École Polytechnique de Montréal

LOG8371E: Software Quality Engineering

Second travaux pratique (TP2)

Professor: Rhouma Naceur

Team members

Student ID	Names
2313663	Xi-Zhen Wang
2246556	Abid Ullah Khan
2316903	Shitian Li
2313674	Fábio Akira Yonamine
2317239	Roberto Molina

Montréal, Quebec
2023

Table of Contents

I. Abstract **2**
II. Introduction **2**
III. Updated Quality Plan **4**
A. Quality Goals 4
B. Assurance Strategies 4
IV. Profiling Approach **4**
V. Load Testing **8**
A. Scenarios of Load Testing 8
B. User Manual 9
C. Report the Results 10
D. Discussion 10
VI. Approach for Performance Improvement **11**
VII. Comparison **12**
VIII. Conclusion **14**
IX. References **14**
X. Apendix **15**
A. Quality Plan Goals 15

I. Abstract

In the dynamic world of software development, our second Technical Project (TP) focuses on achieving optimal performance. We embark on a journey, exploring software performance intricacies. Our path involves defining objectives, measuring resource utilization, and optimizing performance using JMeter and JProfiler. We conduct load testing following a carefully crafted quality plan, ensuring clear objectives and consistent evaluation. Our commitment to enhancing performance is evident in our approach, which includes best practices like gradual load increases, efficient resource utilization, and results management. With JProfiler, we delve into profiling, unearthing performance bottlenecks and transformation opportunities. By selecting the right tools and following best practices, we are well-equipped to navigate the evolving landscape of software performance.

II. Introduction

In the dynamic realm of software development, the pursuit of optimal performance stands as a paramount endeavor. With the commencement of our second Technical Project (TP), we are embarking on an immersive journey designed to arm us with the indispensable skills and knowledge needed to navigate the intricacies of software performance. Through this TP, we are presented with an invaluable opportunity to delve into the multifaceted domain of software performance. Our exploration encompasses a wide range of facets, including defining performance objectives, measuring resource utilization, and employing verification and optimization techniques through profiling and load testing. As businesses increasingly rely on software solutions to drive innovation, efficiency, and competitiveness across industries, understanding the factors that influence software development costs and performance becomes essential. In this context, the ever-evolving landscape of software performance takes center stage as a critical focus area for our project, TP2. Our journey within this intricate world places tools like JMeter and JProfiler in the spotlight. In our relentless pursuit of performance optimization, we turn to JMeter, a versatile tool meticulously designed to subject the software application to a battery of load scenarios [1]. Load testing is a pivotal exercise allowing us to scrutinize the system's resilience under varying levels of demand. JMeter offers us a systematic framework to measure the application's response, CPU utilization, RAM consumption, and other vital performance metrics as the software's load fluctuates from baseline conditions to scenarios of increased and exceptional demand [1-2]. Concurrently, we meticulously craft a quality plan that defines the performance goals and criteria by which we will assess the application's performance. This quality plan acts as a guiding beacon throughout our project, ensuring the objectives remain crystal clear and the evaluation process remains consistent [1].

To enhance our JMeter performance testing, we follow best practices such as avoiding running production load tests in GUI mode, employing response assertions for every request, and double-checking thread and thread group counts [2]. Further recommendations include gradually increasing the simulated load to prevent overwhelming the tested server, distributing the load across average machines, refraining from functional mode, and storing results in CSV JTLs instead of XML [3]. It's also essential to monitor JMeter logs during testing and use naming conventions for test elements to facilitate quick correlation of results with business transactions and requests [3]. Profiling emerges as another pivotal process, revealing the inner workings of the software application. Here, JProfiler, a Java profiler tool, takes the spotlight, helping us analyze performance bottlenecks, memory leaks, CPU loads, and resolve threading issues [4]. The profiling process involves a meticulous examination of the software's execution, empowering us to identify areas in dire need of performance enhancement [4]. JProfiler's CPU profiling feature brings to light the most resource-intensive sections of the application, paving the way for targeted improvements [5]. This process not only exposes the application's shortcomings but also uncovers opportunities for transformation [4]. To optimize the profiling process, we heed the recommendation of selecting the correct Java profiler at the outset of the development process [6]. JProfiler stands as a top choice for many developers due to its intuitive user interface, offering interfaces for performance and memory snapshots, as well as thread profiling for both local and remote applications [6]. It boasts nearly all the features found in VisualVM, including control over system performance, memory profiling, anticipated memory leak fixes, and thread profiling [6]. Configuring sessions in JProfiler is straightforward, and third-party integrations simplify the initial steps. Additionally, JProfiler offers a range of probes that display high-level data from interesting subsystems in the JRE, including JDBC, JPA/Hibernate, JSP/Servlets, JMS, web services, and JNDI.

III. Updated Quality Plan

A. Quality Goals

Table 1 - Definition of quality characteristics regarding performance

Quality Characteristics	Sub-Characteristics	Quality Measure	Goals
Performance efficiency	**Capacity**	The degree to which the maximum limits of a product or system parameter meet the requirements	The software can handle 44000 requests without crashing
	Resource utilization	The degree to which the amounts and types of resources used by a product or system when performing its functions meet the requirements	CPU and RAM reach a peak usage of less than 90% during the increased load

B. Assurance Strategies

In order to observe the capacity of the program we will observe how the system responds to the exceptional increased load using JMeter. This load submits 44000 requests, that is the number of requests that we want that the system handles without crashing. So if the system supports the exceptional increased load without crashing we can deduce that our goal has been validated.

To assure if our goal for resource utilization has been met, we have to observe how the system behaves in the increased load in JMeter. After running the load we can observe our report the results of how the CPU and RAM have responded to the increased load. If the system ends up using less than 90% of the CPU and RAM, we can conclude that our goal has been certified.

IV. Profiling Approach

For the profiling section, we have downloaded the VisualVM application, which allows us to get statistics regarding the CPU and RAM usage of PetClinic. As VisualVM itself does not generate any load to the application under testing, we also have to run JMeter, which will artificially generate the requests and, therefore, enable the analysis of the app's performance. It's important to note that, during the tests, we would always connect our laptops to a wall plug and also close any applications not related to this study itself, because modern computers decrease battery usage by lowering performance, for example, and to avoid excessive memory usage by other apps.

First, to define how many users are equivalent to each of the load types, we ran JMeter with different numbers of concurrent users, increasing that until our computers reached as closest as possible to 100% of global CPU usage, as the instructions for this practical assignment states that we should focus on CPU usage for profiling.

Once we got the amount of concurrent users that makes our global CPU usage reach the closest to 100%, we defined it as the "Exceptionally Increased" load, and from there, we subsequently defined the needed remaining load types. Additionally, we took notes of the CPU usage taken by the PetClinic app itself and also memory usage, which will be important for the load testing and improvement sections of this report. The table below contains all values collected during the profiling process.

Table 2 - Definition of load types and observed resource usage

Load type	Number of users	JProfiler Process Load (CPU Usage for Petclinic) [%]	Petclinic Memory Usage
Reduced	11	16%	0.91GB
Average	30	38.58%	0.55GB
Increased	100	68.75%	0.56GB
Exceptionally Increased	2000	89.05%	1.24GB

Setting up the JMeter tool to run with 11 concurrent users, which would equals to reduced load, we observe the following pattern for CPU usage, generated by JProfiler, in Figure 1. It shows a peak usage of 60%.

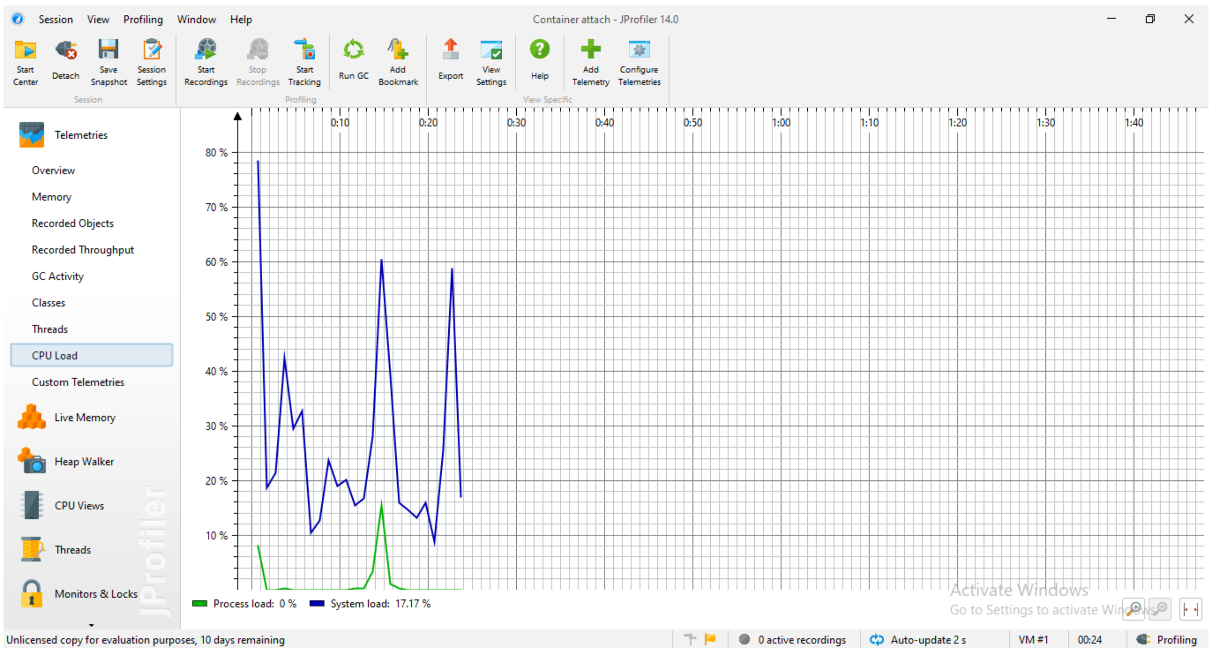


Figure 1 - CPU usage of 11 concurrent users (Reduced Load)

For the average load, which was defined as 30 concurrent users, in Figure 2, below, we observed a peak of 94% of CPU usage in JProfiler.

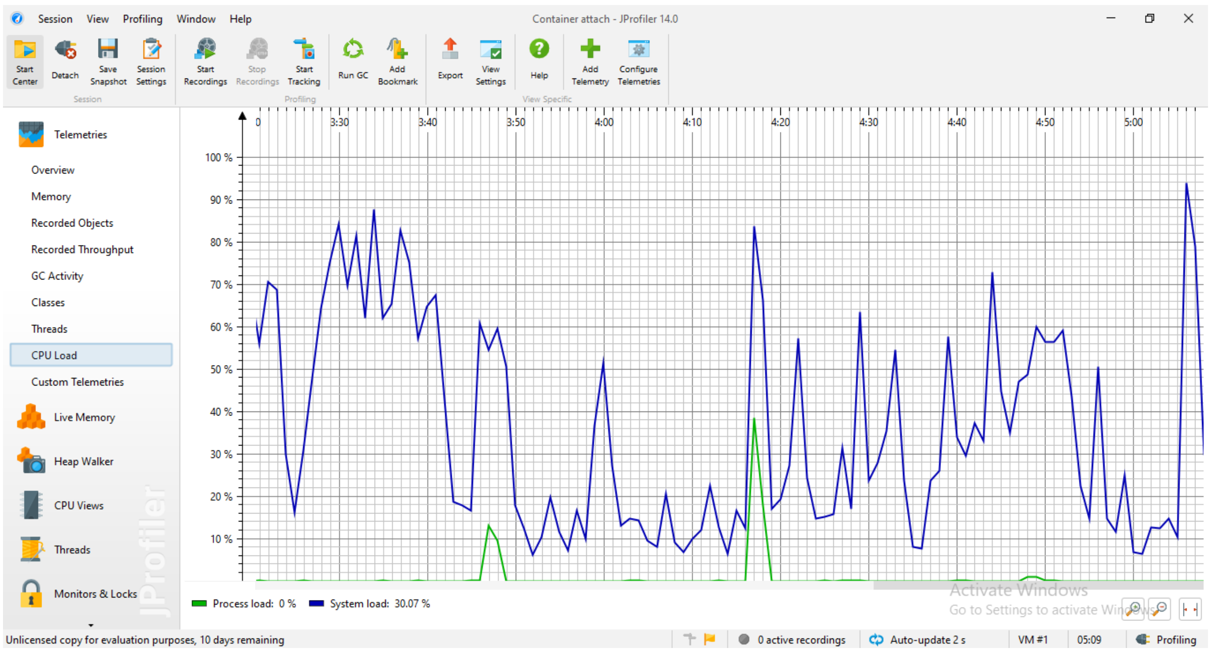


Figure 2 - CPU usage of 30 concurrent users (Average Load)

For the increased load, which was defined as 100 concurrent users, in Figure 3, below, we observed a peak of 100% of CPU usage in JProfiler.

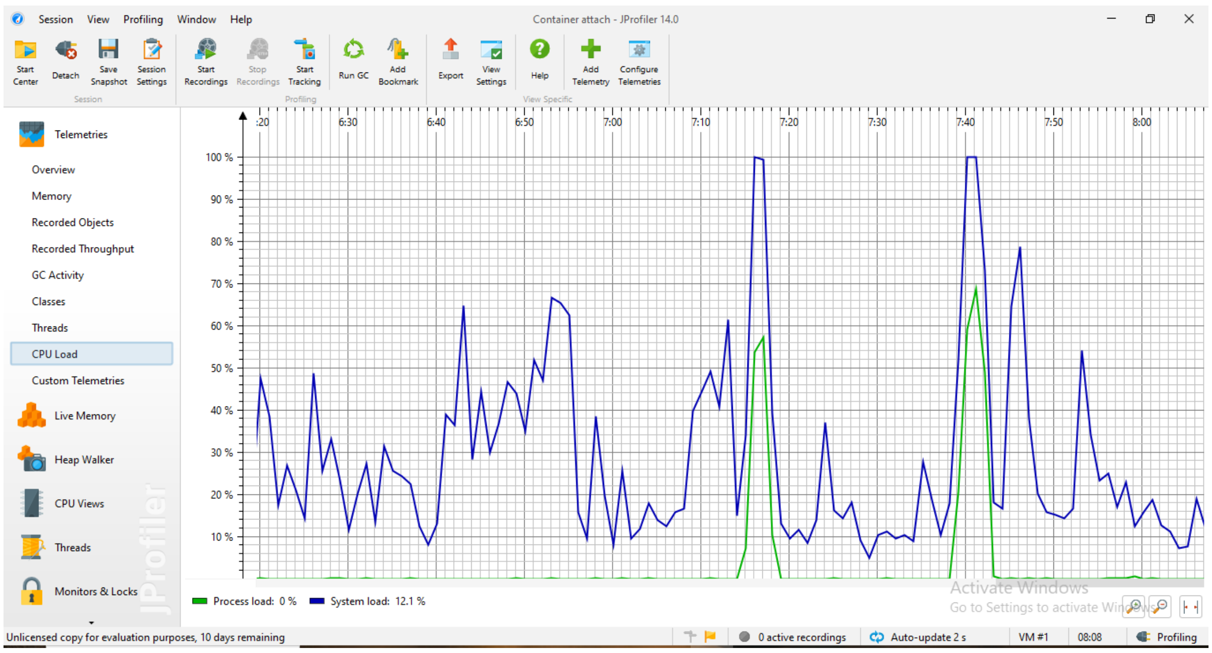


Figure 3 - CPU usage of 100 concurrent users (Increased Load)

Finally, for the exceptionally increased load, which was defined as 2000 concurrent users, in Figure 4, below, we observed a peak of 100% of CPU usage in JProfiler.

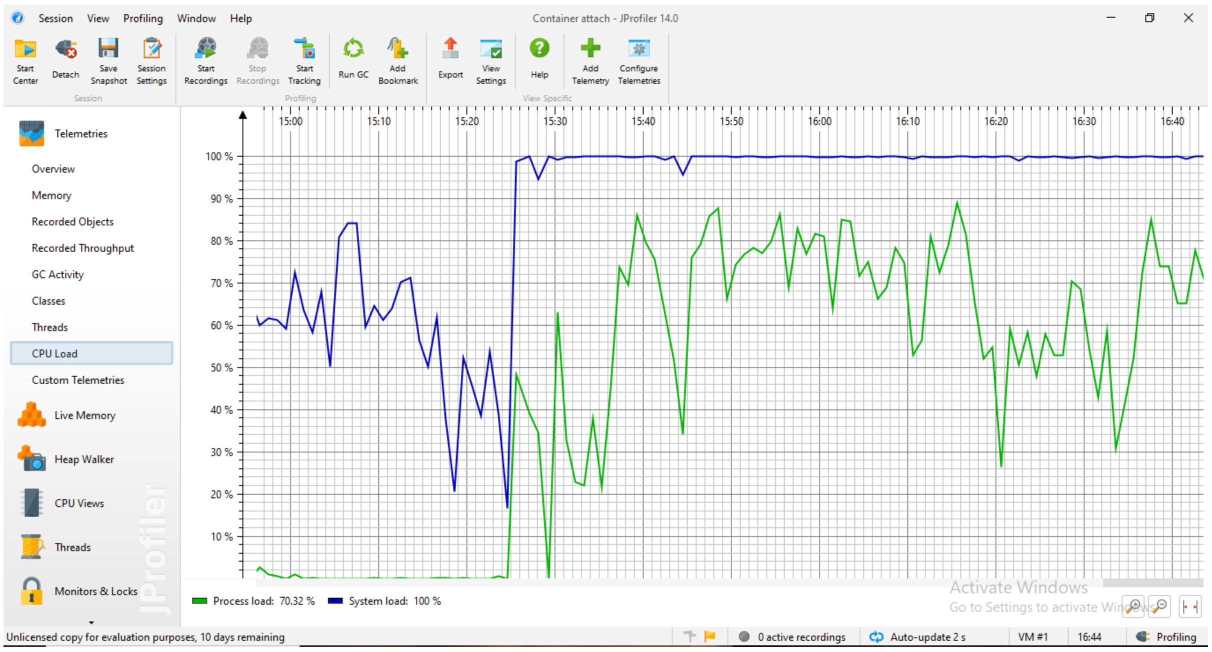


Figure 4 - CPU Usage of 2000 concurrent users (Exceptionally Increased Load)

It is important to state here that, during our testings, the recorded values of CPU and memory greatly varied between tests, even when using the same computer and the same number of concurrent users (load level), without any other apps concurrently running. We are yet to determine why the computer doesn't follow a clear pattern in our tests, but we are suspicious it may be related to processor throttling, as all of our members were using laptops to run these tests, and our laptop's fans started to run at higher speeds right after we start running these tests. Thus, it's not expected to obtain the same results in later testing.

In more details, we first downloaded and ran the following commands in a Terminal screen, in order to run PetClinic (as instructed in the README file, available on the GitHub page):

```
git clone https://github.com/spring-petclinic/spring-petclinic-rest/git
cd spring-petclinic-rest
./mvnw spring-boot:run
```

After that, we proceeded to download and execute VisualVM, which is the app that we will be using to do the profiling procedure. With the PetClinic application already running on the Terminal, we select, under the Java Virtual Machines (JVM) running locally, the PetClinic Java application running, and choose the "Monitor", where we have all the graphs for CPU, Memory and Threads.

Finally, we launch JMeter, where we are able to run the test plan that was set up, as it is explained later in this report.

V. Load Testing

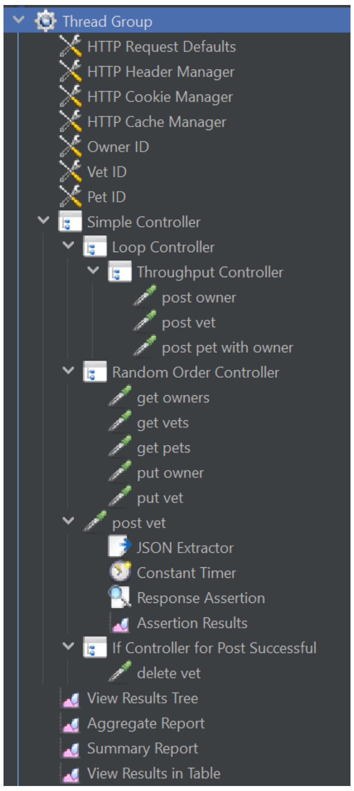
A. Scenarios of Load Testing

In this section, we will explain the load testing scenario that has been designed. We choose the 9 APIs to test and set the 5 controllers to manage the logic. The hierarchical structure of this configuration can be visually represented as demonstrated in Figure 5.

First, we wrap the entire set of controllers and APIs within a "Simple Controller". Subsequently, the "Throughput Controller" controls the volume of executions for its child requests. Within this controller, we integrated the three POST APIs to create an owner, vet and pet. Furthermore, we have configured the "Throughput Controller" with a value of 10%, which means in a script where a particular request is intended to be invoked a total of 100 times, the implementation of this controller will effectively reduce its execution to 10 times. To further structure and control the execution flow, we have enclosed this setup within a "Loop Controller" with the loop set to execute two times, which in turn gives us better control over the number we've defined.

In the next stage, we establish the "Random Order Controller" to orchestrate the execution of GET and PUT methods, enabling us to retrieve owner, pet, and vet data, as well as modify the values for owner and vet data. This controller introduces a level of unpredictability by executing the child requests in a random sequence during each test iteration, which enhances the realism of the load testing scenario.

Finally, we defined the "If Controller" to ensure the orderly deletion of data that has been successfully created. The expression `${JMeterThread.last_sample_ok}` within the "If Controller" can dictate that the DELETE method will only be executed when the preceding request has been executed successfully. This conditional control mechanism adds an extra layer of precision to the testing process, guaranteeing that the data deletion process is carried out only when it is logically sound and warranted based on the success of the previous operation.



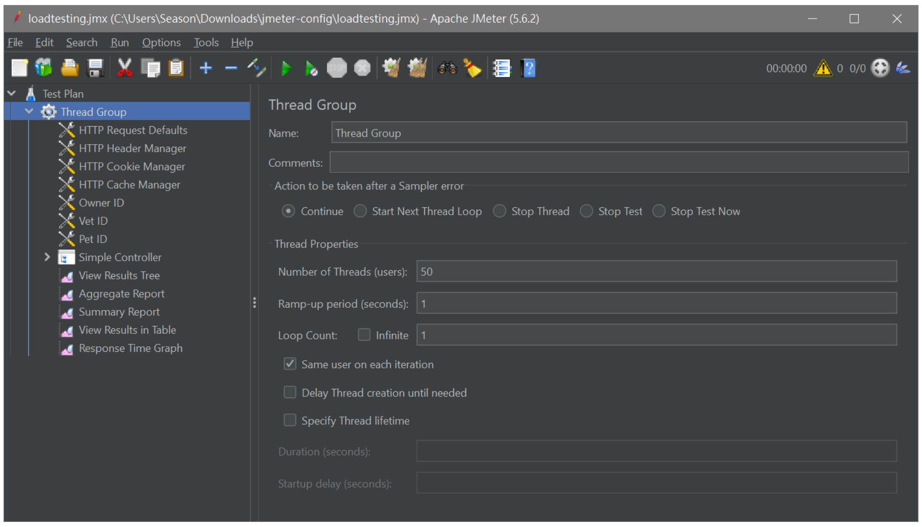
```
Thread Group
    HTTP Request Defaults
    HTTP Header Manager
    HTTP Cookie Manager
    HTTP Cache Manager
    Owner ID
    Vet ID
    Pet ID
    Simple Controller
        Loop Controller
            Throughput Controller
                post owner
                post vet
                post pet with owner
        Random Order Controller
            get owners
            get vets
            get pets
            put owner
            put vet
        post vet
            JSON Extractor
            Constant Timer
            Response Assertion
            Assertion Results
        If Controller for Post Successful
            delete vet
    View Results Tree
    Aggregate Report
    Summary Report
    View Results in Table
```

Figure 5: Hierarchy of JMeter configuration

B. User Manual

To facilitate users access to the JMeter configuration, we have uploaded the JMeter configuration to a GitHub repository to make it easier for users to retrieve and allows users to stay updated with new versions of the configuration. Below, we outline the steps required to execute the JMeter configuration.

First, we run the JMeter batch file. Then, clone the repository from [jmeter-config (github.com)] to the local environment. Once the repository is successfully cloned, open the "loadtesting.jmx" in JMeter application.



loadtesting.jmx (C:\Users\Season\Downloads\jmeter-config\loadtesting.jmx) - Apache JMeter (5.6.2)

```
Test Plan
    Thread Group
        HTTP Request Defaults
        HTTP Header Manager
        HTTP Cookie Manager
        HTTP Cache Manager
        Owner ID
        Vet ID
        Pet ID
        Simple Controller
        View Results Tree
        Aggregate Report
        Summary Report
        View Results in Table
        Response Time Graph
```

Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error: [✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 50

Ramp-up period (seconds): 1

Loop Count: ☐ Infinite 1

☑ Same user on each iteration

☐ Delay Thread creation until needed

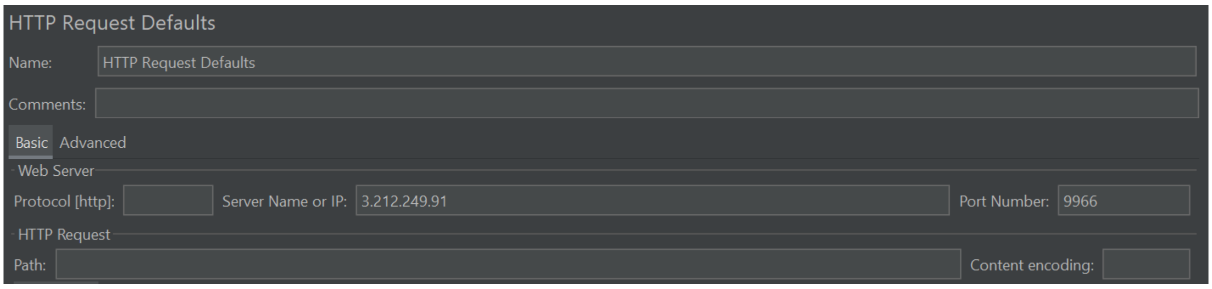
☐ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

Figure 6: JMeter interface

With the configuration file open, users must modify the "Server Name or IP" and "Port Number" found within the "HTTP Request Defaults" (as illustrated in Figure 7). This is a crucial step to ensure that JMeter can establish the connection with your server.


HTTP Request Defaults

Name: HTTP Request Defaults

Comments:

Basic Advanced

Web Server

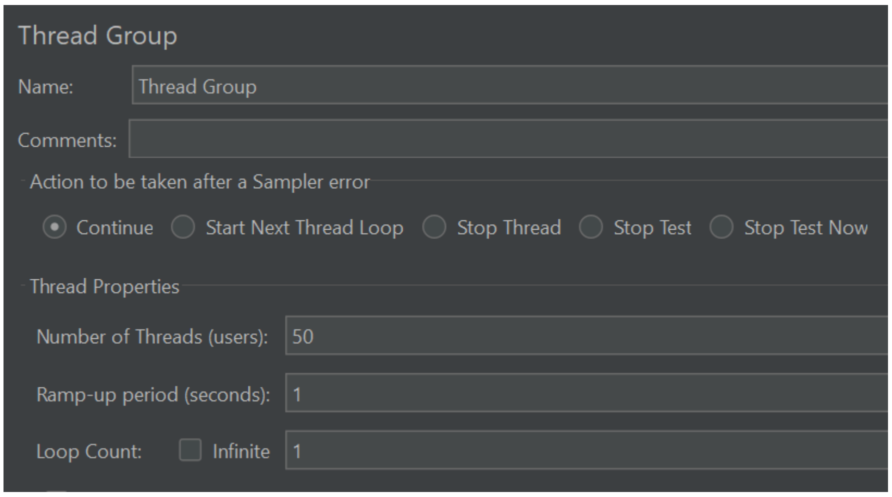
Protocol [http]:   Server Name or IP: 3.212.249.91   Port Number: 9966

HTTP Request

Path: Content encoding:

Figure 7: HTTP Request Defaults settings in JMeter

Subsequently, users can customize the "Thread Properties" to align with the specific testing requirements in "Thread Group" (as shown in Figure 8).


Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 50

Ramp-up period (seconds): 1

Loop Count: Infinite 1

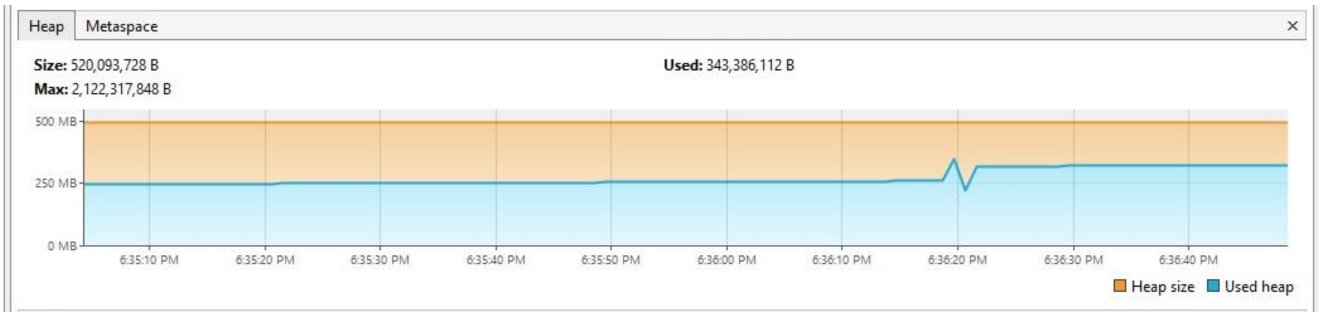
Figure 8: Thread Group settings in JMeter

Also, it is imperative the confirm the presence of three CSV files within the same folder, specifically named as "vet_id.csv", "owner_id.csv", and "pet_id.csv", JMeter relies on these files as data sources, retrieving data for utilization as parameters in API requests. Finally, having completed the aforementioned setup steps, users can initiate the testing process by clicking the "Start" button. The results of the test will be available for review in the generated report.

C. Report the Results

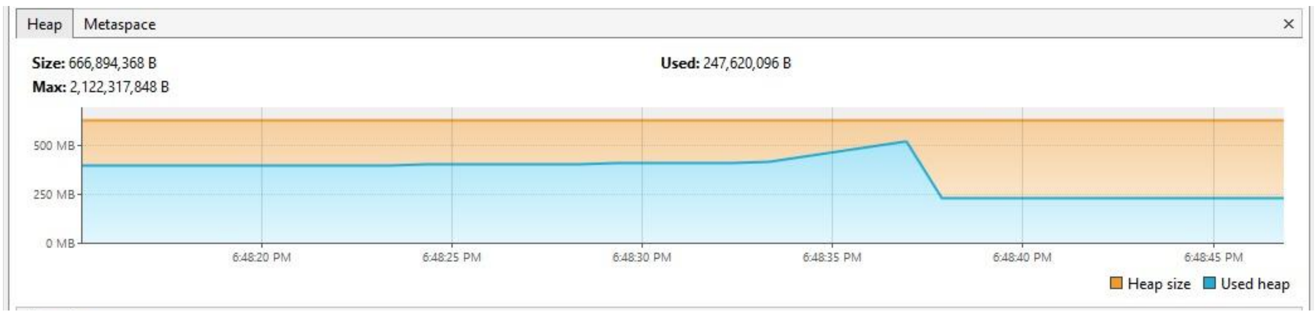
After running the load tests with JMeter, we were able to collect performance measures with JMeter and JProfiler. For the CPU usage, we have already reported and shown the results on table 2 and figures 1, 2, 3 and 4, in section IV, Profiling Approach.

For memory usage, we've obtained the following results in figures 9, 10, 11 and 12, below.


Heap | Metaspace

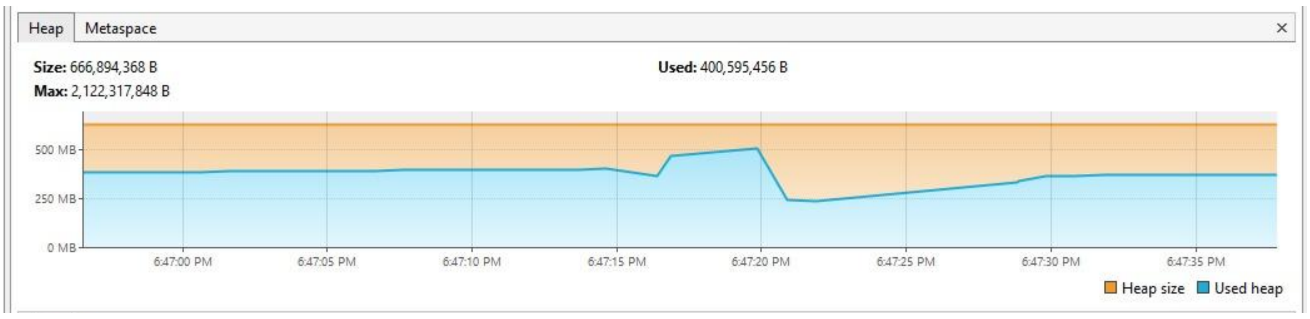
Size: 520,093,728 B **Used:** 343,386,112 B
Max: 2,122,317,848 B

Figure 9: Memory and thread count for 11 concurrent users


Heap | Metaspace

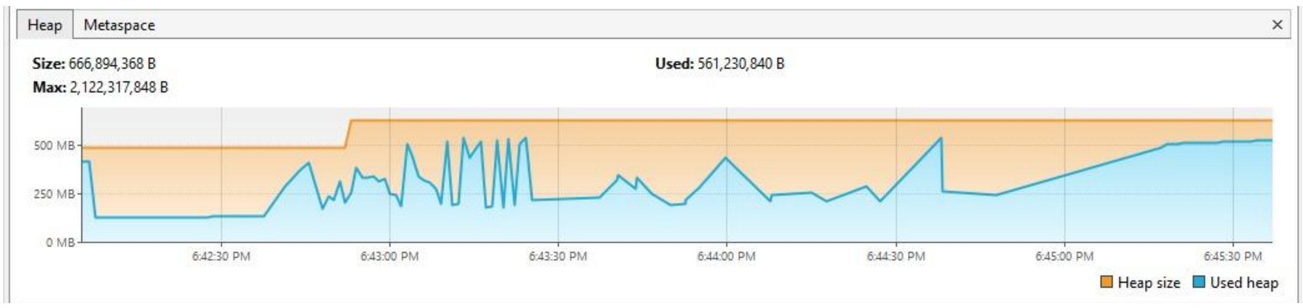
Size: 666,894,368 B **Used:** 247,620,096 B
Max: 2,122,317,848 B

Figure 10: Memory and thread count for 30 concurrent users


Heap | Metaspace

Size: 666,894,368 B **Used:** 400,595,456 B
Max: 2,122,317,848 B

Figure 11: Memory and thread count for 100 concurrent users


Heap | Metaspace

Size: 666,894,368 B **Used:** 561,230,840 B
Max: 2,122,317,848 B

Figure 12: Memory and thread count for 2000 concurrent users

Finally, for the response times obtained with JMeter in figure 13, below, that represent the response times for 11, 30 and 100 concurrent users, from left to right, we observe that the times stay almost constant, even though we slightly increased the load. For the exceptionally increased load, though, shown in figure 14 with all the execution times for all four loads, we observe that the response times increase greatly, because the computer struggles to process all requests accordingly.

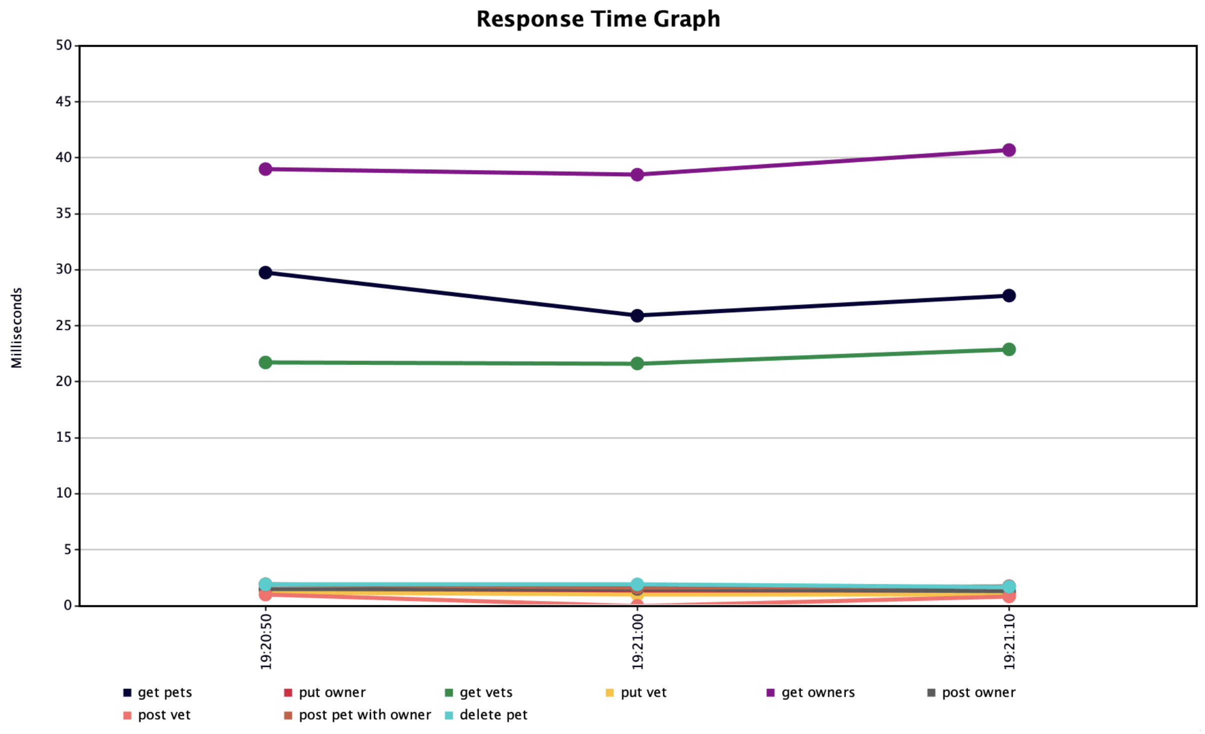


Figure 13: Response times for 11, 30 and 100 concurrent users

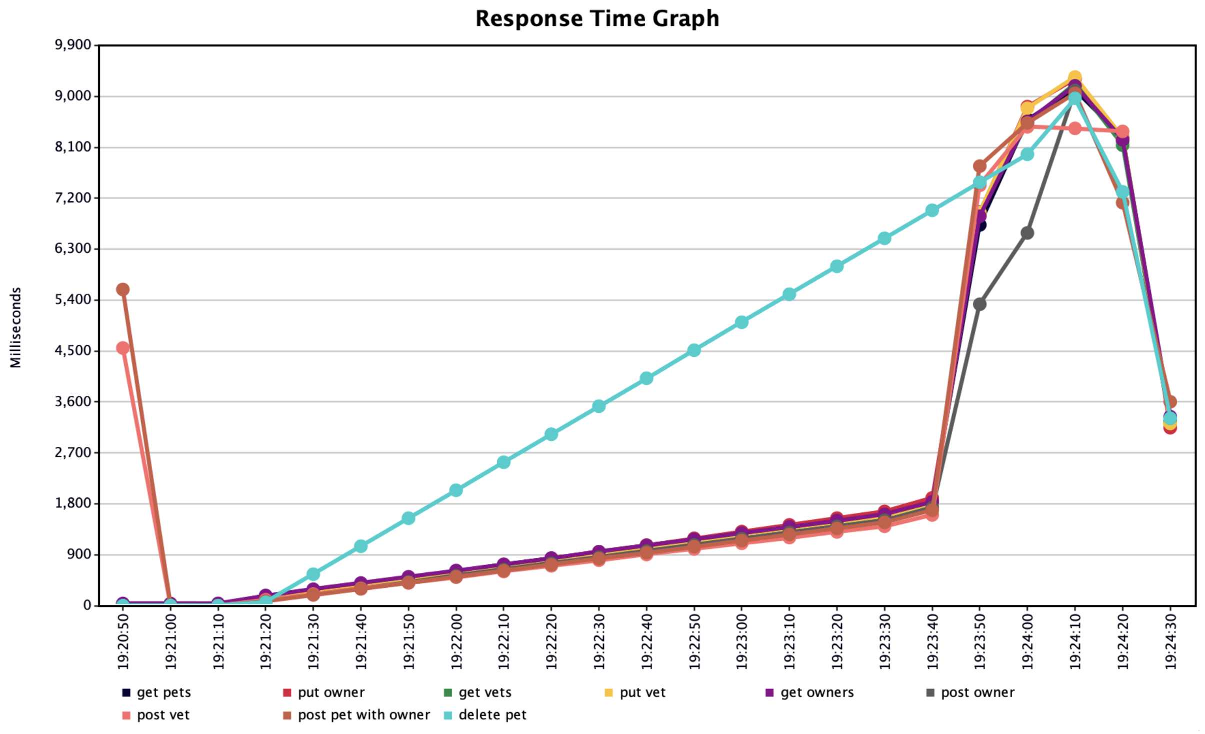


Figure 14: Response times for 11, 30, 100 and 2000 concurrent users

D. Discussion

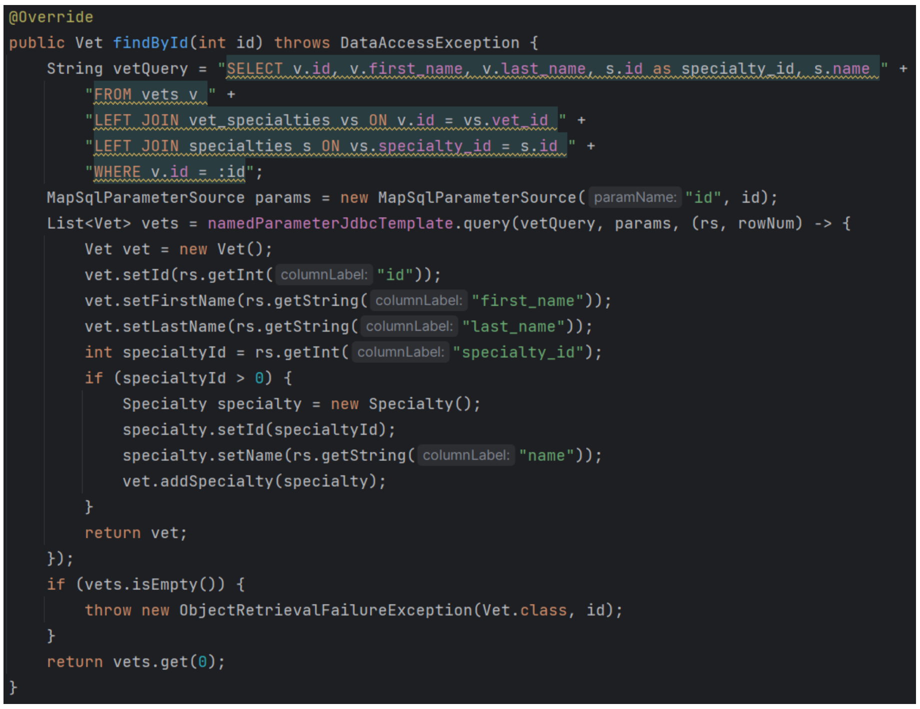
In our pursuit of optimal software performance, we delved into a comprehensive analysis of various aspects in our TP2. Our journey began with the development of a quality plan, where we meticulously outlined our performance objectives and criteria for evaluation. This plan served as our guiding beacon, ensuring that our objectives remained clear and our evaluation process remained consistent. Through this systematic approach, we believe we were able to establish a strong foundation for our performance optimization efforts. A significant part of our work revolved around load testing, a crucial exercise that allowed us to scrutinize the resilience of our software under varying levels of demand. By conducting these load tests using JMeter, we were able to measure the PetClinic's response, CPU utilization, RAM consumption, and other essential performance metrics. Following best practices, such as gradual load increases and efficient resource utilization, we aimed to ensure that the software could withstand real-world demands. This comprehensive load testing approach not only exposed the PetClinic's vulnerabilities but also provided us with valuable insights for performance enhancement. Furthermore, we turned our attention to profiling with the aid of JProfiler. This tool helped us uncover performance bottlenecks, memory leaks, CPU loads, and threading issues. By conducting a detailed examination of the PetClinic's execution, we were able to pinpoint areas in need of improvement. We agree that JProfiler's CPU profiling aspect, which identifies resource-intensive portions of the application, played a pivotal role in focusing our enhancement efforts.

We also believe that the selection of the correct Java profiler at the outset of the development process was crucial. JProfiler, with its intuitive user interface and comprehensive features, emerged as our top choice for profiling, allowing us to analyze performance bottlenecks effectively. Throughout our project, we conducted load testing and profiling meticulously, striving to optimize software performance. We agree that by adhering to best practices, following a well-defined quality plan, and using the right tools, we were well-equipped to navigate the ever-evolving landscape of software performance. Our work not only exposed areas in need of improvement but also provided valuable opportunities for transformation and enhancement in the pursuit of optimal software performance.

VI. Approach for Performance Improvement

After analyzing the results obtained in the previous section, it becomes evident that the system places a significant demand on the CPU, especially when compared to its memory consumption. As a result, it is reasonable to deduce that the bottleneck affecting system performance primarily resides in the CPU.

In response to this identified bottleneck, we have initiated efforts to enhance the efficiency of the requests being made during our testing. Specifically, we have concentrated on optimizing the ***calls*** related to vets and pet owners, which means refactoring six of the nine functions we use. To achieve these improvements, we have undergone substantial changes in the way these functions access and interact with the database. More concretely, we changed the repository files. One example is the findById function in VetRepository:



```java
@Override
public Vet findById(int id) throws DataAccessException {
    String vetQuery = "SELECT v.id, v.first_name, v.last_name, s.id as specialty_id, s.name " +
        "FROM vets v " +
        "LEFT JOIN vet_specialties vs ON v.id = vs.vet_id " +
        "LEFT JOIN specialties s ON vs.specialty_id = s.id " +
        "WHERE v.id = :id";
    MapSqlParameterSource params = new MapSqlParameterSource( paramName: "id", id);
    List<Vet> vets = namedParameterJdbcTemplate.query(vetQuery, params, (rs, rowNum) -> {
        Vet vet = new Vet();
        vet.setId(rs.getInt( columnLabel: "id"));
        vet.setFirstName(rs.getString( columnLabel: "first_name"));
        vet.setLastName(rs.getString( columnLabel: "last_name"));
        int specialtyId = rs.getInt( columnLabel: "specialty_id");
        if (specialtyId > 0) {
            Specialty specialty = new Specialty();
            specialty.setId(specialtyId);
            specialty.setName(rs.getString( columnLabel: "name"));
            vet.addSpecialty(specialty);
        }
        return vet;
    });
    if (vets.isEmpty()) {
        throw new ObjectRetrievalFailureException(Vet.class, id);
    }
    return vets.get(0);
}
```

Figure 15: function findById refactored

In the figure above, it can be observed that one of the functions has been modified. This function before made several simple SQL queries to find a vet by its ID. To reduce the number of accesses to the database, we have consolidated them into a single, more intricate SQL query that serves the same purpose. This strategic adjustment not only streamlines the process but also reduces the number of database accesses, contributing to enhanced system performance.

These modifications aim to reduce the computational load on the CPU and enhance the overall performance of the system. By addressing the root causes of CPU-intensive operations, we anticipate a more responsive and efficient system, ultimately delivering a better user experience.

Note: All the modifications can be observed on GitHub:
https://github.com/Rmolina2002/LOG8371E-TP2.git

VII. Comparison

After refactoring the code we measure the CPU usage again, to observe how our changes have affected the performance of the system. Some of the results obtained are show in the next figure:

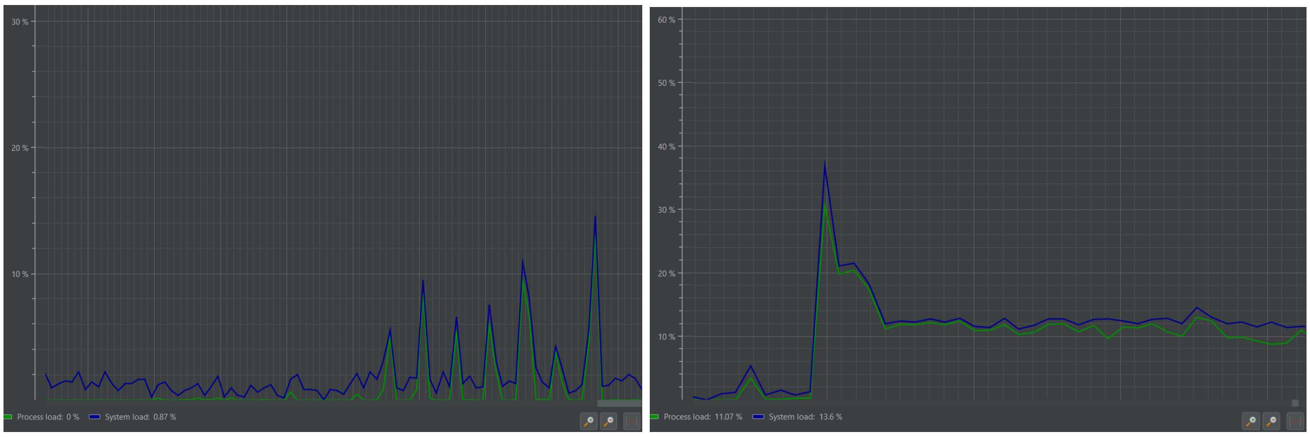


Figure 16: Petclinic CPU usage after refactor for reduced load and for average load

After analyzing all the data, to observe better the differences between the old system and the new one we generated a table that compares both CPU usage

Table 3 - Comparison of the CPU usage

Load type	Observed Petclinic CPU Usage **Before** [%]	Observed Petclinic CPU Usage **After**[%]
Reduced	16%	13.08%
Average	38.58%	30.63%
Increased	68.75%	61.12%
Exceptionally Increased	89.05%	76.31%

The data presented in the table unmistakably reflects substantial enhancements in CPU utilization across a spectrum of scenarios. On average, these enhancements register at an impressive approximate rate of 6.25%. However, there is an additional and intriguing insight that can be gleaned from this dataset.

It is evident that the system's response improves as the workload intensifies. This positive correlation between system performance and increasing load is particularly notable when we direct our attention to two distinct load categories: Reduced load and Exceptionally increased load. Under reduced load, we observe a commendable 2.92% improvement in system usage, while the system continues to deliver an equally remarkable 12.74% enhancement even under the pressure of exceptionally increased load. This trend underscores the system's robustness and scalability.

This multifaceted analysis affirms that not only have significant enhancements been achieved in CPU utilization, but the system also exhibits remarkable resilience in the face of varying workloads. As a result, it is evident that the bottleneck has been effectively mitigated.

VIII. Conclusion

The TP2 focused on software performance optimization, driven by meticulous planning and the use of tools like JMeter and JProfiler. Load testing with JMeter provided valuable insights into performance metrics, while a quality plan ensured clarity in our objectives. Profiling with JProfiler identified bottlenecks and optimization opportunities. It is worth noting that, even before implementing improvements, all our quality goals were diligently addressed and met. Profiling the software with JProfiler enabled us to pinpoint bottlenecks and identify numerous optimization opportunities. This experience reinforced the importance of tool selection and best practices, equipping us to excel in the dynamic software development landscape and drive innovation and efficiency in the industry.

IX. References

[1] https://apidog.com/blog/how-to-optimize-jmeter-performance-testing/
[2] https://www.redline13.com/blog/2020/09/8-tips-for-optimizing-jmeter-test-plans/
[3] https://octoperf.com/blog/2017/10/12/optimize-jmeter-for-large-scale-tests/
[4] https://www.hcltech.com/blogs/jprofiler-tool-capture-performance-bottlenecks
[5] https://www.ej-technologies.com/resources/jprofiler/help/doc/main/cpu.html
[6] https://stratoflow.com/guide-to-java-profilers/

X. Appendix

A. Quality Plan Goals

Quality Characteristics	Sub-Characteristics	Quality Measure	Goals
Performance efficiency	**Capacity**	The degree to which the maximum limits of a product or system parameter meet the requirements	The software can handle 44,000 requests and 2000 users without crashing
	Resource utilization	The degree to which the amounts and types of resources used by a product or system when performing its functions meet the requirements	CPU and RAM reach a peak usage of less than 90% during the increased load