Method for Evaluating the Performance of Web-Based APIs

António Godinho[1](✉), José Rosado[2,3], Filipe Sá[2], and Filipe Cardoso[3,4]

[1] Polytechnic Institute of Coimbra, Coimbra Business School Quinta Agrícola - Bencanta, 3045-231 Coimbra, Portugal
agodinho@iscac.pt

[2] Polytechnic Institute of Coimbra, Coimbra Institute of Engineering Rua Pedro Nunes - Quinta da Nora, 3030-199 Coimbra, Portugal
{jfr,filipe.sa}@isec.pt

[3] INESC Coimbra—Instituto de Engenharia de Sistemas e Computadores de Coimbra, Rua Sílvio Lima, Pólo II, 3030-790 Coimbra, Portugal

[4] Escola Superior de Gestão e Tecnologia, Politécnico de Santarém Complexo Andaluz, Apartado 295, 2001-904 Santarém, Portugal
filipe.cardoso@esg.ipsantarem.pt

Abstract. Application Programming Interfaces (APIs) are available in virtually every programming language. These interfaces make it easier to develop software by simplifying complex code into a more straightforward, manageable structure. APIs provide a standardized interface that allows different applications to communicate and connect easily, streamlining the software development process and making it more efficient and effective. Performance testing of a web API refers to evaluating the performance characteristics of an API accessible via the web. This process involves analyzing performance aspects such as response time, reliability, scalability, and resource utilization. This work defines a test battery using specific open-source tools to assess Web API performance. The tests used are load, stress, spike, and soak tests replicating various scenarios of the volume of users accessing the service or simulating a denial-of-service attack. These tests aim to determine how well an API can manage a substantial volume of traffic and transactions while upholding satisfactory performance standards. Applying Web API performance testing will also enable organizations to implement suitable measures for enhancing performance and guaranteeing smooth user interaction, pinpointing bottlenecks, constraints, or prospective problems in the API's architecture and execution. These tests can also demonstrate the technology's limitations and benchmarking, helping determine a more suitable production platform.

Keywords: web api · full-stack development · performance analysis · performance tools · linux operating systems

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
P. J. Coelho et al. (Eds.): GOODTECHS 2023, LNICST 556, pp. 30–48, 2024.
https://doi.org/10.1007/978-3-031-52524-7_3

1 Introduction

Full-stack development has seen tremendous growth recently due to the increasing demand for web development as the internet, and e-commerce continues to expand. Both mobile and web applications use RESTful Web APIs for authentication, data access, file management, and other resources. RESTful APIs are REST-based APIs that use resource identifiers to represent specific resources intended for interaction between components. The current state of a resource is referred to as a resource representation, which consists of data, metadata describing the data, and hypermedia links that allow for changing the state of the resource [1]. RESTful architectural design is a specific method for implementing APIs, introduced in 2000 by Roy Fielding. This design involves a set of constraints to improve an API's reliability, scalability, and performance [2]. APIs generally serve as interfaces with a set of functions, protocols, and tools to integrate software applications and services. Web APIs, in particular, can be accessed over the web through the HTTP/HTTPS protocols, allowing requesting systems to access and manipulate web resources using standard and predefined, uniform rules. REST-based systems interact through the Internet's Hypertext Transfer Protocol (HTTP) [3]. A Web API enables the front-end or multiple front-ends for different web application devices to communicate with the back-end by sending requests to specific endpoints and receiving data in response, as shown in Fig. 1. According to a survey from the developer nation in 2020, a staggering 90% of developers utilize APIs, demonstrating that the proliferation of APIs has played a crucial role in the growth of the developer ecosystem in recent years. The high adoption rate of APIs among developers serves as solid evidence that the rise of APIs has significantly impacted and contributed to the expansion of the developer ecosystem [4]. 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.

Analyzing the performance of web applications is a common practice, with most studies focusing solely on testing the application as a whole. However, it is essential to assess the entire solution, including isolating testing of the Web

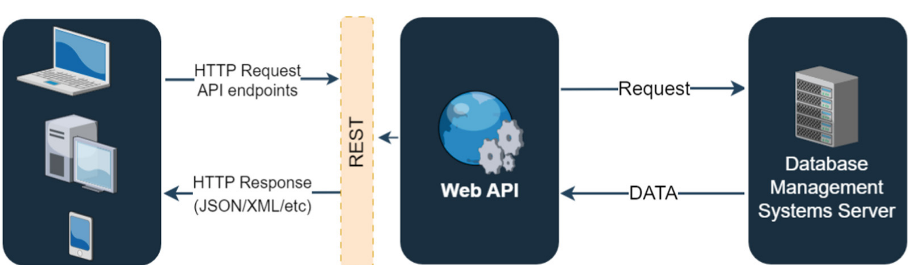

Fig. 1. Web API

API. This approach can effectively identify any potential issues specific to the Web API. This paper introduces a suggested suite of tests designed to evaluate how Web APIs behave across various CRUD (create, read, update, and delete) operations. These tests facilitate the examination of the application's performance under diverse circumstances, encompassing both typical and exceptionally high request rates, as well as prolonged and resource-intensive durations. Furthermore, we provide a collection of tools for assembling the test suite and for visualizing and interpreting the outcomes it generates.

The article is structured into six sections, starting with the Introduction. In this section, readers will gain an understanding of the article's objectives and the reasoning behind them. The second section describes RESTful Web API's technology's functionality, including its norms and practical applications, and an understanding of the key features that distinguish RESTful APIs from other APIs. In the third section, we introduce a set of tools utilized to construct and test the Web API and visualize the test results. The fourth section of API is performance testing, where the test battery is presented and why each test should be applied. The fourth section of the article focuses on performance testing, where the test battery is presented. It explains the reasoning behind each test's application, with the importance of performance testing and the specific tests. The fifth section provides insights into the possible outcomes of the performance testing and also how to visualize the test results. Finally, the sixth section is the Conclusions, which summarizes the importance of running the different tests on each CRUD operation.

2 Related Work

A previous study concentrated on assessing the latency and performance of Web APIs but encountered the challenge of defining a standardized set of tests that could be universally applicable across different technological contexts [5]. Similarly, various research efforts have attempted to compare performance across diverse technologies. Yet, they, too, have faced the limitation of needing a comprehensive test suite adaptable to various scenarios or technological environments [1,6,7].

In a separate line of investigation, some studies have compared performance between two prominent architectural styles for web service development: REST and SOAP/WSDL. However, these studies typically needed to include the utilization of multiple tests with varying loads, limiting the breadth of their performance evaluations. Conversely, numerous other research endeavors have honed in on assessing the performance of Web APIs in the context of microservices-based web applications [8–10].

Earlier studies have delved into Web API performance and benchmarking analysis, with certain ones outlining the methodologies employed to yield their results. However, these studies often grappled with the challenge of creating a standardized testing framework that could be universally applied across diverse technological contexts.

In contrast, the present research addresses these limitations by undertaking a comprehensive examination. This examination encompasses all CRUD (Create, Read, Update, Delete) operations within Web APIs and is intentionally designed to be platform and technology-agnostic.

3 RESTful Web API

A well-designed Web API can expose the functionality of the back-end to other applications and services, allowing for the reuse of existing code and easy integration of new services [11]. Web API has the advantage of allowing different teams and developers to work together more efficiently and build more powerful and flexible web applications [12]. For example, one team can focus on front-end development, another on back-end development, and a third on infrastructure or DevOps. This clear frontier allows each team to have a deeper understanding and expertise in their area of focus, which can result in better quality and more efficient development. Splitting the work across teams can make it easier to manage and scale larger projects [13]. The Web API can be developed using different technologies, from Java, .NET, or JavaScript, using web-development frameworks. A Web API is a set of rules and protocols that allows different software applications to communicate with each other. APIs provide a way for different programs to interact with one another without requiring direct access to the underlying code. APIs are often used to access web-based software, such as social media sites, weather services, and online databases. For example, when a client uses a mobile app to check the weather using a mobile phone, the app is likely using an API to retrieve the data from a weather service's servers [14]. Some web services provide APIs for clients to access their functionality and data. In such a scenario, the API is a set of functions, methods, and protocols that provide access to the functionality and data of a service, such as a database or a web application [15]. Then, Web API, when conforming to the REST architectural principles, are characterized by their relative simplicity and their natural suitability for the Web, relying almost entirely on the use of URIs for both resource identification and interaction and HTTP for message transmission [16].

3.1 Representational State Transfer (REST)

Following protocols, such as SOAP and RESTful web services, can implement Web APIs. The development of mobile applications was the initial driving force for RESTful, adopted over other protocols due to the simplicity of use [17]. There are clear advantages to the use of REST. Typically faster and uses less bandwidth because it uses a smaller message format. Another main reason is that it supports many different data formats, such as JSON, XML, CSV, and plain text, whereas SOAP supports only XML [18]. Representational State Transfer (REST) is a software architectural style that defines the rules for creating web services. RESTful Web APIs are based on the principles of REST architecture, which defines a set of architectural constraints that a web service must adhere

to be considered RESTful, first described by Roy Fielding in his doctoral dissertation [2]. A RESTful Web API must follow these six architectural constraints [19]:

- Client-Server: Separate client and server concerns for independent evolution.
- Stateless: No client state retention, all needed data in requests.
- Cacheable: Clients can cache responses for improved performance.
- Layered System: Clients access API functionality consistently regardless of infrastructure.
- Code on Demand (Optional): Allows downloading executable code for client extension.
- Uniform Interface: Ensures an easy-to-learn and consistent client interaction.

Web APIs facilitate the seamless communication and collaboration of various software systems, which may have been developed using diverse technologies and programming languages. They foster interoperability across a broad spectrum of platforms and devices [20]. By leveraging APIs, developers can deconstruct intricate systems into more manageable, bite-sized components. This approach to modularity streamlines the processes of development, upkeep, and software updates. APIs empower applications to incorporate external services and data sources, broadening their capabilities and granting access to a broader array of services [21].

3.2 API HTTP Verbs

The API Interface should be simple, consistent, self-describing, and supports the most common standard HTTP methods (HTTP verbs): GET, POST, PUT, and DELETE. These verbs are used to indicate the intended action to be performed on the requested resource. Usually, they are translated into the CRUD operations - Create, Read, Update, and Delete [22].

4 Tools

A combination of Prometheus, Fluentd, and Grafana was utilized to facilitate monitoring this work. These tools were employed to collect statistics and create informative dashboards, providing insight into the performance and behavior of the system.

Prometheus is an open-source system monitoring and alerting toolkit. Provides real real-time monitoring and alerting on the performance of microservices-based applications in cloud-native environments. Prometheus uses a powerful query language and a flexible data model that makes it easy to collect and store metrics from various systems and applications. The tool also includes built-in alerting and visualization capabilities [23].

Fluentd is an open-source data collection and logging tool. It can collect data from a wide variety of sources using input plugins and store the data in various destinations using output plugins. For this work, it will be used to read Nginx

logs, parsing them into specific fields to Prometheus. Prometheus can't process the NGINX logs to verify the accesses for each API. Fluentd can split the access logs into specific fields, such as IP, URL, and HTTP code. Then, Prometheus uses Fluentd as a data source. One of the fields relates to the path on the URL, which will be used on queries to Prometheus by Grafana to generate specific charts for each API.

Grafana is a popular open-source time-series data query, visualization, and alerting tool which was developed by Torkel Ödegaard in 2014. It has a highly pluggable data source model that supports multiple time-series-based data sources like Prometheus and Fluentd and SQL databases like MySQL and Postgres [24]. In this work, the data sources will be Prometheus, which provide the source for the virtual machine CPU and RAM, and Fluentd, for the NGINX reverse proxy.

For test and performance, the tools used were: cURL, Hey, and K6. cURL stands for "Client for URLs" and is a command-line tool for transferring data using various protocols. It is commonly used to send HTTP and HTTPS requests. Hey is an open-source load-testing tool for web servers developed by Jaana B. Dogan. It allows users to generate many HTTP requests to a specified endpoint to measure the endpoint's performance and the server it runs on. Hey can be used to simulate different types of traffic, such as concurrent users, and it provides metrics such as request rate, latency, and error rate [25]. K6 is an open-source load-testing tool that allows developers to test web applications and APIs' performance and scalability. It is written in Go, like Hey, and uses JavaScript as its scripting language for testing scenarios. It allows traffic simulation to a website or an API [26]. cURL and Hey was used for initial testing and to verify the testing environment, while K6 is the tool used in the examples in this work, with different setups for each test.

4.1 Test Scenario

The test scenario involved the setup of multiple virtual machines running Linux. Nginx was installed on the head node as a reverse proxy solution. The same VM hosted Prometheus, Grafana, and Fluentd to collect statistics and generate charts. Another VM housed a Java API connected to a third VM running a database engine, as depicted in Fig. 2.

5 WEB API Performance Testing

Performance testing is a task performed to determine how a system accomplishes responsiveness and stability under a particular workload. It can also investigate, measure, validate, or verify other system quality attributes, such as scalability, reliability, and resource usage [27]. It is also an important test to identify bottlenecks and ensure that the software can handle the expected usage and demand. Several tests used to measure website performance may also be applied to Web API. Each test uses the tools presented in Sect. 4, following the three phases of

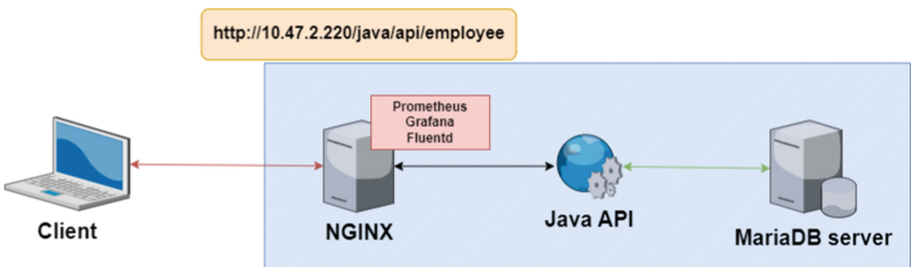

Fig. 2. Test scenario

traditional software testing: test design, test execution, and test analysis [28]. These steps should start by designing realistic loads for each type of test, simulating the workload that may occur in the field, or designing fault-inducing loads, which are likely to expose load-related problems. Once again, the tools from Sect. 4 will process logs, generating charts and tables with statistics [29].

5.1 The 99th, 95th and 90th Percentiles

The 99th percentile is often used as a benchmark for performance testing because it represents a high level of performance. It measures how well a system performs compared to others and helps identify any outliers or issues that need to be addressed. Additionally, using the 99th percentile instead of the average (mean) or median can provide a more accurate representation of system performance, as it eliminates the impact of a smaller number of extreme results. The 95th percentile is also a commonly used benchmark for performance testing because it represents a level of performance that is considered good but not necessarily the best. It can provide a more realistic measure of performance, as it believes there may be some variability in results. Similarly to the previous, using the 90th percentile instead of the average (mean) or median can provide a more accurate representation of system performance, as it eliminates the impact of a smaller number of extreme results. In this case, the 90th percentile can help identify if the system is not meeting the desired performance level and any issues or bottlenecks that must be addressed. The mean and percentiles will be utilized in Grafana to create charts from the tests defined in K6.

5.2 Number of Virtual Users

The initial step in creating testing scenarios is often determining the appropriate number of concurrent users to simulate, establishing the foundation for establishing performance objectives. Although estimating the maximum simultaneous users for a new website can be difficult, for an existing website, numerous data sources, such as Google Analytics, can be utilized to establish performance targets and provide valuable information about the number of concurrent users

likely to be required. To ensure the correctness of the testing environment and to determine a consistent number of virtual users (VUs) for all future tests, multiple pilot tests using different numbers of virtual users should be conducted for new applications (Fig. 3). Analyzing hardware requirements across various VU numbers is crucial for achieving optimal performance, CPU utilization, memory usage, and latency response. Conducting tests with different VUs can reveal diverse CPU and memory requirements behaviors, as demonstrated in Figs. 3 and 4. Notably, in this example, the CPU requirements increase with the number of users. Still, the memory requirements remain similar or even decrease, which contradicts the initial impression, highlighting the importance of analyzing the various hardware components.

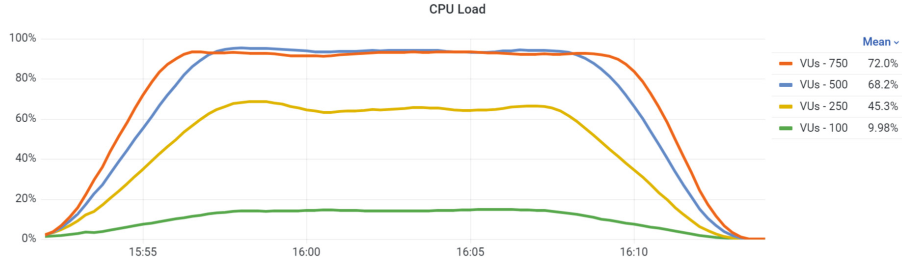

Fig. 3. Pilot tests - CPU requirements

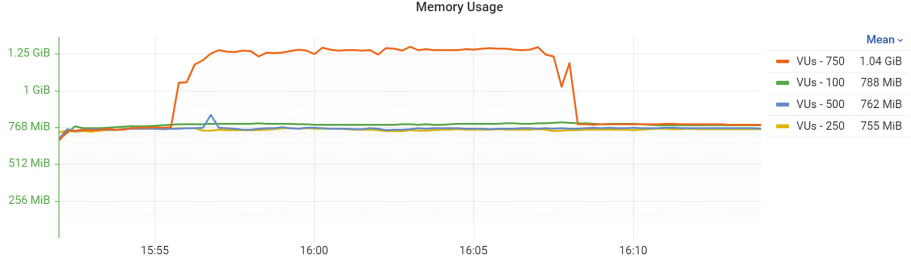

Fig. 4. Pilot tests - Memory requirements

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. 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 "MaxRequestWorkers" directive in its configuration file. The default value is 256 [30], 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 [31]. Assuming that at least two workers are used, the number of allowed connections can be up to 2048.

5.3 Load Testing

Load testing primarily focuses on evaluating a system's current performance in terms of the number of concurrent users or requests per second. It is used to determine if a system is meeting its performance goals. By conducting a load test, you can evaluate the system's performance under normal load conditions, ensure that performance standards are being met as changes are made, and simulate a typical day in the business [32]. These tests are done using tools that use VUs to simulate the requests, as shown in Fig. 5. The configuration file in Fig. 6 specifies a maximum of 100 VUs for the test. As mentioned in Sect. 5.2, this value should be customized based on the expected traffic for a Web API or new applications. Figure 6 also shows that multiple endpoints can be tested simultaneously on the same instance using the same HTTP method.

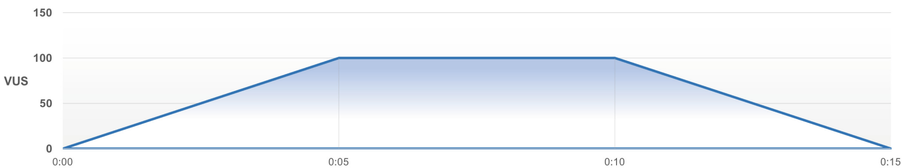

Fig. 5. Load testing - VUs progress over time

5.4 Stress Testing

Stress testing is a form of load testing used to identify a system's limits. This test aims to assess the system's stability and dependability under high-stress conditions. By conducting a stress test, it can determine how the system will perform under extreme conditions and the maximum capacity of the system in terms of users or throughput. Also, the point at which the system will break, how it will fail, and whether it will recover automatically after the stress test is complete without manual intervention, as shown in Fig. 7. The configuration file shown in Fig. 8 specifies the VUs for different time intervals during the stress test, as indicated by the chart in Fig. 7.

```
import { batch } from "k6/http";
import { sleep } from "k6";

export let options = {
        insecureSkipTLSVerify: true,
        noConnectionReuse: false,
        stages: [
                { duration: "5m", target: 100 },
                { duration: "10m", target: 100 },
                { duration: "5m", target: 0 },
        ],
};

export default function () {
        batch([
                ["GET", `http://10.47.2.220/net/api/employee`],
                ["GET", `http://10.47.2.220/net/api/department`],
        ]);
        sleep(1);
}
```

Fig. 6. K6 Load testing

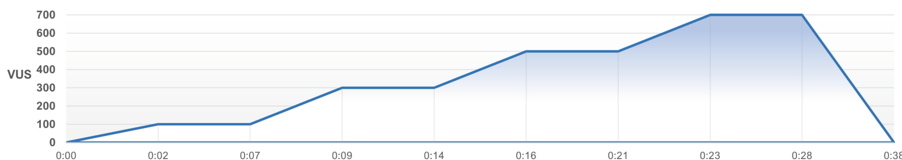

Fig. 7. Stress testing - VUs progress over time

```
import { batch } from "k6/http";
import { sleep } from "k6";

export let options = {
        insecureSkipTLSVerify: true,
        noConnectionReuse: false,
        stages: [
                { duration: "2m", target: 100 },
                { duration: "5m", target: 100 },
                { duration: "2m", target: 250 },
                { duration: "5m", target: 250 },
                { duration: "2m", target: 500 },
                { duration: "5m", target: 500 },
                { duration: "2m", target: 750 },
                { duration: "5m", target: 750 },
                { duration: "10m", target: 0 },
        ],
};

export default function () {
        batch([
                ["GET", `http://10.47.2.220/net/api/department`],
        ]);
        sleep(1);
}
```

Fig. 8. K6 Stress testing

5.5 Spike Test

A spike test is a variation of a stress test that involves subjecting a system to extreme load levels in a very short period. The main objective of a spike test is to determine how the system will handle a sudden increase in traffic and identify any bottlenecks or performance issues that may arise. This type of test can help identify potential problems before they occur in a production environment and ensure that the system can handle expected levels of traffic [32–34]. By conducting a spike test, you can determine how the system will perform under a sudden surge of traffic, most frequently a Denial of Service (DOS) attack, and whether it can recover once the traffic has subsided. The success of a spike test can be evaluated based on expectations, and systems generally react in one of four ways: excellent, good, poor, or bad.

- "Excellent" performance is when the system's performance is not degraded during the surge of traffic, and the response time is similar during low and high traffic;
- "Good" performance is when response time is slower, but the system does not produce errors, and all requests are handled;
- "Poor" performance is when the system produces errors during the surge of traffic but recovers to normal after traffic subsides;
- "Bad" performance is when the system crashes and does not recover after the traffic has subsided, as depicted in Fig. 9.

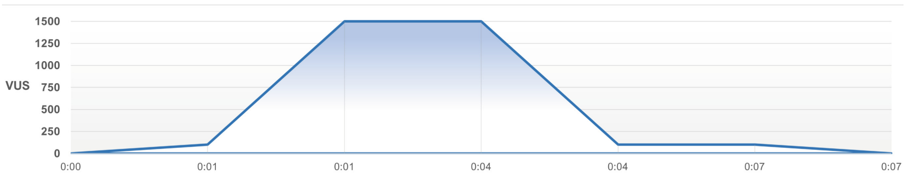

Fig. 9. Spike testing - VUs progress over time

Figure 10 shows the configuration file where it is defined that the VUs will peak at 1500 and sustain for 3 min.

5.6 Soak Testing

Soak testing is used to evaluate the reliability of a system over an extended period. By conducting a soak test, you can determine if the system is prone to bugs or memory leaks that may cause it to crash or restart, ensure that expected application restarts do not result in lost requests, identify bugs related to race

```
import { batch } from "k6/http";
import { sleep } from "k6";

export let options = {
        insecureSkipTLSVerify: true,
        noConnectionReuse: false,
        stages: [
                { duration: "10s", target: 100 },
                { duration: "1m", target: 100 },
                { duration: "10s", target: 1500 },
                { duration: "3m", target: 1500 },
                { duration: "10s", target: 100 },
                { duration: "3m", target: 100 },
                { duration: "10s", target: 0 },
        ],
};

export default function () {
        batch([
                ["GET", `http://10.47.2.220/net/api/department`],
        ]);
        sleep(1);
}
```

Fig. 10. K6 Spike testing

conditions that occur sporadically, confirm that the database does not exhaust allocated storage space or stop working, verify that logs do not deplete the allotted disk storage, and ensure that external services that the system depends on do not stop working after a certain number of requests [34]. To run a soak test, you should determine the maximum capacity that the system can handle, set the number of VUs to 75–80% of that value, and run the test in three stages: ramping up the VUs, maintaining that level for 4–12 h, and ramping down to 0, as shown on Fig. 11. A capacity limit of 75% to 80% for the soak test may place too much strain on the database, potentially causing the test to fail. To prevent this, the test should be conducted with a lower number, for example, using the default capacity limit for the Apache web server, which amounts to 400 connections when set to 80% capacity. The file configuration in Fig. 12 displays the VUs reaching 400 and staying at that level for approximately 4 h.

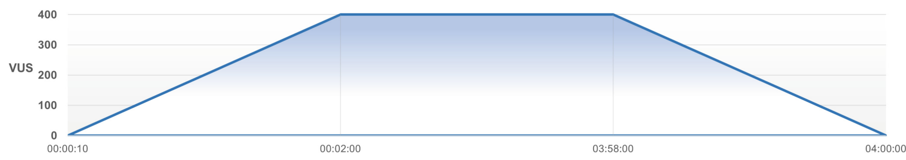

Fig. 11. Soak testing - VUs progress over time

5.7 Tests to All CRUD Operations

The tests outlined in Sect. 5 are exclusively related to the GET method. When evaluating an API's performance, testing the other HTTP verbs used for the

```
import { batch } from "k6/http";
import { sleep } from "k6";

export let options = {
        insecureSkipTLSVerify: true,
        noConnectionReuse: false,
        stages: [
                { duration: "2m", target: 400 },
                { duration: "3h56m", target: 400 },
                { duration: "2m", target: 0 },
        ],
};

export default function () {
        batch([
                ["GET", `http://10.47.2.220/net/api/department`],
        ]);
        sleep(1);
}
```

Fig. 12. K6 Soak testing

CRUD operations, such as the POST method for creating new records or the PUT method for updating existing ones, is essential. In these cases, a valid JSON payload must be added and sent to the server, as demonstrated in the example in Fig. 13. This example also showcases using environment variables and virtual users for iteration. The PUT method needs a present identifier to update a record successfully. Testing for deletions is challenging, as most web APIs include the identifier of the record to be deleted in the URL. It necessitates a distinct strategy for deleting a valid existing record, and various methods exist. For testing purposes, you can define the inserted identifier and use only identifiers that combine the iteration and virtual user. Alternatively, you can eliminate the identifier parameter and erase the identifier with the highest value on the database.

6 Results

The tests run on a linux console, using K6 provide initial results via command shell, as shown in Figs. 14 and 15. The results provide valuable performance information, particularly for HTTP request duration - median, percentile 99, and 90, as well as the number of test iterations and iterations per second and the number and percentage of failed HTTP requests. This information reveals the latency of requests at each percentile and the performance and potential bottlenecks of the Web API, as demonstrated by the results. The GET method in Fig. 14 ran without any issues, while with the PUT method (Fig. 15), over 10% of the requests failed due to 40x HTTP errors. Running the test only on the GET method could be deceiving, and it shows the importance of testing all methods, helping to identify potential technology or code problems. Using Grafana helps to gain a deeper understanding of the performance data. Grafana provides charts and visualizations and can be used to cross information such as VU numbers with the number of failed requests. It can also be used to visualize key metrics such as the latency of HTTP requests at the 99th and 90th percentiles, as demonstrated in Fig. 16. In addition to that, it also provides insight

```
import http from 'k6/http';
import { sleep } from "k6";
import exec from 'k6/execution';

export let options = {
        insecureSkipTLSVerify: true,
        noConnectionReuse: false,
        stages: [
                { duration: "5m", target: 100 },
                { duration: "10m", target: 100 },
                { duration: "5m", target: 0 },
        ],
};

export default function () {
        const url = 'http://10.47.2.220/net/api/employee';
        const payload = JSON.stringify({
                fullName: "Teste - " + __ITER + " - " + __VU,
                idDepartment: 4,
                salary: '250000',
                hireDate: '17/02/2018'
        });

        const params = {
                headers: {
                        'Content-Type': 'application/json',
                },
        };

        http.post(url, payload, params);
        sleep(1);
}
```

Fig. 13. K6 POST load test

```
root@openvpn:~/K6# k6 run java-get-spike-test.js --summary-trend-stats="med,p(90),p(99)"

  execution: local
     script: java-get-spike-test.js
     output: -

  scenarios: (100.00%) 1 scenario, 1500 max VUs, 8m10s max duration (incl. graceful stop):
           * default: Up to 1500 looping VUs for 7m40s over 7 stages (gracefulRampDown: 30s, gracefulStop: 30s)


     data_received..................: 198 MB 429 kB/s
     data_sent......................: 21 MB  47 kB/s
     http_req_blocked...............: med=5.67µs  p(90)=8.05µs   p(99)=370.28µs
     http_req_connecting............: med=0s      p(90)=0s       p(99)=252.04µs
     http_req_duration..............: med=2.25s   p(90)=2.48s    p(99)=2.72s
       { expected_response:true }...: med=2.25s   p(90)=2.48s    p(99)=2.72s
     http_req_failed................: 0.00%   ✓ 0          ✗ 225344
     http_req_receiving.............: med=79.89µs p(90)=101.01µs p(99)=180.84µs
     http_req_sending...............: med=25.76µs p(90)=37.18µs  p(99)=94.19µs
     http_req_tls_handshaking.......: med=0s      p(90)=0s       p(99)=0s
     http_req_waiting...............: med=2.25s   p(90)=2.48s    p(99)=2.72s
     http_reqs......................: 225344 489.287271/s
     iteration_duration.............: med=3.29s   p(90)=3.54s    p(99)=3.81s
     iterations.....................: 112672 244.643636/s
     vus............................: 10     min=6        max=1500
     vus_max........................: 1500   min=1500     max=1500
```

Fig. 14. K6 GET Spike test

```
root@openvpn:~/K6# k6 run java-put-spike-test.js --summary-trend-stats="med,p(90),p(99)"

          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: java-put-spike-test.js
     output: -

  scenarios: (100.00%) 1 scenario, 1500 max VUs, 8m10s max duration (incl. graceful stop):
           * default: Up to 1500 looping VUs for 7m40s over 7 stages (gracefulRampDown: 30s, gracefulStop: 30s)


running (7m40.6s), 0000/1500 VUs, 133520 complete and 0 interrupted iterations
default ✓ [======================================] 0000/1500 VUs  7m40s

     data_received..................: 43 MB  94 kB/s
     data_sent......................: 36 MB  79 kB/s
     http_req_blocked...............: med=5.58µs  p(90)=7.12µs  p(99)=347.24µs
     http_req_connecting............: med=0s      p(90)=0s      p(99)=233.8µs
     http_req_duration..............: med=1.65s   p(90)=1.9s    p(99)=2.29s
       { expected_response:true }...: med=1.65s   p(90)=1.89s   p(99)=2.28s
     http_req_failed................: 10.84% ✓ 14474      ✗ 119046
     http_req_receiving.............: med=77.85µs p(90)=96.49µs p(99)=180.94µs
     http_req_sending...............: med=32µs    p(90)=44.31µs p(99)=98.93µs
     http_req_tls_handshaking.......: med=0s      p(90)=0s      p(99)=0s
     http_req_waiting...............: med=1.65s   p(90)=1.9s    p(99)=2.29s
     http_reqs......................: 133520 289.874794/s
     iteration_duration.............: med=2.65s   p(90)=2.9s    p(99)=3.29s
     iterations.....................: 133520 289.874794/s
     vus............................: 1      min=1         max=1500
     vus_max........................: 1500   min=1500      max=1500
```

Fig. 15. K6 PUT Spike test

Fig. 16. Grafana - Latency p99 and p90

into the CPU and memory utilization, as shown in Fig. 17. This complete picture of the Web API's performance allows for quick and easy identification of any potential bottlenecks or areas for improvement. Using multiple sources on Grafana, it is possible to compare different APIs on the same chart, allowing direct comparison, shown in Fig. 18.

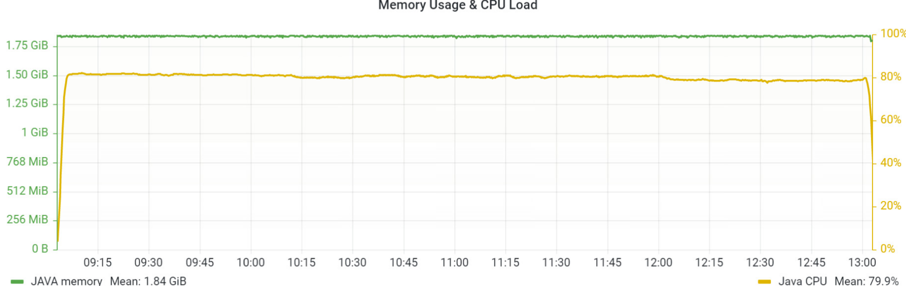

Fig. 17. Grafana - CPU and memory

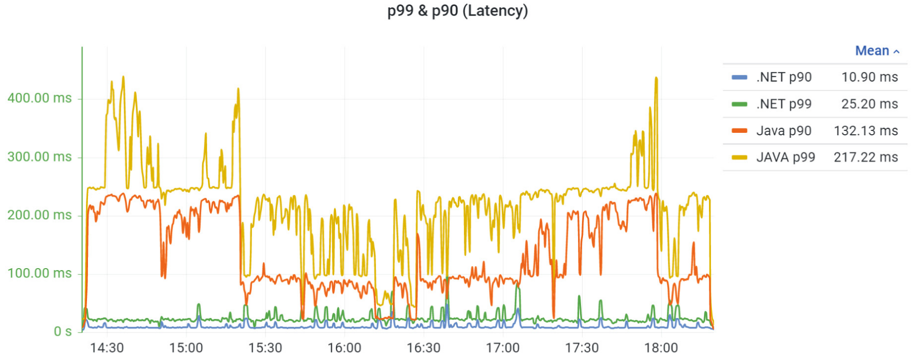

Fig. 18. Soak test .NET vs Java Spring

7 Conclusions

Web API performance is essential because it directly affects the user experience and the application's overall success. Poor API performance can result in slow response times, error messages, and frustrated users. It can decrease user engagement, and e-commerce websites may reduce revenue. On the other hand, fast and reliable API performance can provide a better user experience, increase customer satisfaction, and drive business growth. In addition, efficient Web API performance is crucial for scalability and sustainability. As the number of users and API requests increase, the API must be able to handle the increased load without slowing down or crashing. A well-optimized API can handle significant traffic and requests, allowing smooth and seamless growth. Therefore, monitoring and improving Web API performance should be a priority for any organization that relies on APIs to power their applications and services.

One of the critical aspects of performance testing is defining the number of virtual users that will be used to simulate real user traffic. The number of virtual users required for performance testing will depend on several factors, including the system's nature, the expected user load, and the testing goals.

There are several factors or steps, but conducting a pilot test with a few virtual users ensures that the testing environment is set up correctly and establishes a baseline for the system's performance. From that point, gradually increase the number of virtual users, monitoring the system's performance at each stage. Gradually increase the number of users until the system reaches its maximum capacity or until the testing goals have been achieved.

The tests outlined in this work aim to evaluate a Web API under varying workloads thoroughly. The tests should cover the primary HTTP verbs, including GET, POST, PUT, and DELETE. The GET test should address the most demanding scenario: retrieving all entities from a single endpoint. The POST test focuses on creating new records in the database, the PUT test focuses on updating existing resources, and the DELETE test focuses on removing resources. The comprehensive test suite must be run on the four CRUD (Create, Read, Update, and Delete) operations to identify and eliminate potential performance problems. By thoroughly testing each of the CRUD operations, you can gain confidence in the reliability and scalability of the system and prevent any unexpected issues from arising during production use. Testing all HTTP methods may also help to determine the appropriate number of virtual users for the tests.

The tools presented in this work are a valid method for obtaining real-world results and testing the response limits of the application. The results may be visualized through charts generated by these tools, clearly representing any issues detected during the testing process. Running multiple queries on a single chart allows running the same test on numerous Web APIs, visualizing the results on a single graph, and providing a tool for direct comparison.

Acknowledgements. This work is funded by FCT/MEC through national funds and co-funded by FEDER—PT2020 partnership agreement under the project **UIDB/50008/2020**. 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 Polytechnics of Coimbra and Santarém for their support.

References

1. Hong, X.J., Yang, H.S., Kim, Y.H.: Performance analysis of restful API and RabbitMQ for microservice web application. In: 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), pp. 257–259 (2018). https://doi.org/10.1109/ICTC.2018.8539409
2. Fielding, R.T.: Architectural Styles and the Design of Network-Based Software Architectures. University of California (2000)
3. Karlsson, O.: A Performance comparison Between ASP. NET Core and Express. js for creating Web APIs. [Dissertation] (2021). http://urn.kb.se/resolve?urn=urn:nbn:se:hj:diva-54286
4. Voskoglou, C.: APIs Have Taken over Software Development: Nordic Apis —. Nordic APIs, 20 October 2020. https://nordicapis.com/apis-have-taken-over-software-development/

5. Bermbach, D., Wittern, E.: Benchmarking web API quality. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds.) ICWE 2016. LNCS, vol. 9671, pp. 188–206. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38791-8_11
6. Kronis, K., Uhanova, M.: Performance comparison of Java EE and ASP. NET core technologies for web API development. Appl. Comput. Syst. **23**(1), 37–44 (2018)
7. Karlsson, O.: A Performance comparison between ASP. NET Core and Express. js for creating Web APIs (2021)
8. Rathod, D.: Performance evaluation of restful web services and soap/wsdl web services. Int. J. Adv. Res. Comput. Sci. **8**(7), 415–420 (2017)
9. Akbulut, A., Perros, H.G.: Performance analysis of microservice design patterns. IEEE Internet Comput. **23**(6), 19–27 (2019)
10. El Malki, A., Zdun, U.: Combining API Patterns in Microservice Architectures: Performance and Reliability Analysis (2023)
11. Geewax, J.J.: API design patterns. Simon and Schuster (2021)
12. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating web APIs on the world wide web. In: 2010 Eighth IEEE European Conference on Web Services, Ayia Napa, Cyprus, pp. 107–114 (2010). https://doi.org/10.1109/ECOWS.2010.9
13. Vainikka, J.: Full-stack web development using Django REST framework and React (2018)
14. Richardson, L., Amundsen, M., Ruby, S.: RESTful Web APIs: Services for a Changing World. O'Reilly Media, Inc., Sebastopol (2013)
15. Ong, S.P., et al.: The materials application programming interface (API): a simple, flexible and efficient API for materials data based on representational state transfer (REST) principles. Comput. Mater. Sci. **97**, 209–215 (2015)
16. Neumann, A., Laranjeiro, N., Bernardino, J.: An analysis of public REST web service APIs. IEEE Trans. Serv. Comput. **14**(4), 957–970 (2018)
17. Halili, F., Ramadani, E.: Web services: a comparison of soap and rest services. Mod. Appl. Sci. **12**(3), 175 (2018)
18. Sohan, S.M., Anslow, C., Maurer, F.: A case study of web API evolution. In: 2015 IEEE World Congress on Services. IEEE (2015)
19. Archip, A., Amarandei, C.M., Herghelegiu, P.C., Mironeanu, C.: RESTful web services-a question of standards. In: 2018 22nd International Conference on System Theory, Control and Computing (ICSTCC), pp. 677–682. IEEE, October 2018
20. Noura, M., Atiquzzaman, M., Gaedke, M.: Interoperability in internet of things: taxonomies and open challenges. Mob. Netw. Appl. **24**, 796–809 (2019)
21. Michel, F., Faron-Zucker, C., Corby, O., Gandon, F.: Enabling automatic discovery and querying of web APIs at web scale using linked data standards. In: Companion Proceedings of the 2019 World Wide Web Conference, pp. 883–892, May 2019
22. Ozdemir, E.: A general overview of RESTful web services. Applications and approaches to object-oriented software design: emerging research and opportunities, pp. 133–165 (2020)
23. Coarfa, C., Druschel, P., Wallach, D.S.: Performance analysis of TLS web servers. ACM Trans. Comput. Syst. (TOCS) **24**(1), 39–69 (2006)
24. Chakraborty, M., Kundan, A.P.: Grafana. Monitoring Cloud-Native Applications, pp. 187–240. Apress, Berkeley, CA (2021)
25. Dogan, J.: RAKYLL/Hey: HTTP Load Generator, ApacheBench (AB) Replacement. GitHub, Rakyll. https://github.com/rakyll/hey/
26. Deliver Fast and Reliable Digital Experiences with K6. k6, K6 Grafana Labs. https://k6.io/deliver-fast-and-reliable-digital-experiences-with-k6/

27. Khan, R., Amjad, M.: Web application's performance testing using HP LoadRunner and CA Wily introscope tools. In: 2016 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, pp. 802–806 (2016). https://doi.org/10.1109/CCAA.2016.7813849
28. Harrold, M.J.: Testing: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering (2000)
29. Jiang, Z.M., Hassan, A.E.: A survey on load testing of large-scale software systems. IEEE Trans. Softw. Eng. **41**(11), 1091–1118 (2015). https://doi.org/10.1109/TSE.2015.2445340
30. Apache MPM Common Directives. mpm_common - Apache HTTP Server Version 2.4, The Apache Software Foundation. https://httpd.apache.org/docs/2.4/mod/mpm_common.html#maxrequestworkers
31. NGINX - Core Functionality. NGINX. http://nginx.org/en/docs/ngx_core_module.html#worker_connections
32. Malik, H., Jiang, Z.M., Adams, B., Hassan, A.E., Flora, P., Hamann, G.: Automatic comparison of load tests to support the performance analysis of large enterprise systems. In: 2010 14th European Conference on Software Maintenance and Reengineering, Madrid, Spain, pp. 222–231 (2010). https://doi.org/10.1109/CSMR.2010.39
33. Malik, H., Hemmati, H., Hassan, A.E.: Automatic detection of performance deviations in the load testing of large scale systems. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE (2013)
34. Hasanpuri, V., Diwaker, C.: Comparative analysis of techniques for big-data performance testing. In: 2022 Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC). IEEE (2022)