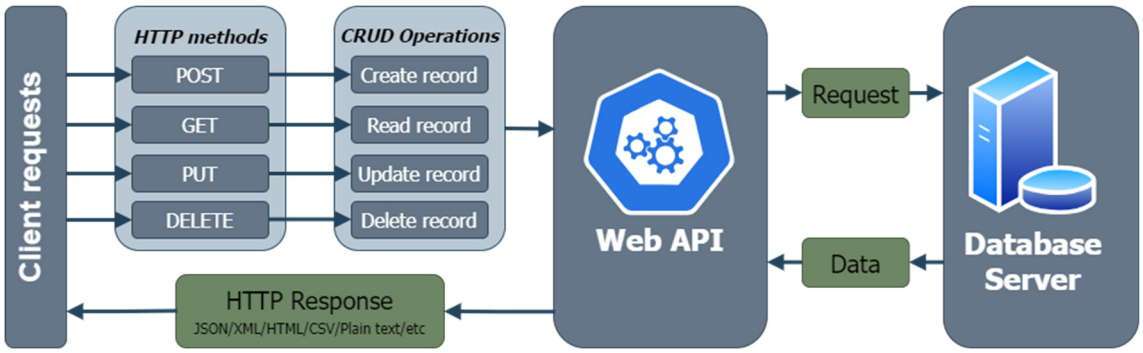

Figure 1 - Web API endpoints for CRUD operations

Related Work

Performance testing is a type of testing that aims to assess the responsiveness and stability of a system when subjected to a specific workload. Additionally, this form of testing can also aid in examining, evaluating, and validating other key quality attributes of the system, including scalability, reliability, and resource utilization (*The Art of Application Performance Testing, 2nd Edition [Book]*, 2014). This section provides a review and summary of the work of various researchers who have evaluated and investigated the performance of web servers, database servers, and Web APIs over the years.

In (Iyengar, MacNair and Nguyen, 1997), A. Iyengar et al. simulated a heavily loaded Web server. They determined the distribution of request latencies for different set parameters, where many pages are created dynamically. Web server performance is limited by the server's processing power, not the network.

In (Jain *et al.*, 2020), P. Jain et al. considered the performance metrics: Page Load Time, Start Render Time, Speed Index, and First CPU Idle. All except the Start Render Time may be applied to Web APIs. The Page Load Time determines the time taken to load every element on the website. Speed Index is the average time the different visible parts take to get displayed. First, CPU Idle is the point of time at which a page is least interactive, making the window capable enough to handle user input.

In (Bermbach and Wittern, 2016), David Bermbach et al. focus on two main qualities. Availability, where the web API can respond to the requests, and if it is or is not fully functional. The other quality is performance, split into latency and throughput. Latency describes the time between the start of a request at the client and the end of receiving a response from the client. Conversely, throughput represents the number of requests a web API handles at a given time.

In (Khan and Amjad, 2016), Rijwan Khan et al. emphasized the significance of performance testing for web applications, highlighting various testing processes such as load testing, soak testing, smoke testing, and stress testing. The focus of the paper is on the application of smoke testing to a developed web application. It emphasizes the responsibility of the tester to thoroughly assess all aspects of the software before delivery, ensuring the provision of error-free and reliable software to the customer.

In assessing Web-Based APIs' performance, drawing upon insights obtained from earlier research by the same authors is crucial. The study titled "Method for Evaluating the Performance of Web-Based APIs", by Godinho et al., established a test battery, employing specific open-source tools to appraise Web API performance. This study constitutes a foundational reference for our current research, validating its practical application in evaluating various technologies (Godinho *et al.*, 2024).

API Performance Tests

The set of tests used in this work to evaluate Web-Based performance was referenced in section

Related Work.

Load Testing focuses on evaluating the performance of your system concerning the number of concurrent users or requests per second. It can also be utilized to replicate a typical business day. The load test is possible to evaluate the present performance of the system during regular load scenarios.

Stress Testing is a form of load testing employed to establish a system's boundaries. Its objective is to validate the dependability and consistency of a system in harsh conditions. This test is conducted to ascertain how your design will function in extreme situations and determine the maximum capacity of your system concerning users or throughput. Also, identify the breaking point and failure mode of your system.

A Spike Test is a stress test that differs from traditional stress testing by rapidly increasing the load to extreme levels within a brief time frame. The objective of a spike test is to evaluate the system's ability to cope with sudden surges in traffic and identify any performance bottlenecks or issues that may arise. This test enables early detection of potential problems before they occur in a production environment and ensures the system can handle anticipated traffic levels.

Finally, Soak Testing is employed to verify the dependability of a system over a prolonged period under heavy load. This test confirms that the system does not experience any bugs or memory leaks that can lead to a crash or restart. To identify bugs related to race conditions that occur sporadically. Ensure that the allocated storage space for your database is not depleted for log files. Confirm that the services will continue to function after endless requests.

Technologies and Tools

There are many programming languages, different technologies, and frameworks for Web API development, as stated in section RESTful Web API. This work aimed to match up .NET (version 6) with Java, used to develop these Web APIs in both technologies, Object/Relational Mapping for the database interaction.

Object-Relational Mapping (ORM) encompasses solutions for mapping business objects to relational data by separating persistence concerns on a persistence layer as collections on object-oriented programming language (Yoder and Johnson, 1998). Developers can interact with databases using objects and a high-level object-oriented API rather than writing complex SQL code. These objects allow the application to create, read, update, and delete operations, commonly known as CRUD operations, that can be performed on the database using the object model of the program. ORM tools provide an abstraction layer between the application code and the database, allowing developers to work with data in a way that is independent of the underlying database structure and technology.

.NET

.NET is a software development framework created by Microsoft to build a wide range of applications, including web, mobile, desktop, gaming, and IoT. While versions 4.x and previous versions only supported Windows, .NET 5.0 introduced cross-platform support. With .NET 5, developers can create applications that run on multiple operating systems, including Windows, Linux, and macOS. Also, .NET 5 merged the features of .NET Framework, .NET Core, and Xamarin into a single framework, making it easier to develop and maintain applications.

Entity Framework

Entity Framework —(EF) is an ORM package produced by Microsoft that allows .NET applications to store data in relational databases, shown in Figure 2.

Figure 2 - Microsoft Entity Framework

EF supports different approaches for database access, like Code First, Database First, and Model First, which allows developers to choose the one that better suits their needs. EF also supports different types of databases, like SQL Server, SQLite, MySQL, and PostgreSQL.

Java

Java is a class-based, object-oriented, programming language and computing platform designed to have as few implementation dependencies as possible. It is a cross-platform language, meaning compiled Java code can run on all platforms supporting Java without recompilation. Java is one of the most popular programming languages in use today.

This work uses Java Spring Boot, an open-source Java framework that makes creating standalone, production-grade, Spring-based applications with minimal configuration and boilerplate code easier. Spring Boot makes it easy to create standalone, web-based, and micro-services-based applications that require minimal configuration, reducing the time and effort required for setup and development.

Java Persistence API

Java Persistence API (JPA) is a Java specification for managing, persisting, and accessing data between Java objects/classes and a relational database. It is a part of the Java Enterprise Edition (Java EE) platform and provides a standard way to interact with databases in a Java environment. JPA is similar to Microsoft's Entity Framework in the .NET framework. JPA provides a set of annotations and interfaces that can be used to define mappings between Java classes and database tables. It also provides a powerful and flexible query language, called the Java Persistence Query Language (JPQL), that allows developers to retrieve and manipulate data in a way that is similar to querying in-memory objects. JPA is a specification; therefore, it is implemented by different providers, such as Hibernate, EclipseLink, and OpenJPA, which provide their implementation of JPA; this allows developers to choose the one that better suits their needs. In this work, Hibernate will be used as a provider. In a relational database, the connection between the application code and the database will be handled by Java Database Connectivity (JDBC), shown in Figure 3.

Figure 3 - JPA and the Java ORM layer

Tools

For monitoring, stats, and dashboards creation tools for this work, it was used a combination of Prometheus, Fluentd, and Grafana.

Prometheus (Turnbull, 2018) is an open-source monitoring and alerting platform that provides a multi-dimension data model by collecting data in the form of time series from data sources. Prometheus employs a pull model over HTTP to manage the real-time metrics in a time series database and utilizes PromQL to enable flexible queries and real-time alerting. In contrast to blackbox monitoring, as performed by Nagios/Icinga, suitable only for classical admin jobs, Prometheus promotes a whitebox monitoring approach, thus aiding in administering the internal specifics about the state of the micro-services (Sukhija and Bautista, 2019). It also includes built-in alerting and visualization capabilities that allow users to set up alerts and visualize their data. Overall, Prometheus is a popular and highly versatile tool for monitoring and analyzing system and application performance (Coarfa, Druschel and Wallach, 2006). Prometheus doesn't allow splitting logs into different fields, on this work, into different APIs, but Fluentd provides this work.

Fluentd (Ismail *et al.*, 2017) is an open-source data collector that is used to unify logging data and other time-series data from various sources in real time. It is designed to handle large volumes of data and can route data to multiple destinations, including storage systems, message queues, and analytic tools. Fluentd can collect data from various sources, including logs, events, and metrics, and send them to Prometheus.

Grafana (Chakraborty and Kundan, 2021) is an open-source analytics and visualization platform that allows you to create customizable dashboards for monitoring and analyzing data from various sources. It provides a centralized platform for creating and sharing interactive, real-time, visualizations, alerts, and panels that make it easy to understand and monitor complex data.

For testing the API, the tools used were: curl and k6 (*k6 Documentation*, no date). cURL is a command-line tool for transferring data over various protocols, including HTTP, FTP, and SMTP. Using cURL can send HTTP requests to web servers and receive responses, which helps test and debug web applications and APIs. Hey is an open-source HTTP load-testing tool simulating web applications or API traffic, allowing small scripts to test the performance and scalability of web services quickly.

Test Scenario

The test scenario was done by installing a virtual machine (VM) on Debian Linux and cloning twice to have the same base. The head node uses 8 CPUs and 4 GB of RAM, while the other two nodes also use 4 CPUs but only 2 GB of RAM.

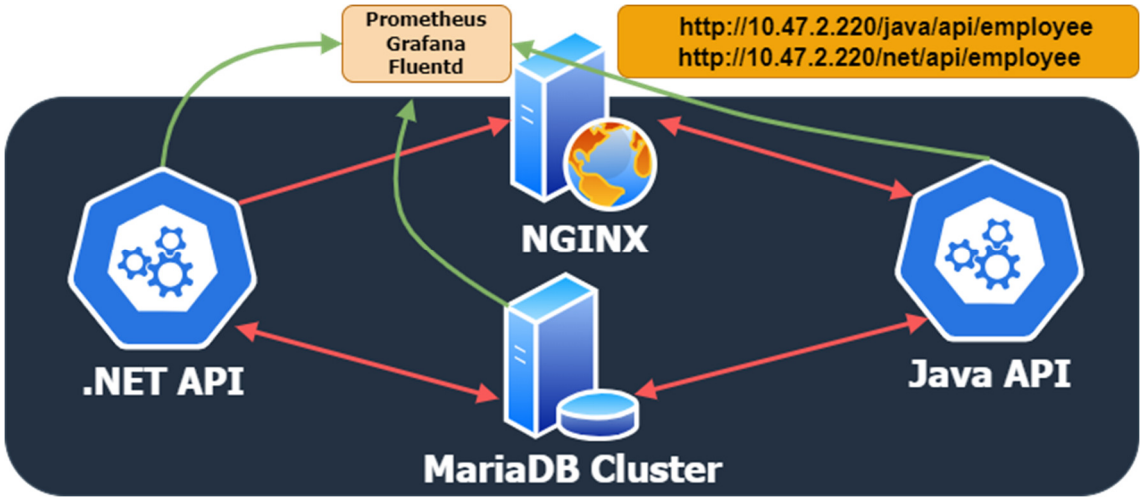

Figure 4 - Test scenario

On the head node, Nginx was installed to act as a reserve proxy solution to access the API's. It was installed and configured on the same node as Prometheus, Grafana, and Fluentd. On the second node was installed .NET 6 SDK, and the .NET API was compiled and set up as a service. Finally, Java SDK was installed on the third node, and the Java API was configured as a service, similar to the process on the other API node. For the database connections and to provide a real-life scenario, there were used the existing clusters with MariaDB and Microsoft SQL Server (Figure 4).

Using Nginx as a reserve proxy, any request to the URI '/net/api/employee' is redirected to the .NET node, and the requests with the URI '/net/api/employee' are directed to the Java node, using their IP's addresses. Both API nodes have a Prometheus node exported installed to allow the central node to collect information about the system (memory and CPU).

The database used for this work contains two tables. The first is the data about the entity Department and the other about the Employee. The Employee table has a foreign key that refers to the primary Id of the Department, shown in Figure 5. The design for this database was that a request to the API employees would require the department name.

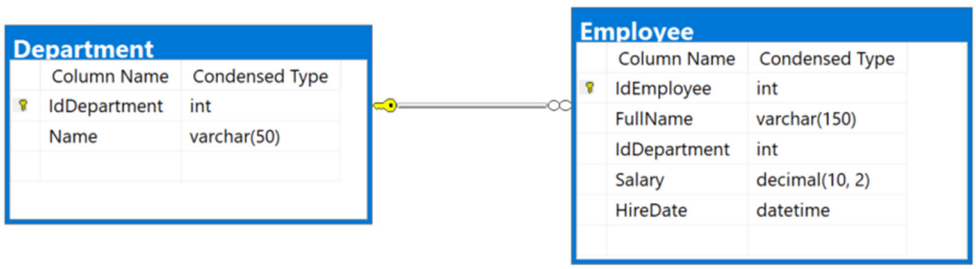

Figure 5 - Database diagram

Developing the two Web APIs to work with the same database was challenging. Using ORMs improves productivity, but the developers lose some control over the queries to the databases. The way to ensure that both APIs made the same requests was to force the Java API to use the same queries that the .NET has done. The .NET API running on debug prints out the queries to the console. That query was then used on the JPA repository implementation on Java, as shown in Figure 6.

```java
@Repository
public interface EmployeeDBRepository extends JpaRepository<EmployeeDB, Long> {

    1 usage
    @Query(value = "SELECT e.IdEmployee, e.FullName , e.IdDepartment as iddepartment, d.Name as namedepartment,e.Salary, e.HireDate\n" +
            "FROM Employee e LEFT JOIN Department AS d ON e.IdDepartment = d.IdDepartment\n" +
            "ORDER BY e.IdEmployee ",
            nativeQuery = true)
    List<EmployeeDB> findAllEmployees();

    1 usage
    @Query(value = "SELECT e.IdEmployee, e.FullName , e.IdDepartment as iddepartment, d.Name as namedepartment,e.Salary, e.HireDate\n" +
            "FROM Employee e LEFT JOIN Department AS d ON e.IdDepartment = d.IdDepartment\n" +
            "WHERE e.IdEmployee = ?1 ORDER BY e.IdEmployee",
            nativeQuery = true)
    EmployeeDB findEmployeeById(Long Id);
}
```

Figure 6 - Java Employee JpaRepository - from Intellij IDEA

Virtual Users (VUs)

A web API can be made available to clients through a web server or reverse proxy, which acts as a gateway to route incoming requests to the appropriate API endpoints and return responses to clients. Implementing these solutions can provide additional functionality like load balancing, caching, and security features that enhance API performance and security. Among the most popular web server and reverse proxy solutions are NGINX and Apache Web Server.

The maximum number of concurrent connections for Apache2 is determined by the MaxClients directive in its configuration file. The default value is 256, but it can be adjusted according to specific requirements. On the other hand, the maximum number of concurrent connections for NGINX is set by the worker_connections directive in its configuration file. By default, NGINX can handle up to 512 connections per worker process, and this value can be increased to a maximum of 1024 connections per worker process. Assuming that at least two workers are used, the number of allowed connections can be up to 2048.

The default user values for K6's tests are set at modest levels, with the stress and spike tests initially configured for 40 and 140 users, respectively. These default values require minimal CPU and memory resources. However, preliminary testing determined that a load test with 100 users would be most suitable for this specific project. For the stress test, the number of users was increased in steps of 200, with load levels set at 100, 300, 500, and 700 users. The peak load for the tests was placed at 1500 users, which represents 75% of the NGINX's allowed connections capacity.

Results

Before running the initial tests on section API Performance Tests, the first results were obtained via Grafana while the nodes ran idle without requests. Within the first hours observed in Figure 7, the Java node required more than double the RAM compared with the .NET node.

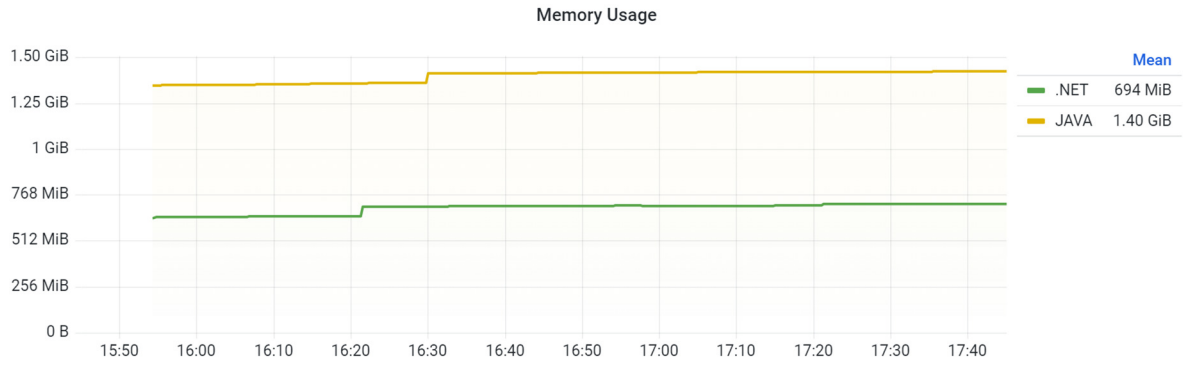

Figure 7 - API - memory consumption idle

The test battery will be repeated using the GET, POST, and PUT HTTP methods. Tests are run in different APIs, but the same test will use the same parameters, duration, and VUs.

GET HTTP method

GET is used to request data from a specified resource. This test is used to retrieve the list of departments and employees.

1. GET - Load testing

The load test to the API used the K6 tool. It is a 20 minutes test, starting from 0 VUs to 100 in 5 minutes, keeping the 100 VUs for 10 minutes, and then 5 minutes to cool down until 0. The memory requirements had grown constantly on both APIs, where the Java API also kept requiring from 30 to 40% more RAM (Figure 8). The opposite happens with CPU requirements. Here, .NET API is more demanding, requiring less than 5% over Java API.

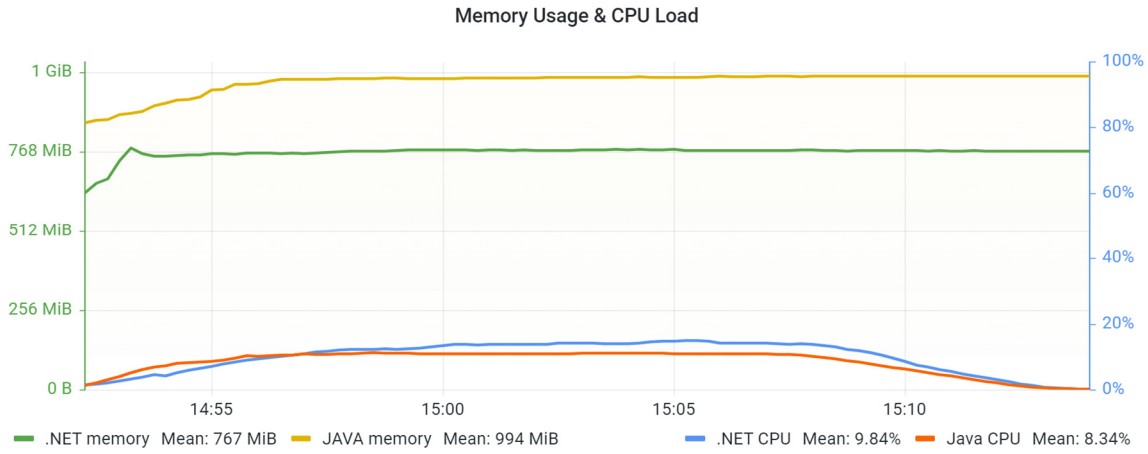

Figure 8 - GET load testing - memory and CPU

The first three requests to the .NET API are always extremely slow compared to any other during the 20-minutes request. The test was restarted several times to verify this behavior. The 99% percentile provides a better image of the response time over time. But the results with the demeanor on the first requests could mislead the reading of the results chart (Figure 9).

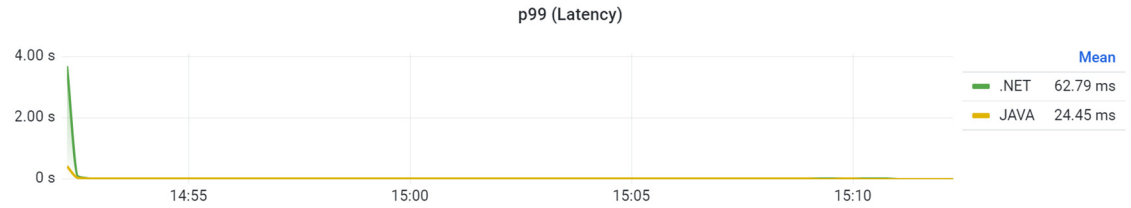

Figure 9 - GET load testing - latency 99% percentile

On the 90% percentile, it is possible to verify that both APIs performed without issues during the load testing, and the difference between them is so tiny that they are virtually identical, on Figure 10 confirmed by the results in Table 1.

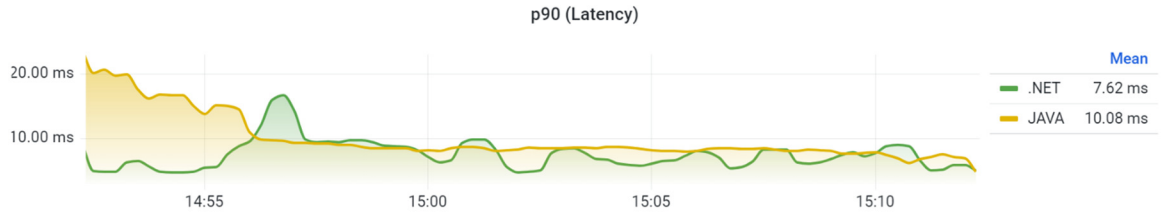

Figure 10 - GET load testing - latency 90% percentile

Table 1 - K6 GET load testing average results

	http req duration (ms)				***http req***	
API	***avg***	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	4.53	9.75	13.69	0.00%	89618	74.63/s
Java	4.44	8.83	12.4	0.00%	89630	74.66/s

2. GET - Stress testing

The stress testing applied was a 38 minutes test. In the first 2 minutes, the number of VUs will rise from 0 to 100 and remain for 5 minutes. Over the next 2 minutes, the number of VUs will increase to 300 and stay for 5 minutes. Using the same time intervals, the number of VUs will reach 700 and, after, will start to cool down, taking 10 minutes to get to 0.

The memory requirements were constant on both APIs, and the Java API also required 35 to 40% more RAM (Figure 11). On the CPU requirements, the .NET increased the demand compared to the Java API, requiring more CPU on average 15% but raising to 30% when the peak of 700 VUs happens.

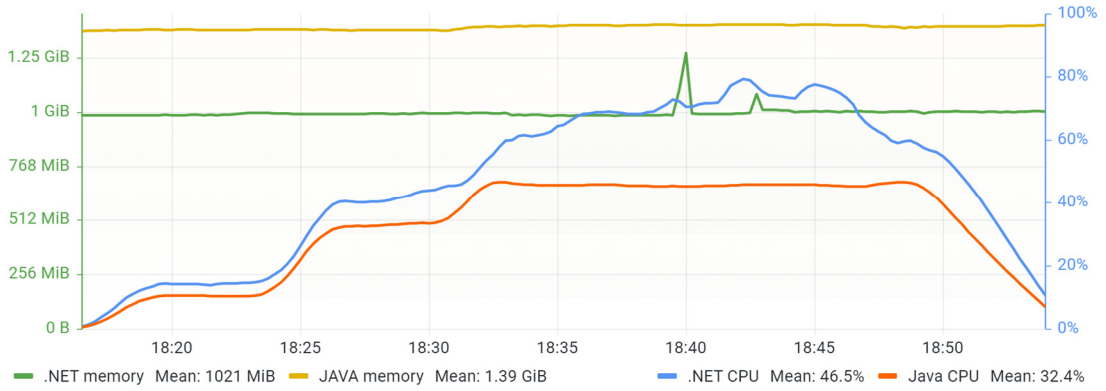

Figure 11 - GET stress testing - memory and CPU

While analyzing the performance of this test, it is possible to understand that the .NET API outperforms the Java API by a large margin. It is also observable that the Java API response times decay after the 300VUs, with slow response times. On the other hand, the .NET API shows a consistent and stable performance throughout the test. It is possible to observe the time differences between both solutions in Figure 12, where the latency of the .NET API is represented in green and blue, and the latency of the Java API is represented in yellow and red.

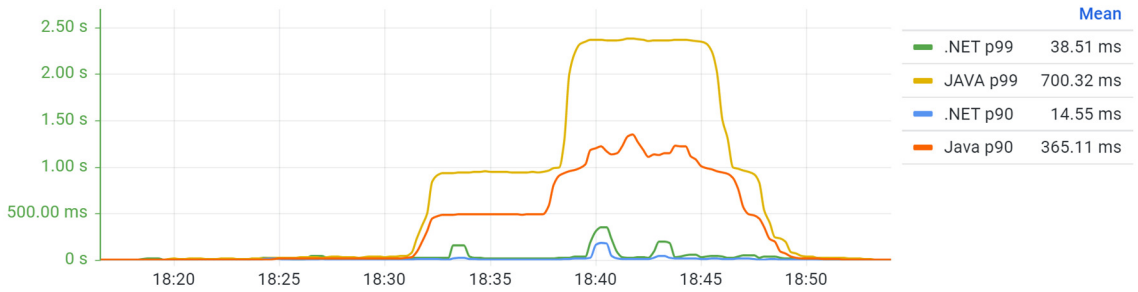

Figure 12 - GET stress testing - latency

Table ***2*** clearly illustrates the superiority of the .NET API in terms of performance and stability. Still, there were no errors while accessing both APIs.

Table 2 - K6 GET stress testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	9.37	13.39	20.35	0.00%	1662746	729.26/s
Java	328.59	886.79	931.82	0.00%	617048	270.53/s

IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

3. GET - Spike test

The spike testing applied was a 7 minutes and 40 seconds test. In the first 10s, the number of VUs will rise from 0 to 100 and will remain for 1 minute. Over the next 10 seconds, the number of VUs will increase to 1500 and stay for 3 minutes. In the next 10 seconds, the number of VUs lowers to 100 and will remain for 3 minutes. The test will finish after 10 seconds when the number of VUs reaches 0. In terms of hardware requirement (Figure 13), and comparing to the stress test results in section

GET - Stress **testing**, surprisingly, the .NET API had the same memory consumption. On the other hand, the Java API required more than 15% of memory.

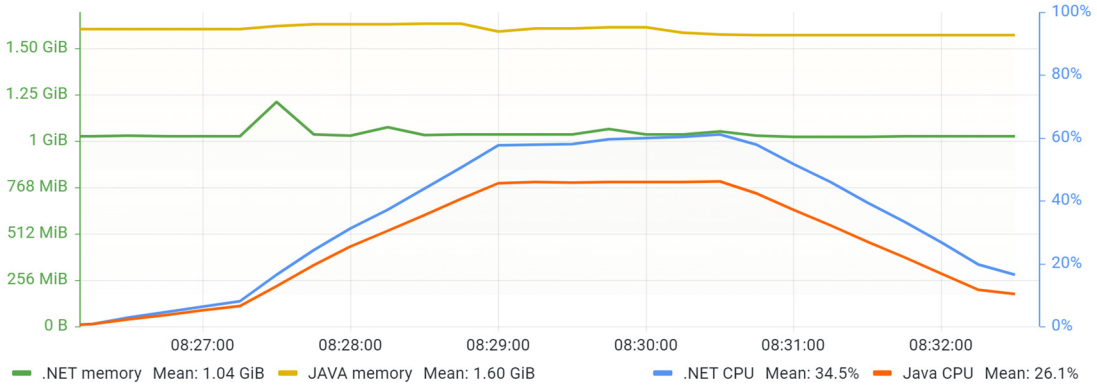

Figure 13 - GET spike testing - memory and CPU

The difference in CPU requirements follows a similar behavior, and the difference between both APIs was negligible when compared with the stress test. It can be observed in Figure 14 that, applying a more demanding test, the performance of Java has degraded. It appears that the Java API cannot handle the increased load from the more challenging test, as evidenced by the degradation in performance. The increase in average response time from milliseconds to seconds and the high response time at the 99th percentile (over 2.6s) indicates that the system is struggling to keep up with the increased demand.

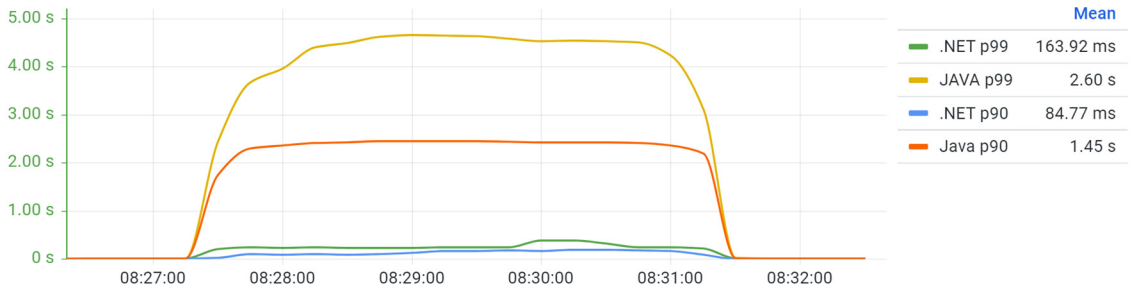

Figure 14 - GET spike testing - latency

By default, the NGINX has defined the works connections to 1024. While running this test the first time, around 1% of the requests could not reach the APIs since, at the peak of the test, there were 3000 simultaneous VUs. Even with a few requests blocked, the test was repeated, directly accessing the endpoints, removing the man in the middle (NGINX), and confirming the results from Figure 14. The test confirmed that the errors were, in fact, from NGINX, and no errors occurred using direct access to the .NET and Java API nodes. The results from

Table ***3*** confirmed that Java API couldn't respond with the same performance as the .NET API.

Table 3 - K6 GET spike testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	198.07ms	475.03	527.76	0.00%	518126	1124.41/s
Java	1.83s	2.58s	2.7s	0.00%	217808	472.75/s

From the same table, it is possible to verify that the .NET can respond to almost two times and a half requests in the same period.

4. GET - Soak testing

The spike testing applied was a 7 minutes and 40 seconds test. In the first 10s, the number of VUs will rise from 0 to 100 and will remain for 1 minute. Over the next 10 seconds, the number of VUs will increase to 1500 and stay for 3 minutes. In the next 10 seconds, the number of VUs lowers to 100 and will remain for 3 minutes. The test will finish after 10 seconds when the number of VUs reaches 0.

The results of the soak test were quite clear-cut. The .NET API performed exceptionally well across all metrics, even where it showed the least favorable outcomes, such as CPU usage. The difference was minimal compared to the Java API. Additionally, it can be observed from Figure 15 that the RAM usage of the .NET API remained stable throughout the test. In contrast, the Java API's RAM usage continued to increase until it reached nearly 1.5 GB. RAM requirements highlight the difference in resource usage efficiency between the two APIs and are essential when choosing an API for a particular use case.

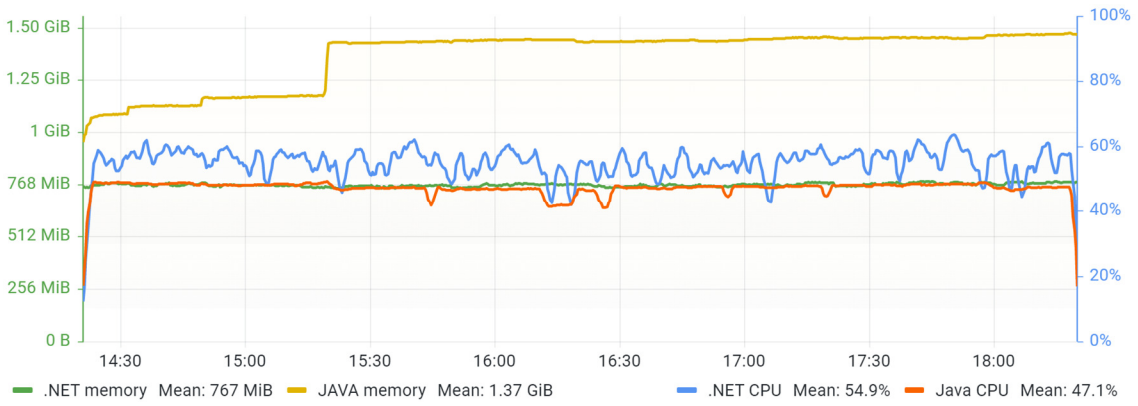

Figure 15 - GET soak testing - memory and CPU

This test made it evident that there was a marked contrast in the performance of the .NET and Java APIs. The .NET API displayed a steady and consistent performance throughout the test. At the same time, the Java API was prone to fluctuations and had the lowest overall performance, as depicted in Figure 16.

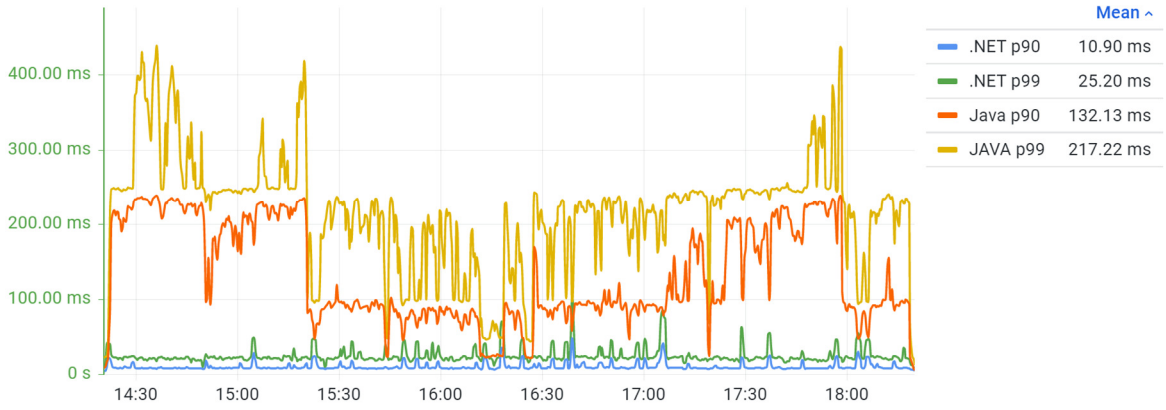

Figure 16 - GET soak testing - latency

This test was the most extensive test applied to the work scenario. The results from

Table ***4*** show that .NET API could handle more than twice the number of requests the Java API could run within a 4-hour. Additionally, the average response time for the Java API was ten times slower than that of the .NET API.

Table 4 - K6 GET soak testing average results

API	http req duration (ms)		http req

	avg	*p(90)*	*p(99)*	*failed*	*total*	*per second*
.NET	6.5	10.38	13.96	0.00%	5667572	393.56/s
Java	61.55	125.96	148.82	0.00%	5336488	370.58/s

POST HTTP method

POST is used to submit an entity to the specified resource, often causing a change on the server. This test is used to add a new employee.

1. POST - Load testing

The memory requirements on both APIs are similar to the GET method; the Java API also kept requiring more RAM (Figure 17), more than the double. Compared with the GET method CPU requirements, both APIs require almost the same percentage of CPU, but now JAVA is more demanding and requires around 4% more.

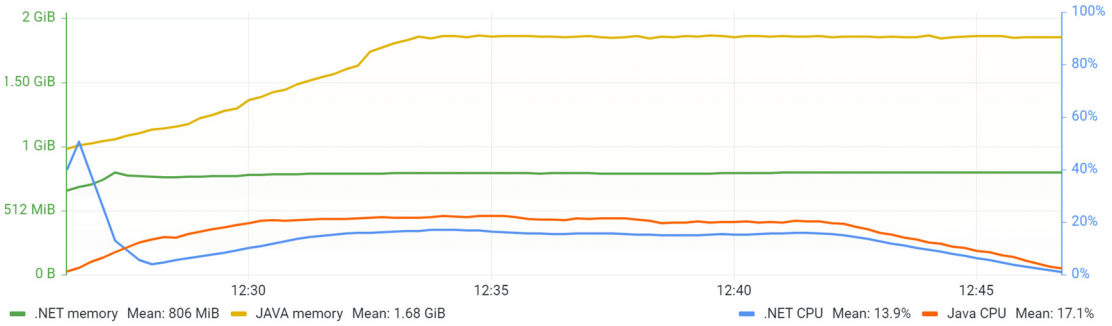

Figure 17 - POST load testing - memory and CPU

On the GET method, we observed that the .NET API had the first requests extremely slow, and, on the POST method, it is possible to watch the same behavior. On the 90th, the .NET API performs better by a small margin.

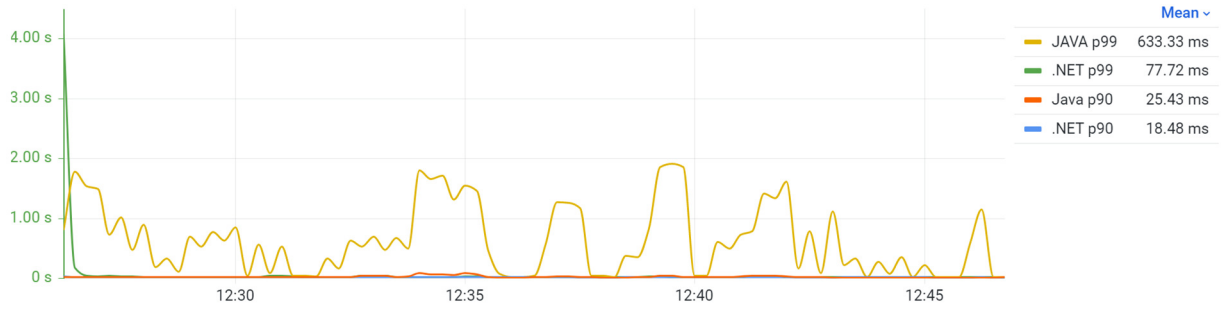

Figure 18 - POST stress testing - latency

But with a closer look using the 99th, it is possible to observe the erratic performance on the Java API (Figure 18), also confirmed by the results in Table ***5***. The most critical parameter on this test was the percentage of HTTP requests that failed, with 15.43%.

Table 5 - K6 POST load testing average results

API	http req duration (ms)				http req	
	avg	*p(90)*	*p(99)*	*failed*	*total*	*per second*
.NET	7.67	16.69	23.45	0.00%	89299	74.36/s
Java	8.22	38.7	837.08	15.43%	87661	73.01/s

2. POST - Stress testing

Both APIs had consistent memory requirements with the previous test. The Java API required twice as much RAM as the .NET API (as shown in Figure 19) with similar results for the GET method. In terms of CPU requirements, both APIs showed very close results with only minimal differences.

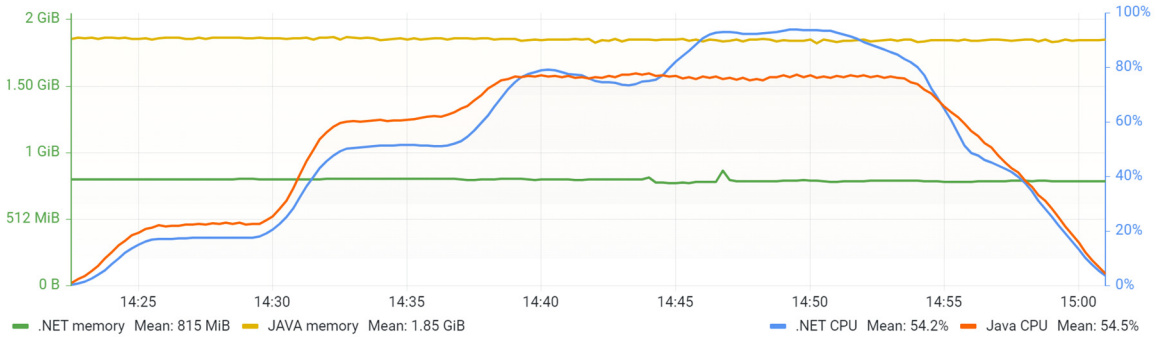

Figure 19 - POST stress testing - memory and CPU

IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

Using POST requests, the results obtained when compared to the GET method are identical, shown in Figure 20, where the .NET API latency was less than a half of the Java API. Both APIs have peaks of latency, but, while the .NET API followed the curve of the VUs, the Java API had an unstable behavior, and, when looking at the chart, almost a line more visible on the 99-percentile curve.

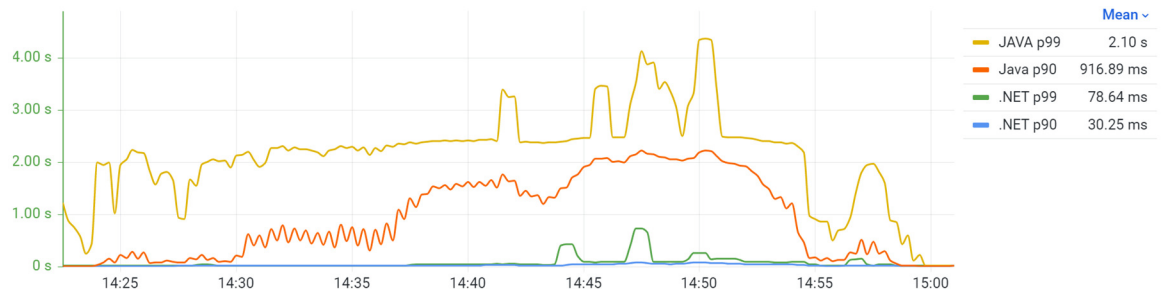

Figure 20 - POST stress testing - latency

Table ***6*** provides additional support for the previous results that indicate the Java API under-performed on this test. The response times rise from milliseconds to seconds on the Java API, with a 38.60% of HTTP requests failed.

Table 6 - K6 POST stress testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	13.45	52.19	94.26	0.00%	823521	361.17/s
Java	264.82	1740	2280	38.6%	572631	251.09/s

3. POST - Spike test

In terms of hardware requirement, and compared to the results of the GET method, the .NET API requires more CPU. At the peak of the test, the .NET API required around 95% while the Java API was 75%. On the other hand, Java demanded more RAM, almost 70%, while .NET performance was similar to the GET method. On the spike test using the POST method, the RAM consumption of the .NET API was identical to the GET method, while the Java API had an increase of 15%, as shown in Figure 14.

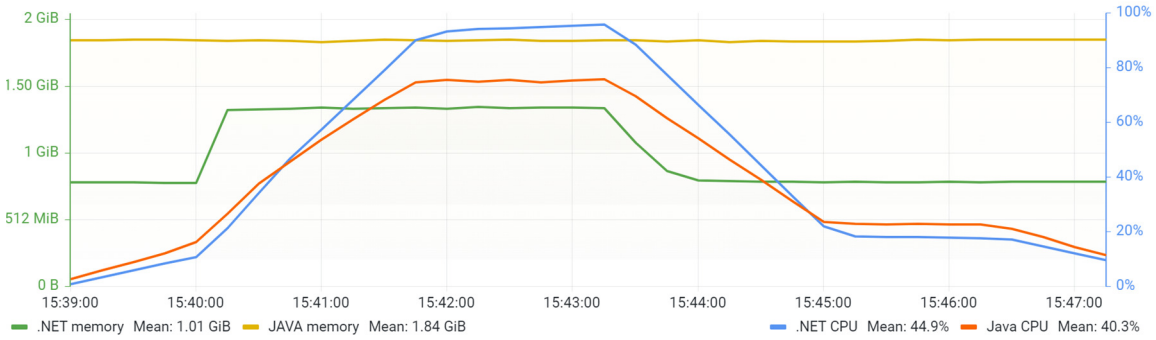

Figure 21 - POST spike testing - memory and CPU

The results from **Table 7** and on Figure 22 confirmed that .NET API had better performance than Java API. Figure 22 also indicates that the behavior of both API's is according to the number of the VUs, as it was expected and different from the GET method. Once again, the number of HTTP requests that failed is considerable.

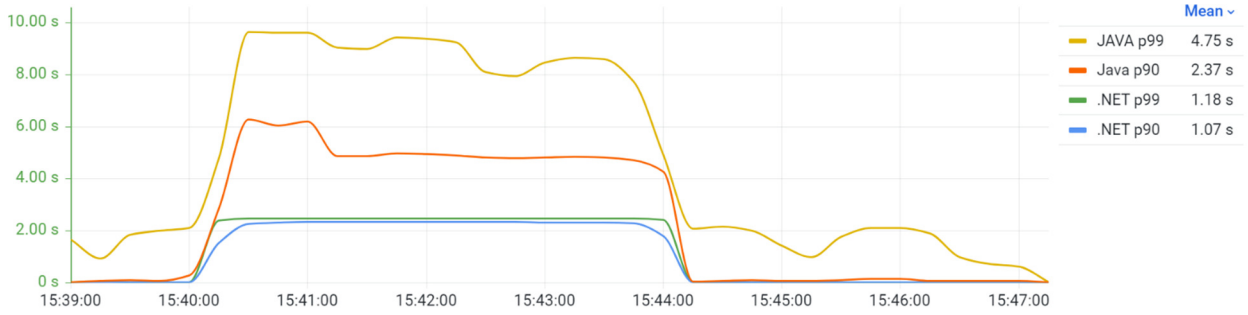

Figure 22 - POST spike testing – latency

Table 7 - K6 POST spike testing average results

API	http req duration (ms)				http req	
	avg	*p(90)*	*p(99)*	*failed*	*total*	*per second*

.NET	12.26	28.34	43.4	0.00%	5629988	390.96/s
Java	125.66	1330s	1790s	36.41%	4413447	306.48/s

4. POST - Soak testing

Looking at the results of the soak test, again both APIs were consistent through the test. On this test, .NET clearly performed better than Java API, requiring 17,5% less RAM and 57% of the CPU requirements, as shown in Figure 23.

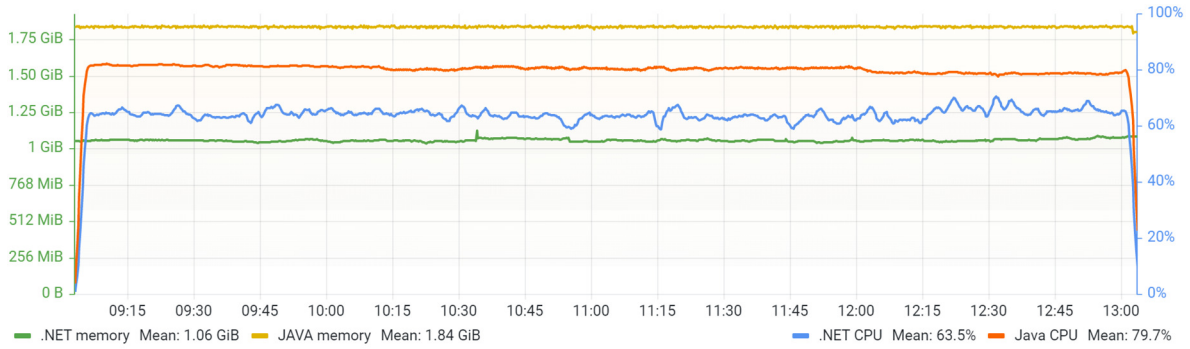

Figure 23 - POST soak testing - memory and CPU

The results indicate that when using the POST method, the .NET API performed better than the Java API, with the difference in performance becoming more pronounced when using the 99th percentile (as seen in Figure 24). It's worth noting that the results of only the 90th percentile could be misleading, as the Java API had unstable response times.

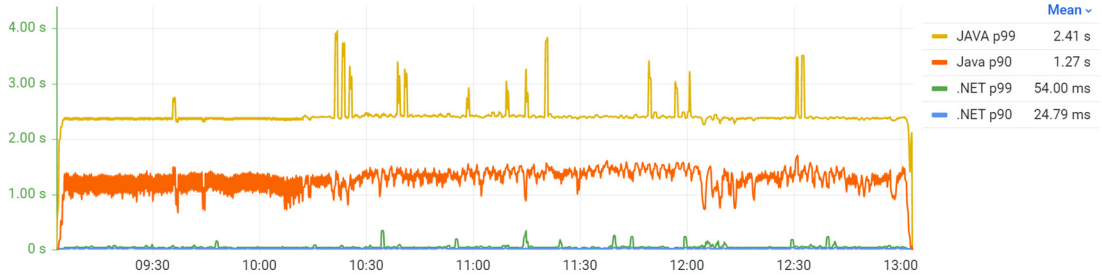

Figure 24 - POST soak testing - latency

Table ***8*** highlight variations in latency values, where the unit used for the .NET is milliseconds and for the Java is seconds. Also, very significantly Table ***8*** reveals the discrepancy in latency values and, crucially, a 36%+ error rate. The findings in

on the Java API, a concerning error rate exceeding 36%. This high number of errors could point towards a coding error or platform bug and thus warrant further examination.

IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

Table 8 - K6 POST soak testing average results

API	http req duration (ms)				http req	
	avg	*p(90)*	*p(99)*	*failed*	*total*	*per second*
.NET	12.26	28.34	43.4	0.00%	5629988	390.96/s
Java	125.66	1330s	1790s	36.41%	4413447	306.48/s

PUT HTTP method

The PUT request is employed to alter a resource on the server by modifying an entity. In this particular test, it was used to update an existing employee. The request must include a payload containing all required data fields and the header must be set to "Content-Type: JSON" for the request to be valid.

1. PUT - Load testing

On this test, both APIs are again similar to the GET and POST methods, where the Java API also kept requiring more RAM (Figure 25), while the differences in CPU requirements are minimal.

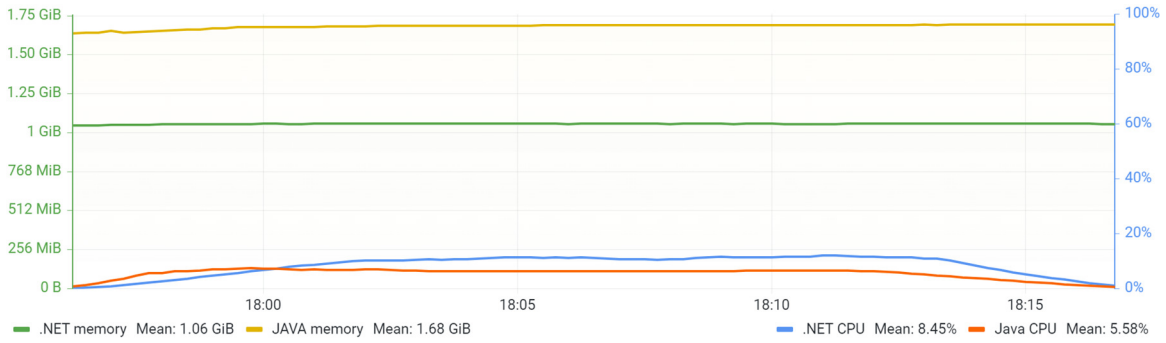

Figure 25 - PUT load testing - memory and CPU

The latency results show that both Web APIs performed similarly throughout the test, as portrayed in Figure 26. On the previous methods, the gap in performance between both APIs was significant.

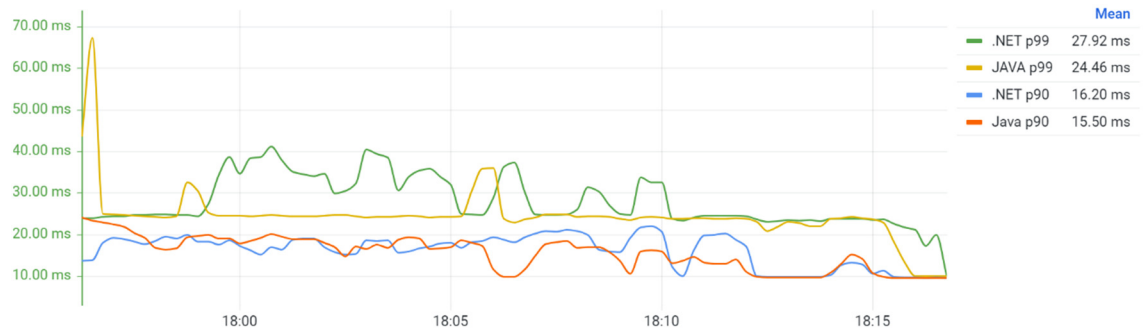

Figure 26 - PUT load testing - latency

The findings in

Table ***9*** and the chart in Figure 26 demonstrate similar results for the average of HTTP requests, and, therefore, the total number of requests and requests per second.

Table 9 - K6 PUT load testing average results

API	http req duration (ms)				http req	
	avg	*p(90)*	*p(99)*	*failed*	*total*	*per second*
.NET	8.31	17.34	25.95	0.00%	89290	74.35/s
Java	8.44	13.81	18.91	0.00%	89327	74.40/s

2. PUT - Stress testing

On the stress test, the Java API requires almost half of the CPU, as shown in Figure 27, while the Java API also kept requiring more RAM (Figure 25), while .NET API requires around 60% of the RAM, similar with the GET and POST methods

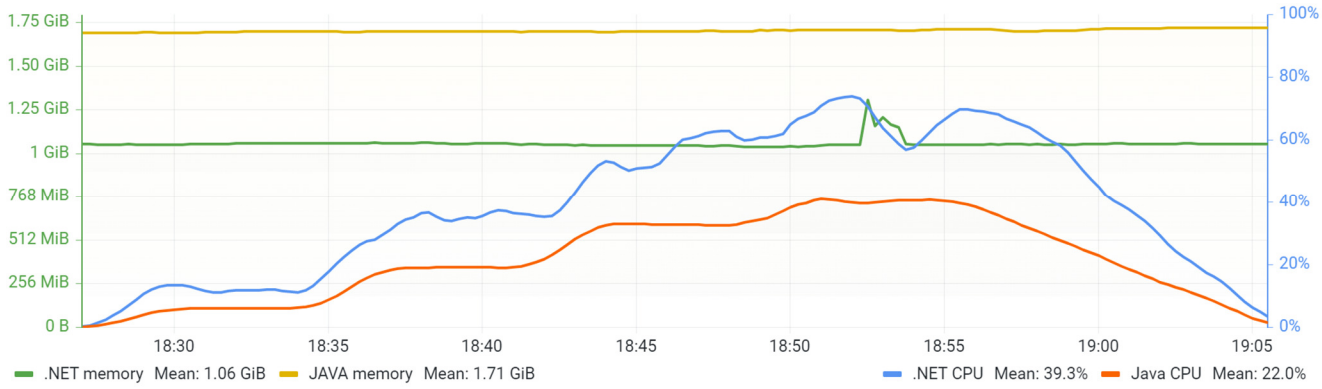

Figure 27 - PUT stress testing - memory and CPU

IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

Using PUT requests, the results were different from all test run at this point. The Java API performed regularly with around 50% of the latency when compared with the .NET API. The .NET API had demonstrated some instability, shown in Figure 28.

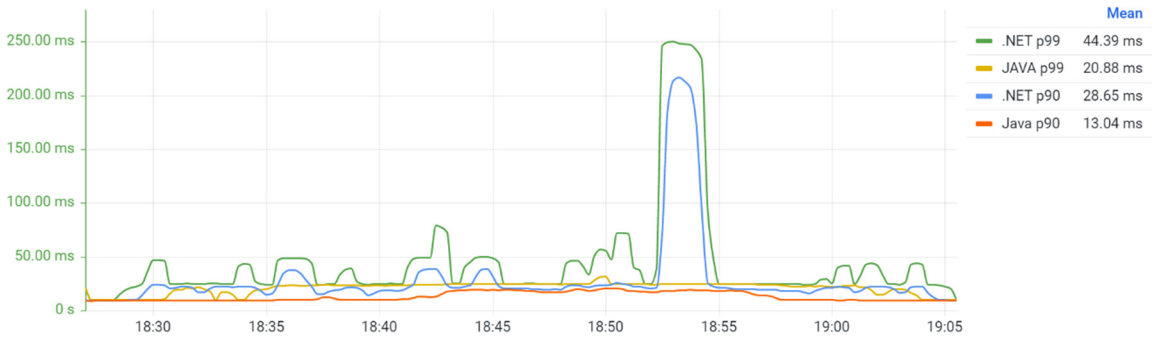

Figure 28 - PUT stress testing - latency

Table ***10*** confirmed the results that indicate the .NET API under-performed on this test. Both APIs haven't failed HTTP requests.

Table 10 - K6 PUT stress testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	10.04	31.85	177.69	0.00%	826365	362.30/s
Java	8.54	13.64	18.83	0.00%	832346	364.97/s

3. PUT - Spike test

In terms of hardware requirement comparing to the results of the GET and POST methods, both had similar behaviour. The .NET API requires about 60% of RAM, but more 70% in CPU, as shown in Figure 29.

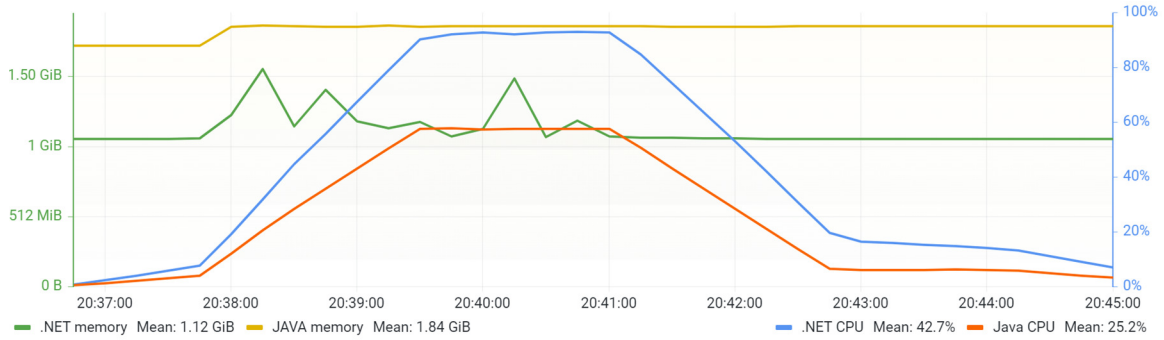

Figure 29 - PUT spike testing - memory and CPU

On the spike test, the results were different from the stress test. On this test, the .NET API latency was regular without fluctuations and with values around 40% of the Java API, as shown in Figure 30.

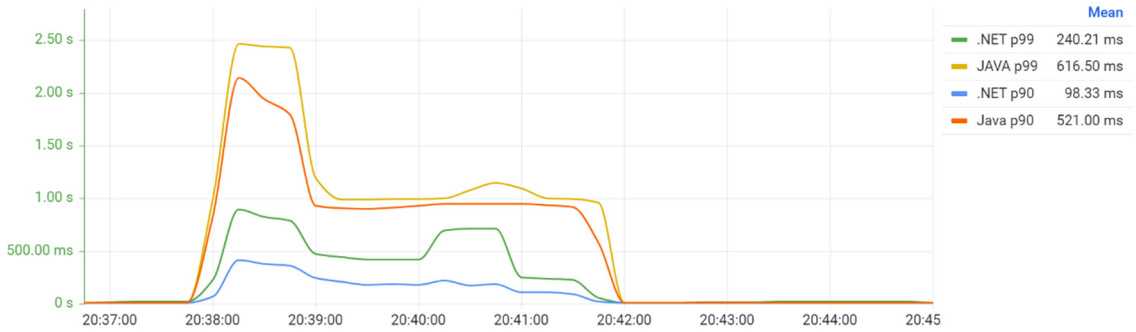

Figure 30 - PUT spike testing - latency

Table ***11*** confirms that the .NET API performed significantly better than the Java API, particularly at the 90th percentile. Both APIs had no HTTP request errors.

Table 11 - K6 PUT spike testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	28.83	279.58	491.06	0.00%	291490	632.59/s
Java	546.91	987.81	1200	0.00%	204564	444.25/s

IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

4. PUT - Soak testing

Upon reviewing the results of the soak test, it is evident that both APIs demonstrated consistency throughout the test. However, the .NET API required 40% less RAM but had a higher CPU requirement of 90%, as illustrated in Figure 31. It is worth noting that the behavior of the .NET API differed from the other tests, with a fluctuating CPU chart line throughout the test, although the range was not significant.

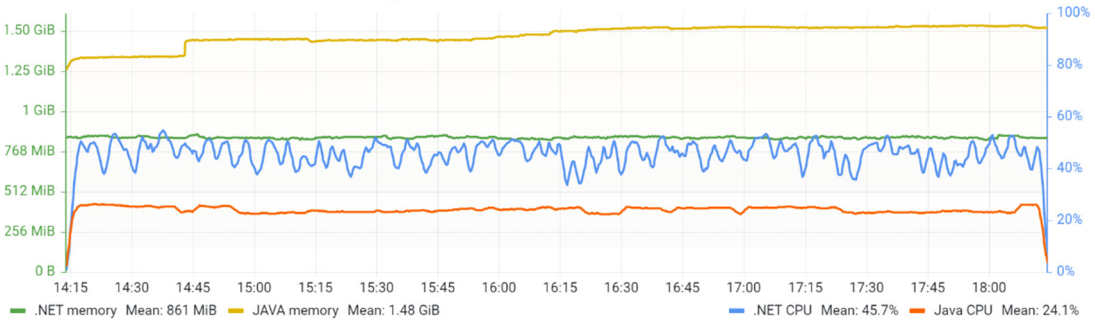

Figure 31 - PUT soak testing - memory and CPU

Regarding latency, both APIs delivered comparable results, as evidenced by Figure 32 and

Table ***12***, whether considering the average values, total HTTP requests, or requests per second.

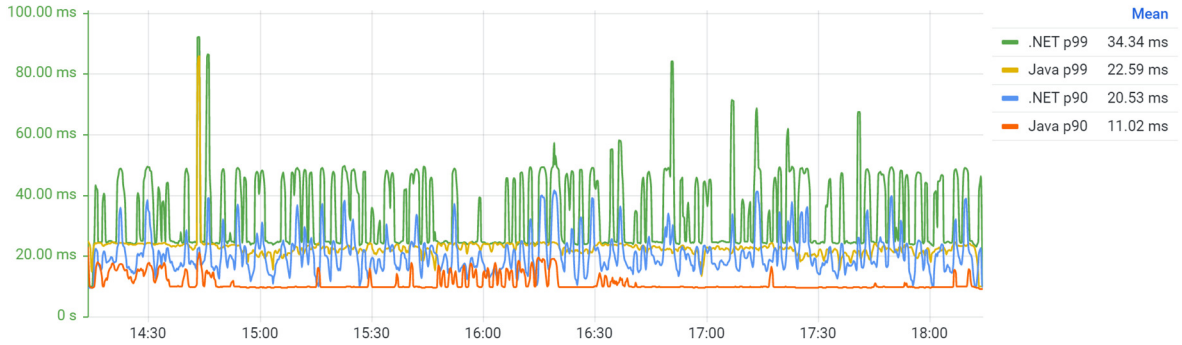

Figure 32 - PUT soak testing – latency

Table 12 - K6 PUT soak testing average results

API	http req duration (ms)		http req

Cite this Article as: Antonio GODINHO, Jose ROSADO, Filipe SA and Filipe CARDOSO (2024)," Performance Comparison of RESTful Web APIs using a Test Suite: .NET vs. Java Spring Boot ", Journal of Software & Systems Development, Vol. 2024 (2024), Article ID 478010, https://doi.org/10.5171/2024.478010

	avg	*p(90)*	*p(99)*	*failed*	*total*	*per second*
.NET	9.21	19.21	34.45	0.00%	5644050	391.94/s
Java	7.7	10.52	15.36	0.00%	5662375	393.21/s

DELETE HTTP method

The DELETE method removes a resource on the server, such as an existing employee, in this test. The request should contain a payload with all relevant data fields, and the header must be set to "Content-Type: JSON".

1. DELETE - Load testing

APIs' memory and CPU requirements are similar to the GET and PUT methods. The .NET API kept requiring around 60% RAM (Figure 33) of the Java API.

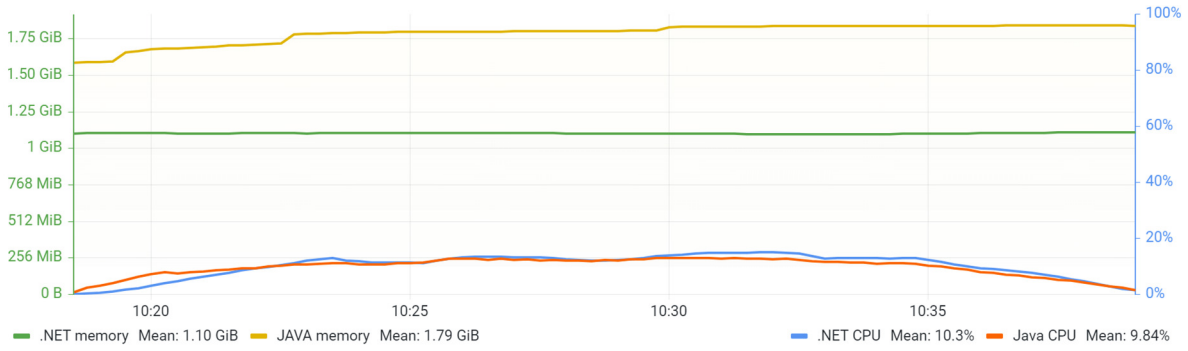

Figure 33 - DELETE load testing - memory and CPU

For the DELETE requests, the results were different from the other methods. Even with many VUs, the chart lines were constant through the test, except for two peaks on both APIs, shown in Figure 36. Those simultaneous peaks of both APIs may indicate a database delay and cannot be attributed to the APIs. With the obtained results, the .NET was faster across the load test, as depicted in Figure 34.

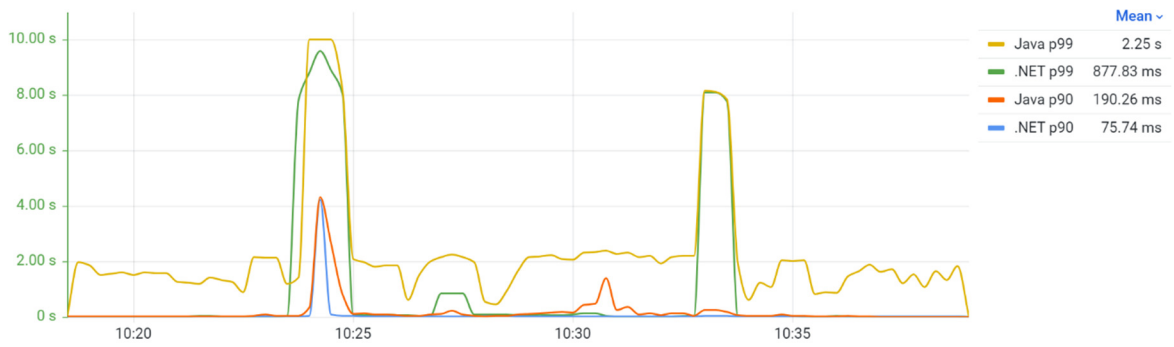

Figure 34 - DELETE load testing - latency

Table *13* and the chart in Figure 34 reveal comparable findings, including over 50% of errors on HTTP requests. These results indicate that the Java API struggled to keep up with the HTTP requests.

Table 13 - K6 DELETE load testing average results

API	http req duration (ms)				http req	
	avg	*p(90)*	*p(99)*	*failed*	*total*	*per second*
.NET	9.95	22.6	100.57	0.00%	86091	71.72/s
Java	6.63	34.76	1110	50.69%	81751	68.12/s

2. DELETE - Stress testing

On the stress test, the Java API had better performance in terms of CPU and also performed better than the other methods. However, once again, the .NET API required only 60% % of RAM (as shown in Figure 35).

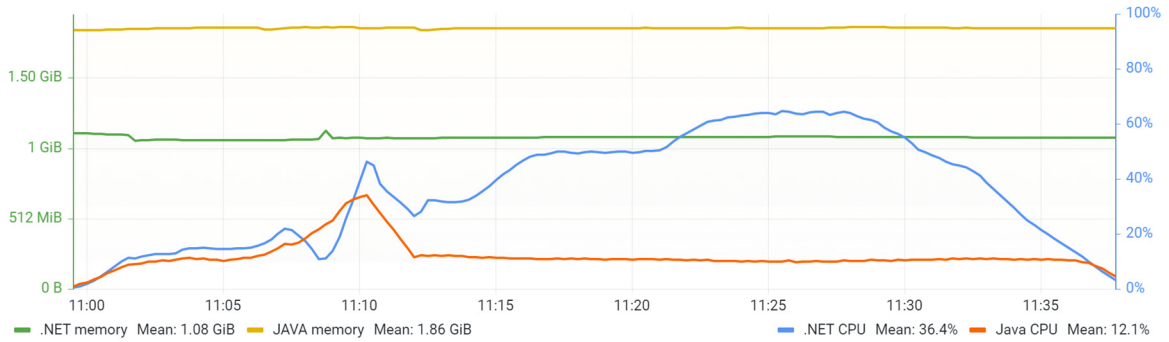

Figure 35 - DELETE stress testing - memory and CPU

In the DELETE requests, the results were similar to the other methods. The .NET API was faster all across the test (Figure 36), with responses from 75% to 80% faster.

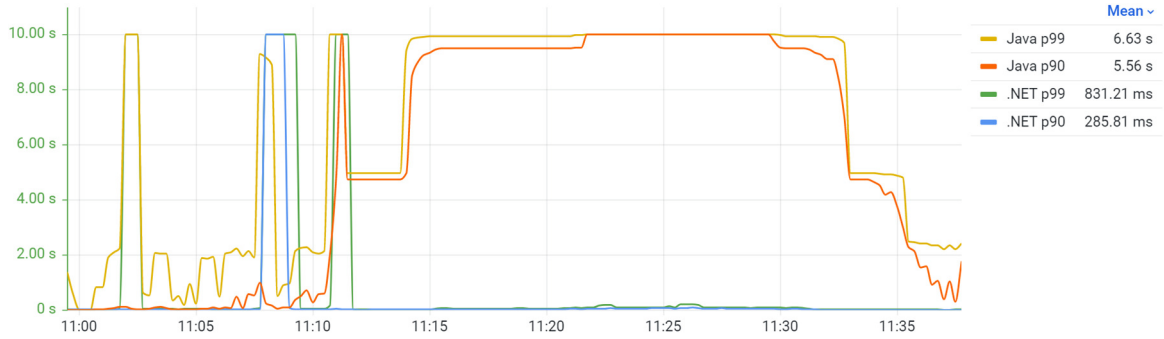

Figure 36 - DELETE stress testing - latency

The report from the command line was critical to understand that both APIs had problems processing the requests, as shown in Table ***14***. Still, there is a 33% gap between both APIs, where the Java API reaches almost 78% of the failed requests.

Table 14 - K6 DELETE stress testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	11.56	40.53	97.48	45.09%	800340	350.90/s
Java	7.08	56.4	5880	77.89%	181930	79.77/s

IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

3. DELETE - Spike test

Looking at the CPU requirements, it could indicate that the Java API was able to outer perform the .NET API, Figure 37.

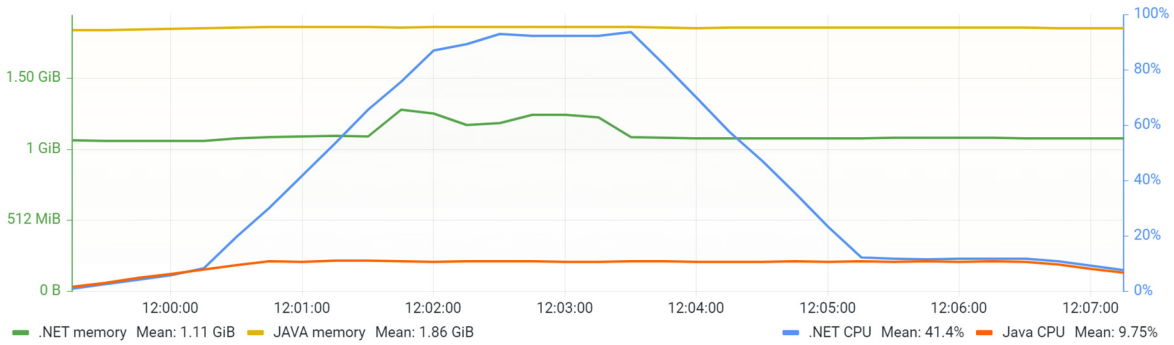

Figure 37 - DELETE spike testing - memory and CPU

Observing Figure 38, both APIs had the same behavior as the other HTTP methods, where the .NET API has better latency.

IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/478010/
Vol. 2024 (2024), Article ID 478010, 31 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.478010

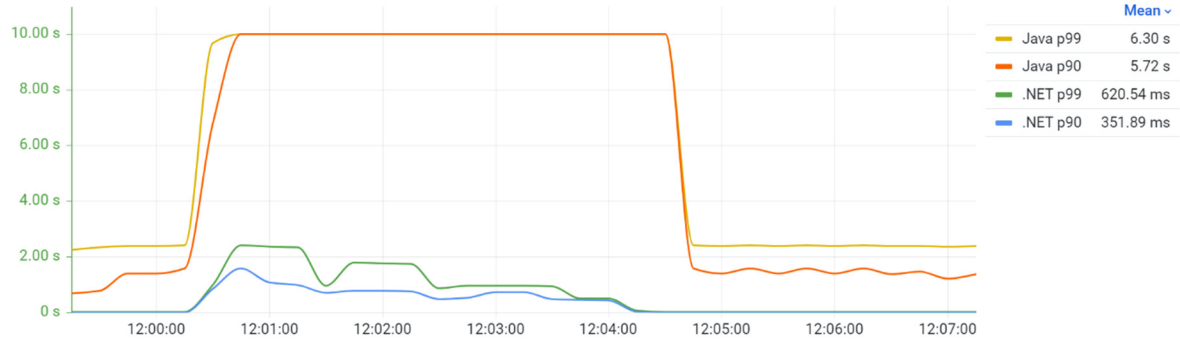

Figure 38 - DELETE spike testing - latency

The results in Table 15 were surprising due to the high number of errors on both APIs, especially on the Java API, where the percentage of errors was close to 100%. This test has shown that Java API could not process the requests, and this problem had to be investigated.

Table 15 - K6 DELETE spike testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	317.45	613.36	1240	49.99%	232389	504.22/s
Java	1370	2808	3114	99.00%	27707	60.14/s

4. DELETE - Soak testing

Once again, the Java API required more RAM, and, after thirty minutes, the CPU requirements dropped to almost 10%, as shown in Figure 39.

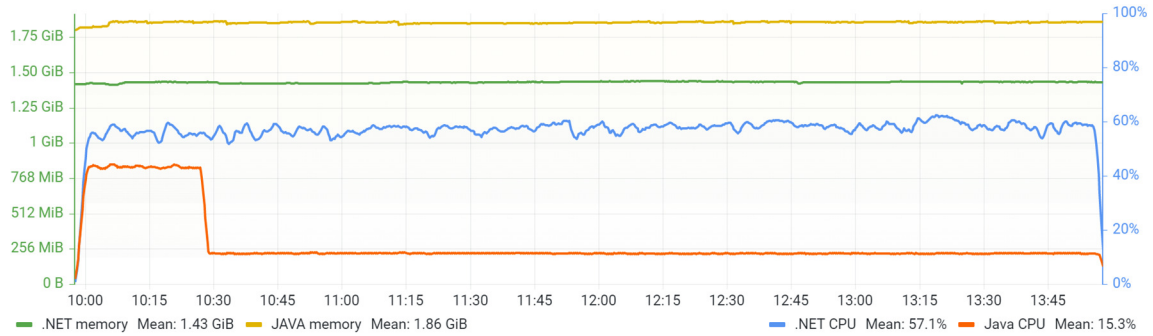

Figure 39 - DELETE soak testing - memory and CPU

The response times of the .NET API were constant and stable across the test. On the side, the Java API response times were over eight seconds, which indicated some problems when compared with the averages of 32 and 42 milliseconds of the .NET API on the percentiles 90th and 99th, as seen in Figure 40.

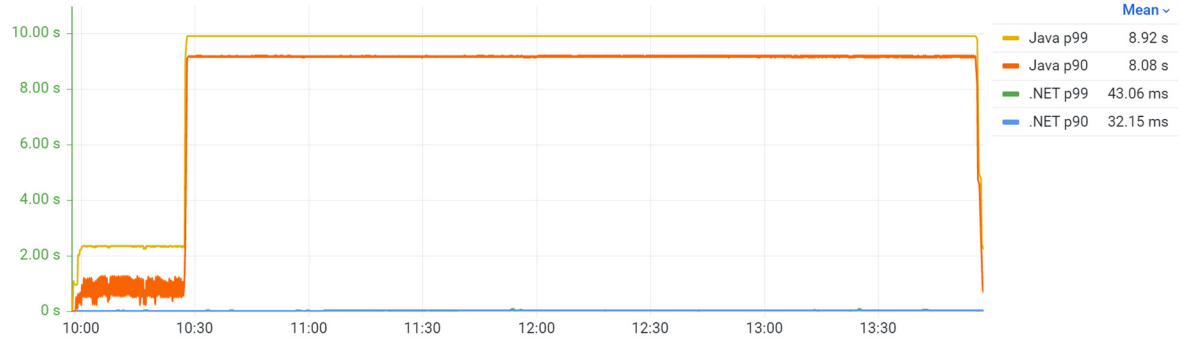

Figure 40 - DELETE soak testing - latency

The results from

Table ***16*** demonstrated the disparity of values of the failed HTTP requests, where the JAVA API had close to 85%.

Table 16 - K6 DELETE soak testing average results

API	**http req duration (ms)**				**http req**	
	avg	***p(90)***	***p(99)***	***failed***	***total***	***per second***
.NET	13.84	29.25	41.86	9.79%	5619734	390.25/s
Java	11.79	977.36	1240	84.75%	1425702	99.00/s

Failed HTTP requests

For a comprehensive understanding of the functionality of the Web APIs, it is crucial to comprehend the reasons behind the failed HTTP requests. These requests may highlight coding errors, which developers can detect and resolve. Moreover, failed HTTP requests can adversely affect the user experience, resulting in dissatisfaction and a negative impression of the application.

Table ***17*** displays the issues associated with the Java API, particularly concerning the POST and DELETE methods, where there is a substantially high rate of failed HTTP requests. This high rate of failed requests could potentially cause issues for applications utilizing this API.

Table 17 - All methods API errors

Method	API		Test		
		Load	Stress	Spike	Soak
GET	.NET	0.00%	0.00%	0.00%	0.00%
	Java	0.00%	0.00%	0.00%	0.00%
POST	.NET	0.00%	0.00%	0.00%	0.00%
	Java	15.43%	38.6%	39.09%	36.41%
PUT	.NET	0.00%	0.00%	0.00%	0.00%
	Java	0.00%	0.00%	0.00%	0.00%
DELETE	.NET	0.00%	45.09%	49.99%	9.79%
	Java	50.69%	77.89%	99.00%	84.75%

Discussion

Implementing two Web Restful APIs using different technologies but sharing a common database posed a significant challenge. The primary difficulties stemmed from issues related to variable naming and proper alignment with the database columns. These difficulties were exacerbated by the fact that the ORMs (Object-Relational Mapping) employed were case-sensitive and sensitive to capitalization.

Despite these challenges, Microsoft's Entity Framework was highly influential in generating all the necessary code when working with a database featuring two tables connected through a foreign key (FK) relationship. It seamlessly managed relationships between tables and created navigation properties, an aspect that Java lacks. In contrast, the Java Persistence API ORM encountered limitations in accessing data from both tables, necessitating a more intricate solution. Queries involving multiple entities had to be manually constructed.

In the initial testing phase, it was observed that when spike tests were applied to both APIs, the root cause of the problem was identified as the

NGINX server, which could not handle the 3000 simultaneous requests. However, the tests also revealed that a reverse proxy could be an effective solution. When running two simultaneous spike tests with the reverse proxy in place, the error rate was less than 2%.

Finally, using .NET is straightforward to obtain the inserted ID of a new record. Getting the inserted ID is another Java ORM limitation that does not allow the INSERT INTO statement to return something. Also, the execution of multiple statements, e.g., INSERT INTO ...; SELECT ..., is not supported by Spring Data JDBC.

Conclusions

Comparing the performance and hardware demands between established technology like Java and newer alternatives like .NET can provide valuable insights into selecting the most suitable option for a RESTful Web API. Four tests were carried out to assess both APIs' performance and hardware requirements: load testing, stress testing, spike testing, and soak testing. The findings indicated that while the .NET API outperformed Java, it also necessitated higher CPU usage.

Under medium to heavy loads, the Java API exhibited suboptimal performance, with response times escalating as the number of virtual users (VUs) increased. Regarding resource utilization, the .NET API demanded more CPU power, whereas the Java API consumed a higher proportion of memory, indicating greater resource consumption.

The test results imply that the .NET API is better equipped for handling high traffic volumes and time-sensitive scenarios, processing more requests with faster response times. Conversely, deploying the Java API in a cloud environment, where servers can be scaled to meet demand, may incur higher costs due to RAM constraints. These tests underscore the importance of aligning API choices with project-specific needs and requirements, as different solutions may excel in distinct contexts.

The tests also underscored the importance of evaluating API performance across all HTTP verbs. Regardless of hardware demands or latency, certain technologies may perform flawlessly with one HTTP method while encountering a notable error rate with another. Refrain from neglecting tests across all methods, which risks yielding incomplete or misleading conclusions. The results highlighted the Java Web API's challenges in handling POST and DELETE HTTP methods, especially under strenuous testing conditions.

Furthermore, it's crucial to acknowledge the current need for studies providing accurate estimates of the CRUD operation usage percentages in a Web API. These data could be a promising avenue for future research utilizing existing logging data.

Acknowledgment

This work is partially funded by National Funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the projects UIDB/00308/2020, UIDB/05583/2020 and MANaGER (POCI-01-0145-FEDER-028040). Furthermore, we would like to thank the Applied Research Institute (i2A) and the Polytechnics Institute of Coimbra for their support.

References

- Bermbach, D. and Wittern, E. (2016) 'Benchmarking Web API Quality', in A. Bozzon, P. Cudre-Maroux, and C. Pautasso (eds) *Web Engineering*. Cham: Springer International Publishing (Lecture Notes in Computer Science), pp. 188–206. Available at: https://doi.org/10.1007/978-3-319-38791-8_11.
- Chakraborty, M. and Kundan, A.P. (2021) 'Grafana', in M. Chakraborty and A.P. Kundan (eds) *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*. Berkeley, CA: Apress, pp. 187–240. Available at: https://doi.org/10.1007/978-1-4842-6888-9_6.
- Coarfa, C., Druschel, P. and Wallach, D.S. (2006) 'Performance analysis of TLS Web servers', *ACM Transactions on Computer Systems*, 24(1), pp. 39–69. Available at: https://doi.org/10.1145/1124153.1124155.
- Fielding, R.T. (2000) *Architectural styles and the design of network-based software architectures*. phd. University of California, Irvine.
- Godinho, A. *et al.* (2024) 'Method for Evaluating the Performance of Web-Based

APIs', in P.J. Coelho, I.M. Pires, and N.V. Lopes (eds) *Smart Objects and Technologies for Social Good*. Cham: Springer Nature Switzerland (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering), pp. 30–48. Available at: https://doi.org/10.1007/978-3-031-52524-7_3.

- Ismail, B.I. *et al.* (2017) 'Reference architecture for search infrastructure', in *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*. *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pp. 115–120. Available at: https://doi.org/10.1109/ICCSCE.2017.8284390.
- Iyengar, A., MacNair, E. and Nguyen, T. (1997) 'An analysis of Web server performance', in *GLOBECOM 97. IEEE Global Telecommunications Conference. Conference Record*. *GLOBECOM 97. IEEE Global Telecommunications Conference. Conference Record*, pp. 1943–1947 vol.3. Available at: https://doi.org/10.1109/GLOCOM.1997.644616.
- Jain, P. *et al.* (2020) 'Performance Analysis of Various Server Hosting Techniques', *Procedia Computer Science*, 173, pp. 70–77. Available at: https://doi.org/10.1016/j.procs.2020.06.010.
- *k6 Documentation* (no date). Available at: https://k6.io/docs (Accessed: 11 March 2024).
- Khan, R. and Amjad, M. (2016) 'Web application's performance testing using HP LoadRunner and CA Wily introscope tools', in *2016 International Conference on Computing, Communication and Automation (ICCCA)*. *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 802–806. Available at: https://doi.org/10.1109/CCAA.2016.7813849.
- Lee, G., Jin, Y. and International Society for Computers and Their Applications (eds) (2019) *34th International Conference on Computers and Their Applications (CATA 2019): Honolulu, Hawaii, USA, 18-20 March 2019*. *International Conference on Computers and Their Applications*, Red Hook, NY: Curran Associates, Inc (EPiC series in computing, volume 58).
- *RESTful Web APIs [Book]* (2013). Available at: https://www.oreilly.com/library/view/restful-web-apis/9781449359713/ (Accessed: 11 March 2024).
- Rodriguez, A. (2008) 'RESTful Web services: The basics', *The basics* [Preprint].
- Sukhija, N. and Bautista, E. (2019) 'Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus', in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pp. 257–262. Available at: https://doi.org/10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00087.
- *The Art of Application Performance Testing, 2nd Edition [Book]* (2014). Available at: https://www.oreilly.com/library/view/the-art-of/9781491900536/ (Accessed: 11 March 2024).
- Turnbull, J. (2018) *Monitoring with Prometheus*. Turnbull Press.
- Voskoglou, C. (2020) *APIs Have Taken Over Software Development | Nordic APIs |, Nordic APIs*. Available at: https://nordicapis.com/apis-have-taken-over-software-development/ (Accessed: 11 March 2024).
- Yoder, J.W. and Johnson, R.E. (1998) 'Connecting Business Objects to Relational Databases'.