Table of Contents

Chapter 1. Introduction

1.1. Motivation

Nowadays, Application Performance Monitoring (APM) has received great attention as it represents an area of information technology (IT) that focuses on making software application programs perform as expected, provide users with a quality end-user experience and thus enlarge its usage and business worldwide.

Consequently, having the license thesis as a basis web application upon which Application Performance Monitoring might be developed, served as a proper research-oriented topic for the dissertation thesis.

As far as the license thesis concerns, it might be summarized as the development of an electronic Scrum tool which uses Java technologies, its purpose being to define and construct a system capable of providing a tool used as an agile project organizer. More precisely, the application was intended to offer the functionalities of an online tool which allows the persons involved in the Scrum process to perform it regardless of any inconveniences such as physically distributed teams, unreachable customer or any other factors that may negatively affect the Scrum process and thus the corresponding software project.

By developing the previously mentioned electronic tool, the traditional taskboard, represented inside companies as a whiteboard offering relevant information about the project's progress and the implementation of the project's tasks, is replaced by an online one. The latter provides the same tasks management functionalities and is reachable by all the stakeholders involved in the Scrum process, as well as the Customer and other persons interested in the project's progress. This way, the developers, the Scrum Master and the Product Owner are given the opportunity to visualize and update the taskboard any time.

Consequently, by considering this project as an electronic Scrum tool, there are other functionalities provided by the aforementioned methodology, such as a burndown chart representation corresponding to every Sprint which is automatically generated by the application. Furthermore, as in the real case, the Scrum Master has the ability to post comments to the tasks which are in the "In Review" panel of the taskboard, that are accessible only to the implementer of the certain task. Also having an administrative role, the Scrum Master is enabled to perform CRUD (Create, Read, Update and Delete) operations on all the developers involved in the project's implementation and on the untaken tasks.

As the presented functionalities brought by the online Scrum tool are already available on the market, the license thesis project brought several substantial improvements by considering important aspects of the Scrum process not implemented yet by any other similar tool. One of them was based on the design and implementation of an algorithm which analyses the skills required for implementing a certain task and the ones held by every team member and suggests the most suitable developer for implementing the task. This algorithm followed a complex reasoning process, also taking into account the availability of the suggested developer and the task's priority. Further on, another essential enhancement to a basic electronic Scrum tool is an optimization algorithm which proposes, at the end of each task an increase in the percentage of the technology/ technologies required for implementing this specific task for the developer.

Therefore, the succinct list of the license project's provided functionalities is as follows:

- Taskboard visualization and update
- Burndown chart visualization
- Obtain burndown chart description
- Obtain team information
- Posting and visualizing comments
- Developers and tasks management
- Developers schedule visualization
- Suggest developer for each untaken task
- Propose skill upgrades for each completed task

Therefore, starting from the described license thesis application, the intent of the research and investigation activities conducted on behalf of the dissertation paper, was to monitor and evaluate its performance characteristics through a series of specialized tools. Therefore, by means of these software products, a complete performance monitoring of the application was obtained. Following this initial, research-oriented phase of the dissertation thesis, come the ones focused on developing improvements to the actual version of the license thesis application without providing any new functionalities. These enhancements reside in different areas of the system and might be assessed together or in isolation. After each step, the same performance monitoring tools will be used to assess the optimized application and provide results gathered through a series of measurements.

1.2. Project context

In today's competitive landscape, software applications have become more and more complex, offering a vast variety of functionalities to a large heterogeneous group of customers. Keeping the customers satisfied and content with your business is ensured through a high user experience, also characterized by consistent speed and uptime. The art of managing the performance, availability, serviceability, response time and the overall user experience of software applications is defined by Application Performance Monitoring (APM), also referred as Application Performance Management [1].

The main job of the APM is to monitor the response time, throughput and several other performance metrics associated to entire infrastructure that supports a software application, providing a complete overview of potential bottlenecks and flaws that might disturb the expected operation [4]. More precisely, a suite of specialized tools are used to analyze and diagnose an application's speed, serviceability, operability and other performance metrics of interest in preserving the desired service behavior, as well as enlarging the number of clients. Moreover, a variety of strategies and approaches such as load testing, performed by using the simulation capabilities offered by performance measuring tools, synthetic and real-user monitoring, and root-cause analysis are intensively used in every day's practicing of APM.

Irrespective of the application performance monitoring and management software chose to help a business in gathering and evaluating different aspects related to an application's performance, an essential prerequisite is to ensure its need by researching some APM software use cases. If the conducted investigation reveals a need for APM, the following step is to assess the key features and functions that provided by the APM tools available on the market. Matching those key features with the organization's business needs is crucial in deciding the proper APM software which will best fit the needs. Fortunately, the list of available APM tools is continuously increasing, aiming to develop a large variety of business needs.

1.3. Project objectives

In the previously described context, the objectives defining the dissertation thesis rely in the area of Application Performance Monitoring/Management and targets maximizing the actual performance of an existent web application. The dissertation thesis development might be stated as a suite of dependent processes, expressed as well-defined methodologies, each of them following some pre-established activities, aiming to successfully reach a specific target.

More specifically, starting with the performance evaluation of the existent license thesis application, by means of dedicated tools, the initial state of the system is defined in terms of numeric figures and facts. By analyzing the web application capabilities from specific results associated to the obtained monitoring results, target goals might be formulated. An instance of such a particular goal may be defined as obtaining an improvement in the overall system in order to make it capable of handling a targeted number of concurrent users. This desired capability might be already available, but at a low performance standard, resulting in a large response time to serve an extensive number of simultaneous active customers. In this context, the desired achievement is to considerably improve the application's throughput and response time features in order to successfully handle all requests and maintain a pleasant user experience. Supporting a high amount of workload, here expressed as concurrent users, denotes the system's scalability. As an essential property of systems, the scalability represents a significant demand impacting the customer’s decision when selecting the suitability of a web application in fulfilling his/her needs. Similar to scalability property used in evaluating a web application’s behavior and capabilities, there are several other non-functional characteristics assessed which are intended to be optimized.

Further on from the initial system’s assessment phase, the subsequent stage in the thesis development reviews the possible solutions to be implemented and establishes the most suitable approach to be followed. Following this step comes the suite of performance improvement objectives, each of them following a designated methodology aiming to implement that target objective, thus correcting a possible defect and/or enhance the system’s capabilities. Furthermore, the completion of each improvement is followed by a performance analysis phase aiming to assess the obtained results by carrying out the same initial evaluation techniques.

The research and development activities performed on behalf of the dissertation thesis complete with an overall measurement of the established objectives, determining their suitability by comparing the workload invested in implementing them with the obtained outcome. Finally, the interconnection of all activities performed targets reaching a consistent improvement of the initial web application, thus enhancing its behavior without providing new functionalities.

To conclude this chapter, the dissertation paper is structured as follows: the subsequent chapter offers a description of the actual state of the art in the Application Performance Monitoring area, as well as associated theoretical considerations. Chapter 3 illustrates the tools that are mostly used for measuring performance metrics, with detailed description of the ones explored in the experiments performed, and the technologies used in the web application’s development. The main steps in the dissertation thesis’ research and development are briefly presented in Chapter 4, together with a detailed overview of the testing and experimental activities conducted. Finally, Chapter 5 summarizes the results analysis.

Chapter 2. State of the Art and Theoretical Considerations

2.1. APM

2.1.1. APM Meaning

Even if extensively used in software development literature, the APM (Application Performance Monitoring/Management) term is mistakenly perceived in many contexts. The erroneous aspect of the APM usage comes from its subjective interpretation which may lead from confusions and lack of consistency to even severe mistakes caused by possible misunderstandings. These possible confusions are rooted in slightly different understandings of the APM, which deviates from its essential usage of continuously evaluation of a software application's performance capabilities, by perceiving it from the metrics point of view (Application Performance Measurement) [1]. In this context, the APM meaning is shifted towards metrics and measurement outcomes provided by the application's monitoring.

Even a more significant variation from its appropriate definition is the notion of Portfolio Management used to describe APM in software industry. Its significance is about generating an overall high-level view of software applications in the monitored company's portfolio in order to gain an overview on how they are affecting business goals. Portfolio Management meaning is mostly used in the context of strategic planning and generating IT inventories.

What is APM?
Application Performance Management (APM) is the practice of monitoring the performance and availability of software applications.

Figure 2.1: APM Meaning

However, the Application Performance Monitoring/Management understanding of APM is the one considered in thesis' context, as it focuses on the interconnection of tools, processes and methodologies used to evaluate a system's quality features by taking into account performance-oriented metrics [4]. By using the information provided by continuous APM integration, particular problem-oriented strategies might be designed and further on followed so that to overcome the determined vulnerabilities impacting a system's functioning, a crucial aspect in maintaining users satisfied. The clear advantages of this approach are precision and time saving in operating on a web application's defects and flaws through precise correction at the burning point.

2.1.2. History of APM

According to Sydor [16], monitoring IT systems begun around the 1960s from the necessity to address a simple but essential use case scenario: before a user takes a long walk to the soda machine, is there in fact a soda available? The important part in this statement is represented by the mechanism to interrogate a device and exchange its status information, which has later evolved into *Simple Network Management Protocol* (*SNMP*). Nowadays, many systems world-wide take advantage of SNMP to assess the status of devices attached to the network. The omnipresence of this SNMP technology gave rise to *Availability Monitoring* (AM), a core responsibility of every modern IT corporation.

As stated in the previously mentioned book, IT organizations are measured in part on their *availability percentages*, up to (99.999%) of availability. These measurements are computed by taking into account the total time an application was down or unavailable, subtracting any scheduled downtime (for maintenance, upgrades, etc.), subtracting this value from the entire time the system could perform as expected (conduct useful work), and dividing that calculation by the total time the system could do useful work [16]. In order enhance their availability percentage, IT organizations invest considerable effort in improving uptime by reducing downtime, this usually being their main goal.

Figure 2.2: Monitoring a system's availability around the world

In batch processing operations, which have played the main form of computing activities in 1960s and 1970s, SNMP technology was definitely considered the ideal technology. The reason behind this statement is the need of status information and control management of the suite of activities scheduled in batch processing systems.

With the next generation of the interactive 2-tier Client-Server architecture, the concept of *user experience* (also known as *UX*) was introduced in computer science literature. This new computing architecture viewpoint, takes into account the response time values of requests and responses as they directly might negatively influence the productivity of end-users [2]. As becoming complex, these systems were attached additional messaging and transaction middleware to ensure divide load partitioning across multiple resources. In this context, Simple Network Management Protocol was also very useful to monitor and prevent serviceability and scalability vulnerabilities. Without having such a system in place, the IT organization would have been in charge of identifying possible services' availability issues affecting the overall business.

Further on, a brand new systems management system was developed in parallel with the extensive application of SNMP. Namely, one remarkable effort was FCAPS (fault, configuration, accounting, performance, and security) management, with high applicability in telecommunications industry. This standard was replaced by Information Technology Infrastructure Library (ITIL), defined as the generally accepted model for IT service management, *performance* being considered a designated discipline.
The general idea spread throughout Sydor's book is that improving the overall user experience should start at an early phase in any software project lifecycle. More precisely, by adopting a continuous monitoring strategy, preventive and corrective measures may be rapidly taken in order to remove performance bottlenecks. In this settlement, APM, considered a bridge between FCAPS and ITIL, represents a challenging and rewarding initiative, as it successfully addresses these performance concerns [3].

2.1.3. Understanding APM

The old rigid approach to APM was lately replaced by a robust, user-friendly one, becoming no longer available only for development-operation and system-administrator teams (as known as devops and sysadmins), but also useful for development, testing, operations and business teams involved in different stages of a software application's lifecycle.

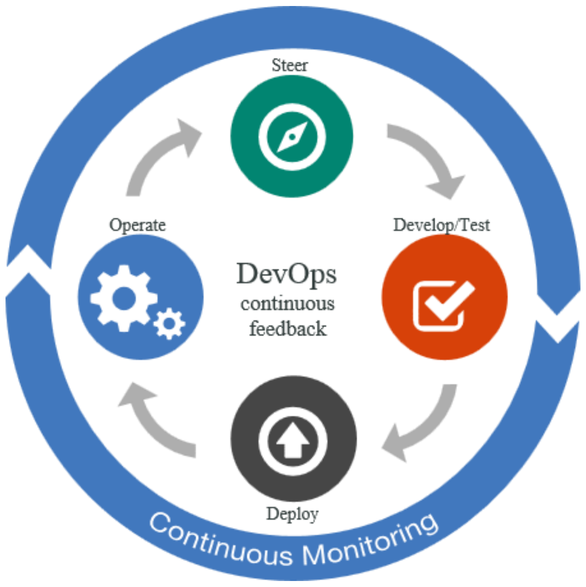


Figure 2.3: Continuous Application Performance Monitoring

A common approach is for developers to continuously assess quality features of their implemented work through code-level troubleshooting tools and monitor the entire project status through generated reports. Furthermore, testers use to take advantage of APM-specific software in order to enhance the testing accuracy, more rapidly spot performance vulnerabilities and perform load testing on the monitored web application's features. Similarly, synthetic intended to ensure high quality user experience by detecting performance problems at an earlier stage in the software applications' lifecycle, are conducted by operation teams (also known as ops) [1]. Even business leaders take advantage of APM's principles and features to carry web transactions in an adequate manner while maintaining and even increasing the revenues by dismissing possible performance threats and vulnerabilities [5].

Summing up all these various perspectives, it might be stated that APM would definitely offer significant benefits to all actors involved in a software application's lifecycle. Consequently, a crystal clear overview of the system under construction will be available to developers, testers, operations and business specialists. This way, possible flaws and performance issues will be detected before a certain software product reaches the release stage.

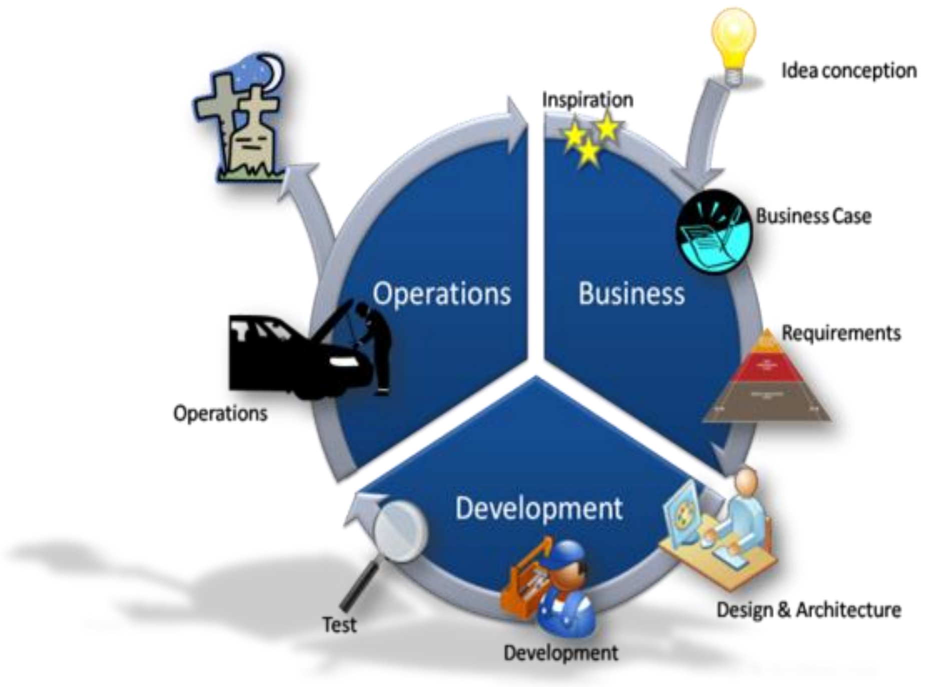


Figure 2.4: APM in software application's lifecycle

Taken from another viewpoint, using APM inside a project ensures information consistency and sharing, thus reducing additional effort invested in distribution and collaboration activities. This way, the overall development effort and time are considerably decreased, while increasing customers' satisfaction.

However, in providing guidelines and recommendations concerning the benefits offered by the usage of performance motoring when developing a software product, careful attention has to be paid to the differences among applications, their particular goals. These aspects are essential points that have to be taken into consideration when assessing the enhancements offered by an in-place APM system. Even if the final goal of APM is to maintain and improve the number of active customers through an optimal user experience and level of satisfaction, some systems benefit more than others from an extensive APM. Keeping the customer content while taking advantage of the services provided by a web application is becoming more difficult due to the complexity and diversity of alternatives products available on the market.

Furthermore, APM might be perceived as a bottleneck detector, offering precise information regarding the system's point of failure. A system's performance might be affected by various elements such as external APIs, web and application servers, cloud-hosting platforms, database workbenches, mobile carriers, and alternative internet browser engines. As a consequence, an unresponsive page of the currently surfing web application will prevent the customer to take advantage of the desired services and therefore, determine him/her to leave the application in favor of an alternative one with a solid serviceability rate [6]. This scenario is undesired by every company as it directly affects its business and therefore, impressive efforts are invested in order to avoid this situation.

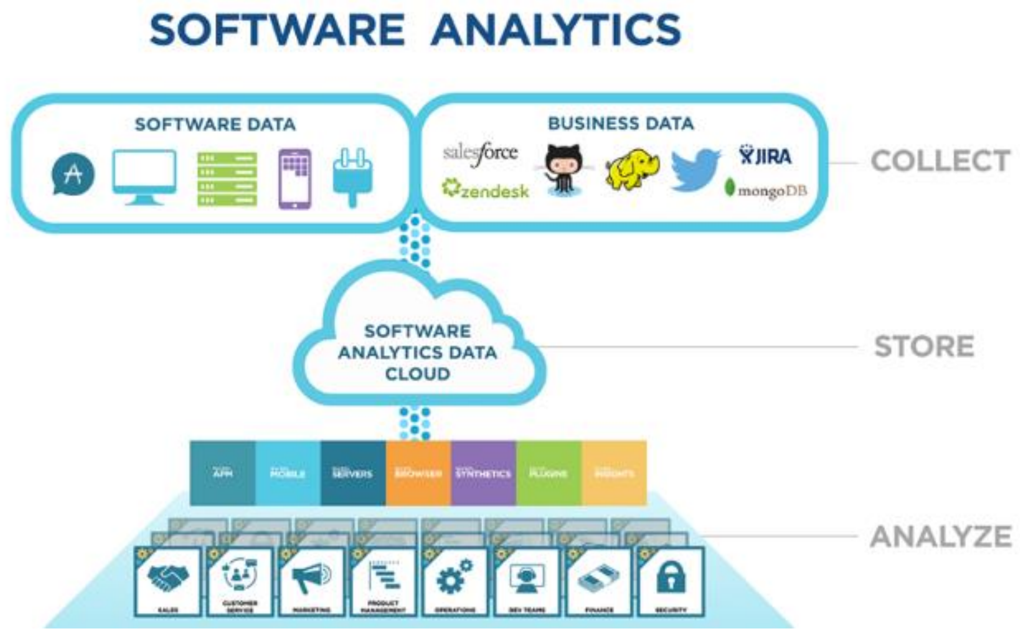


Figure 2.5: APM Overview

However, not all potential bottlenecks can be prevented and threats may affect even a highly stable and solid system. Consequently, each business should be prepared to deal with any problem and vulnerability that will probably come into the worst moment. The most agnostic aspect involved in such a situation is the lack of knowledge regarding the problem's root cause. By having in place the right performance monitoring tools, issues like these can be rapidly identified and their roots determined, thus allowing to take corrective measures as soon as possible. An easy-to-use and interactive APM system may reveal specific points of failures and accurate problem-oriented solutions will be implemented at the burning point [7]. Even more, a consistent root-cause analysis strategy might drill down to the specific source code line where the vulnerability gets its start.

In addition, through means of load-testing and real-user monitoring tools, business providers have the opportunity to self-experience same problems faced by end users around the globe. There are dedicated APM tools which have the capacity to emulate and monitor different users' mobile devices and browsers, revealing information regarding the response time a certain page gets rendered [15]. Such monitoring functionalities are among the most useful as they allow applications' owners to prioritize quality optimization efforts. More precisely, in order to offer the required level of content and satisfaction to the end user, specific performance boosting methods may be implemented in favor of other metrics and bugs that are of smaller importance for the business.

2.1.4. APM Advantages

Among all tools used throughout the lifecycle of a software product, application performance monitoring (APM) ones may have the most impact on an organization's bottom line. The reason behind this statement is that APM can proactively identify and address performance issues before they negatively affect the business through downgraded system quality characteristics [9]. Among the most frequent vulnerabilities faced by a web application are those that cause response time delays, impact employees' productivity and erode customer satisfaction.

Furthermore, the suite of APM software tools available generates database information containing historical performance trends records, allowing IT teams to easily prevent and avoid root causes of performance issues. A collection of the most critical benefits offered by APM are listed as follows:

- *Significant increase in sales and revenue*

In today's continuously demanding economy, gaining or losing a customer is determined by the serviceability and operability of the system in use. Having in place an application performance monitoring system can prevent and also detect a downgrade in the system performance, so that IT can rapidly handle the issue, thus offering the customer the required services [1]. The availability as well as quality characteristics of software applications have a significant impact on the business performance and directly affect its brand image and revenue.

Figure 2.6: Sales and revenue increase

Furthermore, APM offers troubleshooting information and support to corporations and reduce the amount of time a system is unresponsive, this way reducing the revenue risk associated with poor performance [14]. Also, APM helps in maintaining the customers satisfied and content through an enhanced user experience.

- *Reduced downtime, thus providing business continuity*

Being at the heart of any business, IT services are of mandatory availability, unscheduled downtimes causing severe loss for the associated business. In this context, APM plays an essential role in maintaining and improving availability of the system through continuous monitoring of the core services and ensure that the expected level of operability is preserved. Furthermore, APM is responsible to improve the troubleshooting's efficiency and promote safer upgrades and changes. When complemented by the powerful ability of IT Operations Analytics solutions, APM provides actionable insights allowing IT operations teams to predict and prevent downtime, as it represents the major threat for the business due to lost productivity and revenue as well as damaged reputation [17].

].

Figure 2.7: Reduce downtime

Consequently, the greatest advantages of APM are enhancing the availability and quality of services provided to customers and reducing and even removing the risks of service disruptions which can adversely affect overall business.

- *Improve User Experience (UX) and Satisfaction*

One of the most important advantage of APM is satisfying the end-users' needs. In case of poor quality and contention when taking advantage of the services provided, they will not offer this feedback to the business proprietary, but resume using it in favor of an alternative one. This is the process through which an IT corporation might lose trust, customers, and revenues [2]. This is a reality. APM will keep you one step ahead of the game i.e. keeping your end-users using.

Figure 2.8: Improve user experience and satisfaction

Here is where APM comes into play. These virtualization-aware management systems able to rapidly diagnose problems and offer up-to-date information concerning the root causes, help enterprises in apply corrective measures at the burning point, so that to remove the bottleneck. APM's primary goal is to battle the threats and vulnerabilities caused by increased complexity and offer accurate performance overview to businesses, helping to assure faster response times, increased productivity and, most importantly, satisfied end customers.

- *Higher productivity*

Application monitoring tools provide enterprises a complete overview of their environment, by offering information about possible bottlenecks before they affect the system's operations [13]. By taking advantage of this data on the web application' status, IT leaders are given the opportunity to design and implement corrective solutions to overcome the reported threats facing the system.

Figure 2.9: Higher productivity

Moreover, studies show that organizations that support an effective APM system, obtain significant improvements in productivity of IT staff as the large amount of time invested in taking decisions and steps towards fixing a critical bottleneck that affected the application’s behavior was considerably reduced [10]. Therefore, the IT personnel is able to focus more on supporting business users and performing other specific activities.

Apart from preventing the system’s failure, APM support corporations to unify IT performance with business effectiveness, thus focusing on the outcome revenues.

- *Enhance innovation*

Another essential benefit of effective APM is an increase in innovation. As a result, when applications are well managed, development teams are refrained from critical situations, thus being allowed to invest more time on delivering new features and functionalities to support key business decisions more rapidly.

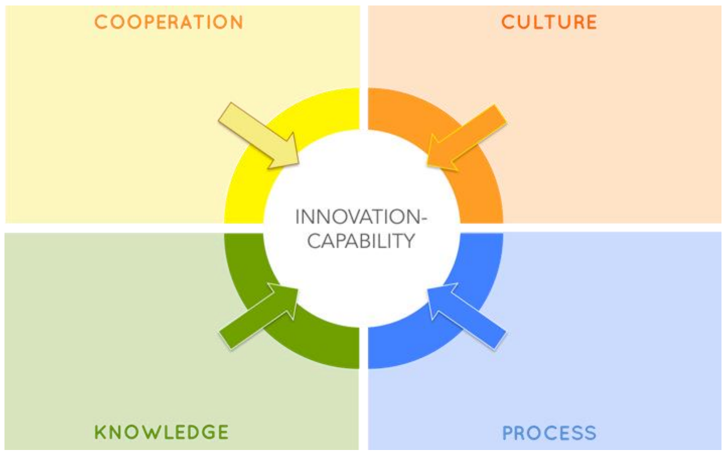


Figure 2.10: Enhance innovation

Furthermore, corporations that intend to transition from agile development to agile delivery not only need a change in culture, but also need to make sure to focus on application performance as it impacts end user experience. This will define whether a software becomes successful or not. By defining whether a system is successful or not in providing the desired products and services to the end customer, APM features greater automation in order to cope with the speed of change, and it fosters greater collaboration between operations (abbreviated as *ops*), testing, business analysts and development teams [12]. This will increase confidence for agile extension movements by continuously control of the system’s operation in order to completely fulfill customers’ needs.

- *Reduce operational costs*

One key benefit of APM is to reduce IT costs and free that savings for investment in new products and services to ensure higher position on the market. Through APM, an enterprise is enabled to fine tune the business so that to both minimize the impact of poor performance, by instant detection of vulnerabilities, and encounter hidden efficiencies that would bring overall cost of service down.

Figure 2.11: Reduce operational costs and increase profits

Taken from another point of view, application performance monitoring is constantly focused on problem detection and allows rapid intervention so that to maintain the customer content. This way, it will determine IT to significantly increase business efficiency by decreasing the number of IT staff members needed to triage a problem and lower operational costs by reducing trouble ticket escalations by 40-50% [11].

2.1.5. Application Performance Monitoring vs. Management

While in literature and even industry, APM has several different meanings, the adequate ones for the scope of the dissertation thesis are *Application Performance Monitoring* and *Application Performance Management*.

According to [18], Application Performance Monitoring is defined as the expectation of the suite of dedicated tools and how to implement them. Alternatively, Application Performance Management is the aspiration of what APM field should become. As it covers other disciplines, Application Performance Management is considered suitable to become itself an IT discipline after its associated concepts and notions will be integrated in the IT literature and industry [20].

However, the End-User Experience (also referred as EUE) is at the heart of it both meanings of APM, and has become the central target of every business.

No matter the term used to describe the acronym APM, it represents an innovative movement, creating a new setup for the business to work with IT metrics [19]. In this movement, both *Application Performance Monitoring* and *Application Performance Management* play important roles by removing threats and criticality from the IT world and helping business develop gracefully [8].

2.2. State of the Art

2.2.1. APM Development

Due to the extension of the Internet, client-server computing, with additionally attached middleware systems, come into great pressure. Highly distributed applications started to become desirable to the more monolithic ones in terms of the expense of software maintenance and the pace at which changes (in features and functionality) were required. With the growing complexity of distributed applications, the SNMP (Simple Network Management Protocol) technology was considered unsuitable as it has major shortcomings.

Throughout the computer evolution process, in the 1980s, the number of small computers exploded with the acceptance of the Personal Computer (PC) by businesses. Moreover, the CPU speed increased and the cost of system memory and disk storage considerably decreased, thus allowing distribution of applications from the mainframe onto the new generation of mini-, and eventually microcomputers [16].

IT organizations were confident about the monitoring infrastructure and started to distribute the idea that software engineering was disrupting application availability. From that period on, software engineering became much more disciplined as a result, while the investment in monitoring started to decrease. The improvement of software engineering were further on extended to middleware and built the foundation for message brokers, integration middleware and application servers around the 1990s.

Figure 2.8: IT development

Furthermore, system's complexity was continuously enhancing as with the extensibility of distributed application engineering: the Internet, Java/.NET technologies, the open source frameworks, as well as web services. However, the development of system's complexity and distribution, determined the need of change in the area of traditional monitoring. The previously used tools become inaccessible to the growing community of application stakeholders and were not able to successfully handle all application performance issues. Consequently, IT corporations concluded that information was not enough to identify and resolve the user experience requirements in the Internet age of distributed computing.

As a result, the monitoring organization started to loose terrain and the new millennium was intended to improve the next generation of monitoring tools, more specialized and offering brand new features aiming to overcome the limitations of traditional SNMP monitoring. Meanwhile, the IT environment began a significant re-sizing in terms of both human and material resources investment [13]. Moreover, business started to participate more active in the IT's landscape, taking the responsibility for possible performance issues. From this period on, an impressive attention was given to transactions and user experience because they play significant roles in business development.

2.2.2. APM Tools available on the market

Today's continuously developing market offers a wide variety of Application Performance Monitoring tools, even freely available to business companies that intend to prevent any possible threat through monitoring. Tens of APM software products aim to attract customers by offering various functionalities at different levels of success, thus differentiating from the concurrent alternatives. Apart from alerting IT managers about the emergency of threats, APM tools help to prevent problems from occurring in the first place by detecting early warning signs that might cause critical problems to the business.

As applications have become much more difficult to manage because of their continuously improving complexity, traditional APM has proven unable to handle them anymore. Therefore, new monitoring software products have emerged which are capable to evaluate all kinds of complex systems, both web and mobile apps [13].

This sub-chapter follows with a comparison among several APM tools that are highly used by IT corporations worldwide:

- *JMeter vs LoadRunner*

JMeter and LoadRunner are two different performance testing tools. Performance testing tools are products which assess various performance characteristics of software applications. Moreover, a system might be monitored through such performance testing software applications that function by increasing the load on them and evaluating the maximum limit up to which they can work in an efficient and effective manner.

- ***<u>JMeter</u>***

JMeter represents an application performance testing – oriented monitoring tool, used to test and assess the supported load Client-Server software applications. As far as its origins concern, JMeter is a Java tool, developed by Apache Software Foundation, Jakarta, or Apache JMeter for short. Furthermore, JMeter operates as an open-source software intended to measure the performance and test the functional behavior of different scaled systems. Even if it started by evaluating web applications, JMeter presently provides a suite of functionalities and features able to test and analyze the performance of software applications.

Figure 2.9: Jmeter icon

Regarding its compatibility, JMeter is able to execute tests on different platforms, both static and dynamic, as Java objects, FTP servers, files, servlets, SOAP, databases and queries and many more, thus easily fit into any particular system.

- ***<u>LoadRunner</u>***

LoadRunner is an automated interactive tool which used to test the performance of a software application. Concerning its vendor, LoadRunner is developed by Mercury Interactive to offer support in identifying the behavior of server and network applications under load normal, stress, and extended testing. However, in November, Hewlett-Packard become the owner of LoadRunner performance testing tool.

Figure 2.10: LoadRunner icon

As far as its supportability concerns, LoadRunner applies to various platforms, application environments, databases, while being an extensive tool which can identify most of the bugs. Moreover, it gathers performance information for both system and component levels by using solid diagnosis and monitoring modules.

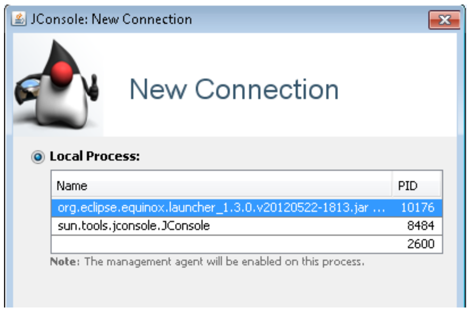
As a comparison among these two testing - oriented monitoring tools, JMeter might be chosen because of its free usage and an unlimited load generation, while LoadRunner has a limited load generation capacity even if purchased at an expensive price. However, LoadRunner’s price pays off because of its highly complexity and suite of extended developed features [11]. Furthermore, JMeter exhibits a simple, easy-to-use user interface, as compared to the impressive, either sophisticated one provided by LoadRunner.

- *JConsole vs VisualVM vs AppDynamics Lite*

The main challenge faced by IT Operations is to must monitor the application run-time, as they currently lack visibility of how systems actually interconnect and operate. In the context of Java applications, the first step towards this direction is to start using free monitoring tools such as JConsole and VisualVM. They are both available with the SDK (Software Development Kit) and offer efficient features for monitoring Java applications.

- ***<u>JConsole</u>***

JConsole offers basic monitoring of the Java Virtual Machine (JVM) run-time by assessing key characteristics such as CPU, memory and threads, and shows associated information under different tabs that summarize the health of JVM memory, threads, classes, virtual machine and MBeans. Being very easy to use, JConsole starts recording information from the moment it gets connected to the JVM. The main strengths of JConsole are high potential in evaluating the mechanics of a JVM at a high level in order to determine potential vulnerabilities such as memory leaks, high thread concurrency and deadlock.



Name	PID
org.eclipse.equinox.launcher_1.3.0.v20120522-1813.jar ...	10176
sun.tools.jconsole.JConsole	8484
	2600

Note: The management agent will be enabled on this process.

Figure 2.11: JConsole opening view

Another feature of JConsole is represented by the MBean viewer which provides accurate reports of a large variety of KPI metrics in real-time in order to check which JVM resources are being exhausted by the application. Concerning its usage, the UI is both simple and easy to navigate but lacks application context with respect to the user requests and transactions that execute within the JVM [11]. One drawback associated to JConsole is related to the overhead it can introduce to an application, and therefore Oracle recommends it being deployed only in development and test environments.

- ***<u>VisualVM</u>***

Being also freely distributable with the Java SDK and very similar with JConsole, VisualVM takes application monitoring one level deeper by providing analysis of thread execution as well as the ability to profile CPU and memory usage of JVM requests, both of which are triggered manually by the user. Furthermore, VisualVM also has a dedicated tab for illustrating memory pool and garbage collection activity intended to determine abnormal trends. Taking into account its features, VisualVM is considered suitable for developers who possess deep application knowledge and are able to piece together VisualVM's reports with the Java components associated to the executed requests. Otherwise, as it provides little context of application requests or user transactions, VisualVM might not be helpful for IT Operations personnel.

Figure 2.12: VisualVM icon

Unfortunately, it exhibits the same shortcoming as JConsole regarding the overhead that it might cause in a production environment when features like CPU and memory profiler are in execution. However, VisualVM remains a helpful tool for development and testing, being widely used as it offers a suite of functionalities at no price.

- ***AppDynamics Lite***

AppDynamics Lite represents another free tool that takes a slightly different approach to application monitoring. Starting at higher level and visualizing the monitored JVM together with its infrastructure interactions, it offers users the opportunity to see the top business transactions that flow through the JVM, creating a level of abstraction from just analyzing raw threads and JVM metrics. The idea behind its operability, is that AppDynamics Lite exhibits a top down approach to monitoring as opposed to bottom up, and provides IT Operations and developers the chance to overview the real business impact before they drill down and analyze data.

Figure 2.13: AppDynamics Lite icon

Therefore, the most useful scenario is that users start from a business transaction reported as being slow and follow its path until getting into the root source code to identify the exact class, method and interface which was responsible for the bottleneck. Consequently, instead of analyzing metrics holistically from different perspectives, the IT specialist may troubleshoot using a business transaction as an anchor all the way down to the root cause and operate on the burning point. Similar to the previously presented tools, Lite supports real-time MBean functionalities as well as alerting capabilities enabling users to monitor application performance in a preventive fashion. The most important advantage of AppDynamics Lite is that it can run in production 24/7 without causing overhead and therefore allowing issues to be rapidly fixed [13].

As a comparative overview among these three application performance monitoring tools, common features might be easily spotted, the essential differences come from the way monitoring, data acquisition and presentation is accomplished. When nominating the most appropriate tool for application monitoring, the important aspect is the user's needs from the outcome [14]. More precisely, developers find helpful a bottom up view with lots of data to analyze, while IT Operations are satisfied with a top down approach so they can troubleshoot and maintain context without getting lost in meaningless details.

Moreover, additional overhead is another concern that has to be taken into account when selecting a performance monitoring software as it varies among tools. However, any of the previously presented solutions offer cheap and easy way to get started with application monitoring, which provides insights about possible vulnerabilities and allows immediate intake of corrective measures.

Chapter 3. Technological Considerations

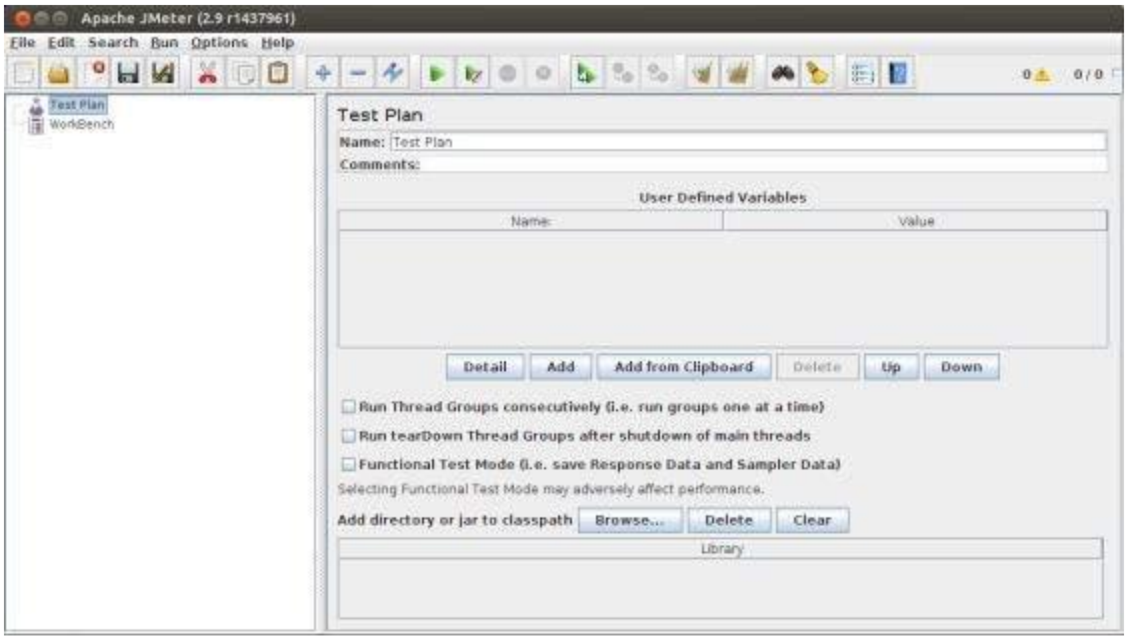
3.1. Tools used on behalf of the Research and Experiments conducted

Among the large variety of Application Performance Monitoring (APM) and performance testing tools available on the market, *JMeter* and *VisualVM* ones were used throughout the investigation and research phases of the dissertation thesis, as well as in conducting experimental activities intended to validate the implemented performance improvements. The following subsections outline relevant information related to these tools and their fundamental features.

3.1.1. Apache JMeter

Being an Open Source testing software, Apache JMeter works as a 100% pure Java application for load and performance testing of web applications. Basically, JMeter supports several categories of tests like load, functional, performance, regression, and many other ones, and requires the installed JDK's version to be minimum 5.

Concerning its Graphical User Interface (GUI), JMeter exhibits an intuitive and easy-to-use UI by taking advantage of Swing's graphical API. Consequently, it can be launched on any environment / workstation that accepts a Java Virtual Machine (JVM).



Name: Test Plan
Comments:

☐ Run Thread Groups consecutively (i.e. run groups one at a time)
☐ Run tearDown Thread Groups after shutdown of main threads
☐ Functional Test Mode (i.e. save Response Data and Sampler Data)
Selecting Functional Test Mode may adversely affect performance.
Add directory or jar to classpath

Figure 3.1: JMeter GUI

Apache JMeter is mostly used to test both functional behavior and performance characteristics of software applications on either static or dynamic resources. Research shows an extensive usage of the JMeter tool in simulating heavy loads on servers, networks or certain objects in order to test their strength and analyze the overall application's performance features under different load types [21]. This way, accurate information regarding quality metrics and different system capabilities might be obtained and, afterwards, performance analysis will be conducted and possible vulnerabilities impacting the application's operability spotted.

The basic idea behind its functionality, is that JMeter works by simulating a group of concurrent users sending requests to the target server, and afterwards returns statistical data indicating the performance of the software application under test via a suite of visuals (tables, graphs, charts, diagrams, and many other graphical constructs). The main steps involved in JMeter operation are depicted in Figure 3.2 below.

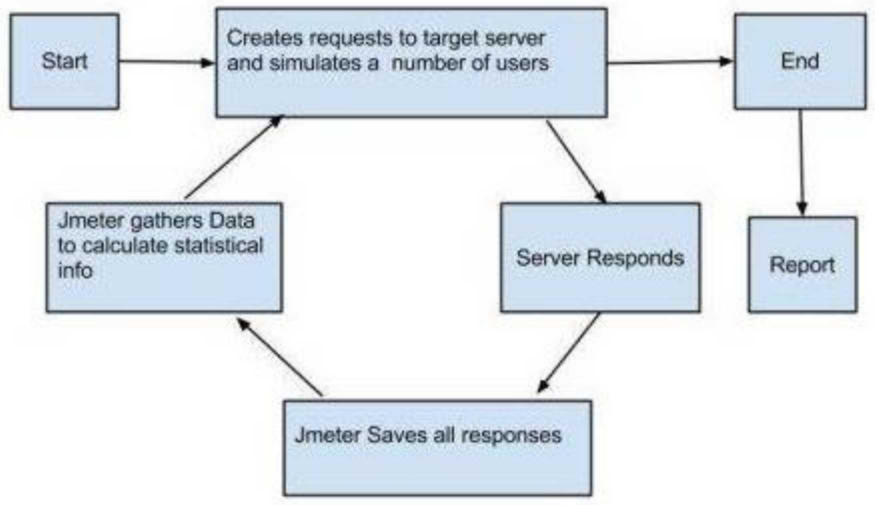

Start
Creates requests to target server and simulates a number of users
End
Jmeter gathers Data to calculate statistical info
Server Responds
Report
Jmeter Saves all responses

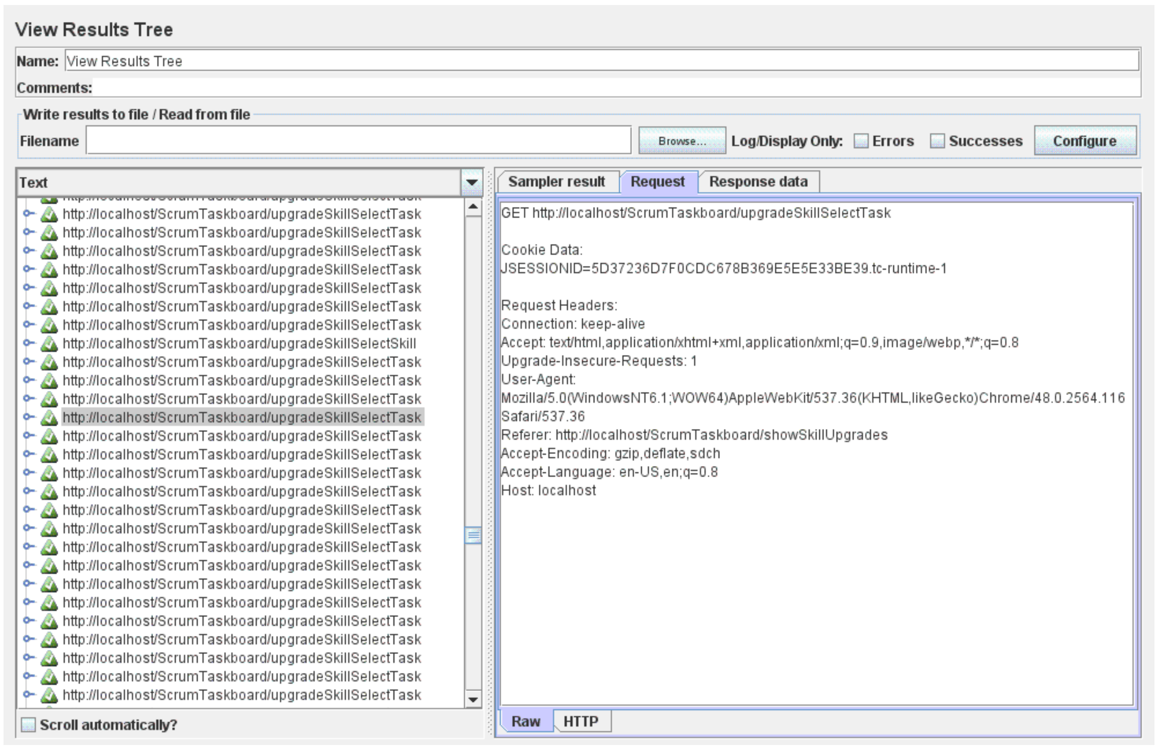
Figure 3.2: JMeter operation

After successful installation of the JMeter, its features and functionalities might be practiced by using the tool in either GUI or non-GUI mode. According to [22], the main difference among these operating modes relates to the load executed on the system under which the JMeter tests are run. Even if it is more intuitive to use and offers a higher user experience, the GUI mode causes additional overhead to the system's performance, especially when many visual elements are added to the test.

Despite of the extra charge added to the system under test, the most preferred JMeter approach is the GUI one, especially for novice users. At any test starting point is the creation of a test plan, the container for running tests which defines what and how to test the desired functionality. The suite of all visual elements involved in a JMeter test, as well as configuration settings, are added to the test plan: thread groups, logic controllers, sample-generating controllers, listeners, timers, assertions, and configuration elements [21].

A summary of the main JMeter features available are listed as follows:

- High portability and extensibility, being able to load and test performance of different servers and services like HTTP, HTTPS, FTP databases, SMTP, TCP, SOAP, REST and many others.
- By having a simple and intuitive GUI, it is capable to generate different types of reports as well as a suite of visual representations of the measured system's properties.
- Offers a full multi-threading framework that allows concurrent sampling by many threads and simultaneous sampling of different functions by separate thread groups.
- Provides personalization through data visualization and evaluation plugins.
- Offers several reporting possibilities with different specializations, based on the listener elements added to the testplan as output elements. Therefore, a large variety of listeners might be included in a certain testplan depending on the main purpose of the performance monitoring as well as the subsequent investigations to be conducted. The most commonly used listener is the *View Results Tree* one responsible to display request and response information. The *View Results Tree* test plan component is mainly chose in executing functional tests. Figure 3.3 shows a snapshot of JMeter's *View Results Tree* listener.
- In case more advanced graphical and statistical data is required, several other listener components are available in the JMeter's toolbox. More precisely, a large variety of performance metrics and statistical information such as the number of samples used, throughput, error, standard deviation are reported from the testplan's execution. Moreover, this data might be written to different formatted files (plain text, csv or xml).



```
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectSkill
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
http://localhost/ScrumTaskboard/upgradeSkillSelectTask
```

```
GET http://localhost/ScrumTaskboard/upgradeSkillSelectTask

Cookie Data:
JSESSIONID=5D37236D7F0CDC678B369E5E5E33BE39.tc-runtime-1

Request Headers:
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent:
Mozilla/5.0(WindowsNT6.1;WOW64)AppleWebKit/537.36(KHTML,likeGecko)Chrome/48.0.2564.116
Safari/537.36
Referer: http://localhost/ScrumTaskboard/showSkillUpgrades
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Host: localhost
```

Figure 3.3: Snapshot of *View Results Tree* listener

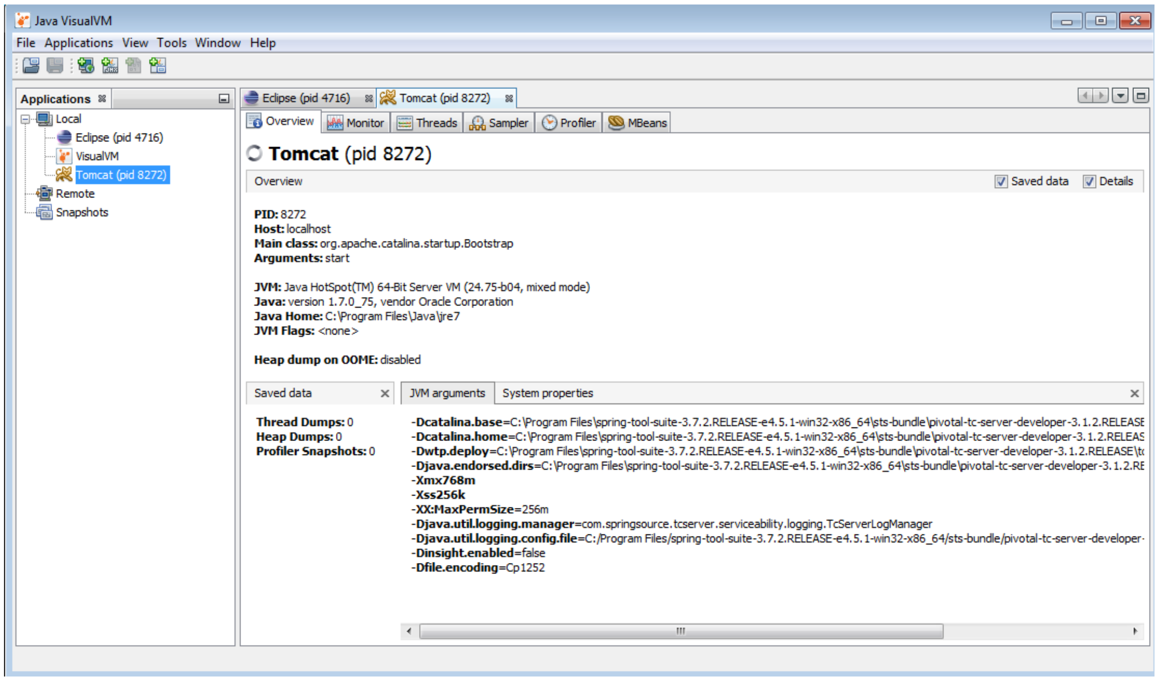
3.1.2. VisualVM

VisualVM represents a tool used to monitor Java technology-based applications (Java applications) while they are running on a Java Virtual Machine (JVM). As an application performance monitoring tool which freely comes when installing Oracle/Sun JDK's of minimum version 6, VisualVM is still able to monitor applications running on JDK 1.4 and higher. Its main responsibility is to offer a suite of monitoring options, among which the profiling and sampling ones are of interest for our research. The profiler and sampler tools accessible from the VisualVM's IDE are available for both CPU and memory load as performance indicators.

VisualVM exhibits a multitude of out-of-the-box functionalities that would definitely meet various software applications necessities available to developers, test engineers, system administrators (also known as sysadmins) and even end-users submitting bug reports.

Regarding its Graphical User Interface, VisualVM interface is divided into two parts, the Applications window situated in the left side and the Main window extended in the right and center sections of the entire tool window. The Applications window outlines a tree structure listing the applications running on local and remotely connected hosts [23]. The monitoring activities begin after selecting the target node from the Applications window.

Following the monitored application selection is the opening of the Main window displaying detailed information under six tabs: Overview, Monitor, Threads, Sampler, Profiler and MBeans. A snapshot of the VisualVM's entrance GUI is outlined in Figure 3.4.



```
Tomcat (pid 8272)

Overview                                              [✓] Saved data  [✓] Details

PID: 8272
Host: localhost
Main class: org.apache.catalina.startup.Bootstrap
Arguments: start

JVM: Java HotSpot(TM) 64-Bit Server VM (24.75-b04, mixed mode)
Java: version 1.7.0_75, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jre7
JVM Flags: <none>

Heap dump on OOME: disabled

Saved data                 JVM arguments   System properties

Thread Dumps: 0            -Dcatalina.base=C:\Program Files\spring-tool-suite-3.7.2.RELEASE-e4.5.1-win32-x86_64\sts-bundle\pivotal-tc-server-developer-3.1.2.RELEASE
Heap Dumps: 0              -Dcatalina.home=C:\Program Files\spring-tool-suite-3.7.2.RELEASE-e4.5.1-win32-x86_64\sts-bundle\pivotal-tc-server-developer-3.1.2.RELEAS
Profiler Snapshots: 0      -Dwtp.deploy=C:\Program Files\spring-tool-suite-3.7.2.RELEASE-e4.5.1-win32-x86_64\sts-bundle\pivotal-tc-server-developer-3.1.2.RELEASE\t
                           -Djava.endorsed.dirs=C:\Program Files\spring-tool-suite-3.7.2.RELEASE-e4.5.1-win32-x86_64\sts-bundle\pivotal-tc-server-developer-3.1.2.RE
                           -Xmx768m
                           -Xss256k
                           -XX:MaxPermSize=256m
                           -Djava.util.logging.manager=com.springsource.tcserver.serviceability.logging.TcServerLogManager
                           -Djava.util.logging.config.file=C:/Program Files/spring-tool-suite-3.7.2.RELEASE-e4.5.1-win32-x86_64/sts-bundle/pivotal-tc-server-developer-
                           -Dinsight.enabled=false
                           -Dfile.encoding=Cp1252
```

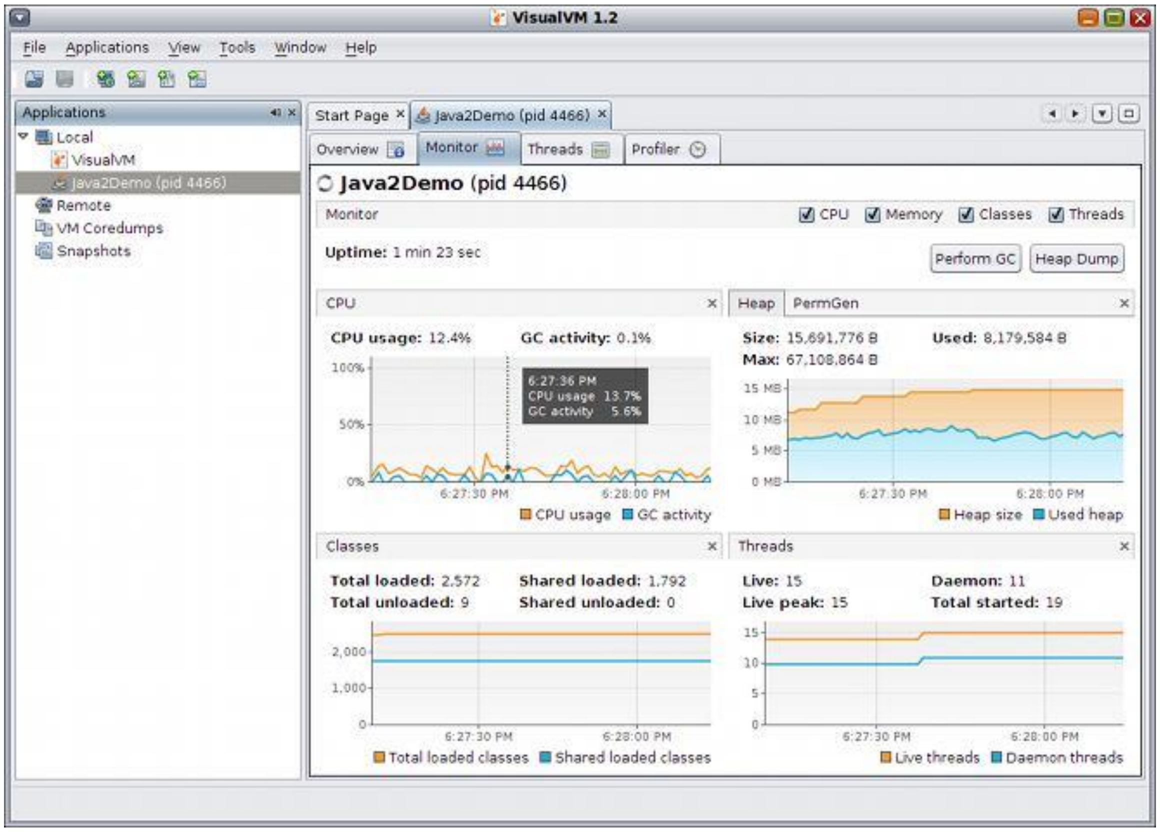
Figure 3.4: VisualVM's GUI

Each of these tabs contains embedded information specific to the owned responsibility, allowing the technical user to explore and analyze various performance measurements as well as additional functionalities.

An overview of the main features provided by VisualVM are displayed in the following:

- Allows monitoring of both local and remote Java applications. Additionally, the user might manually create a JMX connection to other applications [24]. As a result, it allows checking the Java applications running on the system as well as the existence of a remote J2EE server process alive.

- Offers monitoring capabilities of an application's performance as well as its memory consumption. More precisely, VisualVM supervises various characteristics associated to the target application, such as CPU usage, Garbage Collection (GC), heap and permanent generation (PermGen) memory, number of loaded classes and currently running threads. This way, strange behavior which may cause vulnerabilities will be detected and corrective actions undertaken in order to avoid the emergent threat. Figure 3.5 outlines the application performance and memory consumption monitoring view.



Java2Demo (pid 4466)

Uptime: 1 min 23 sec

CPU usage: 12.4% GC activity: 0.1%

6:27:36 PM
CPU usage 13.7%
GC activity 5.6%

Size: 15,691,776 B Used: 8,179,584 B
Max: 67,108,864 B

Total loaded: 2,572 Shared loaded: 1,792
Total unloaded: 9 Shared unloaded: 0

Live: 15 Daemon: 11
Live peak: 15 Total started: 19

Figure 3.5: VisualVM – monitor application performance and memory consumption

Among the most commonly identified flaws are memory leaks, cases in which the garbage collection might be manually triggered. Furthermore, a heap dump might be collected which makes application heap contents available to explore.

- All application threads (threads running on behalf of a Java process) are supervised and listed in a timelined table, as presented in Figure 3.6. This allows monitoring thread activity and identifying inefficient blocked or unused thread workers.

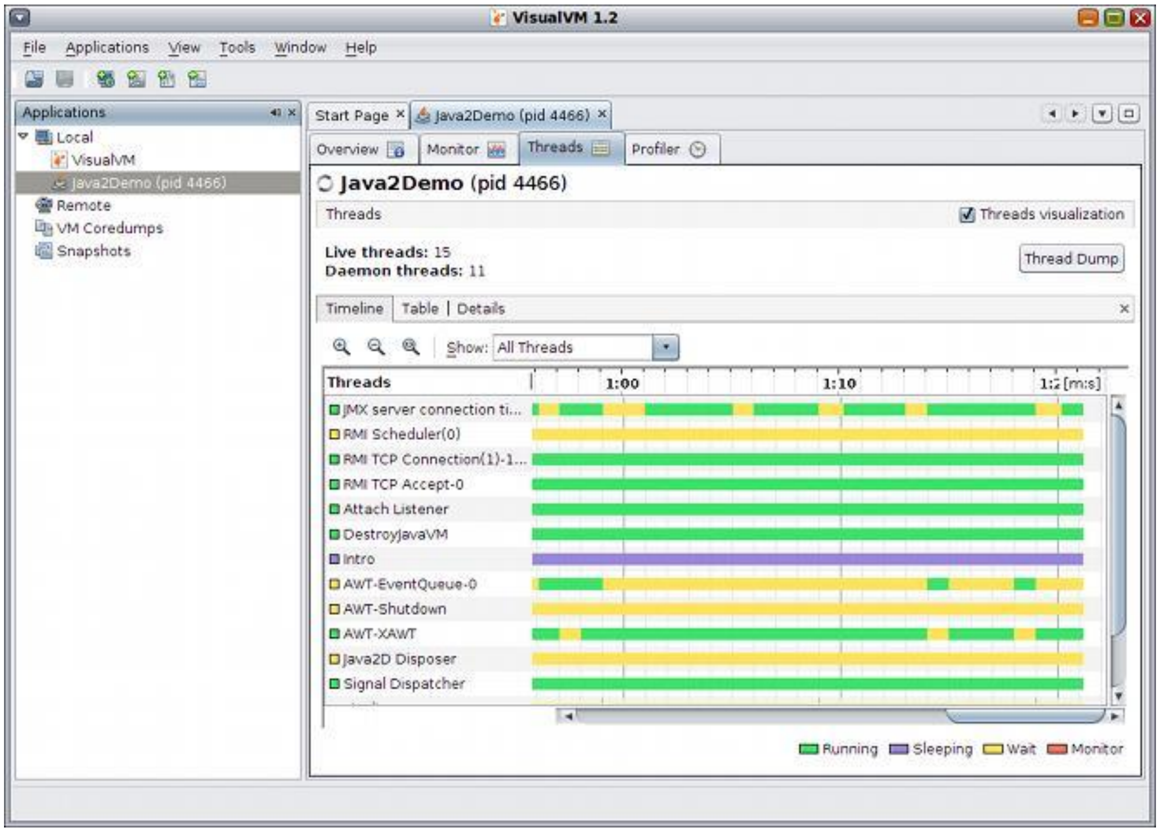
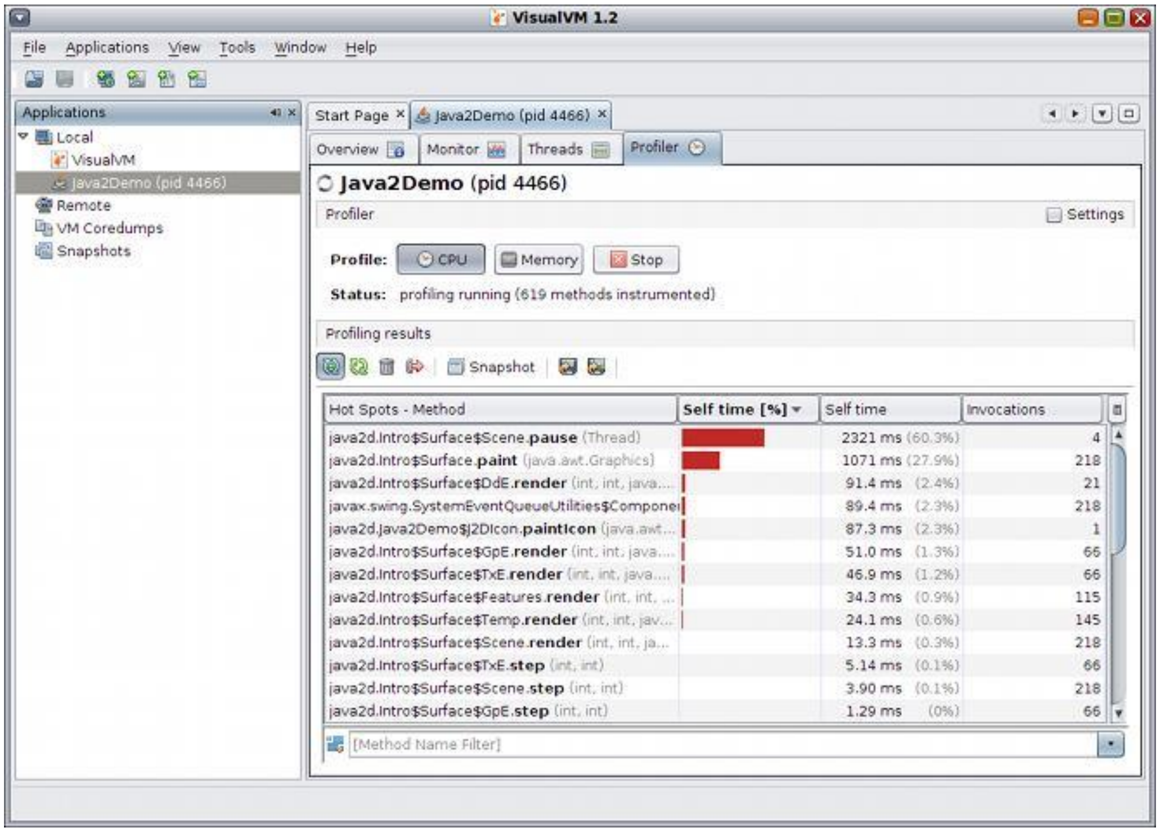


Figure 3.6: VisualVM – monitor application threads

- Profiling of application performance in order to offer analysis information regarding memory allocation. More precisely, VisualVM contains an embedded profiler which provides relevant data by just a mouse click: detecting the place where most of the time is being spent or the objects that consume most of the memory. The VisualVM's profiler interface is illustrated in Figure 3.7.
- Very intuitive record and visualization of thread dumps. Furthermore, simultaneous thread dumps of multiple running applications might be recorded at once to start detect possible distributed deadlocks.

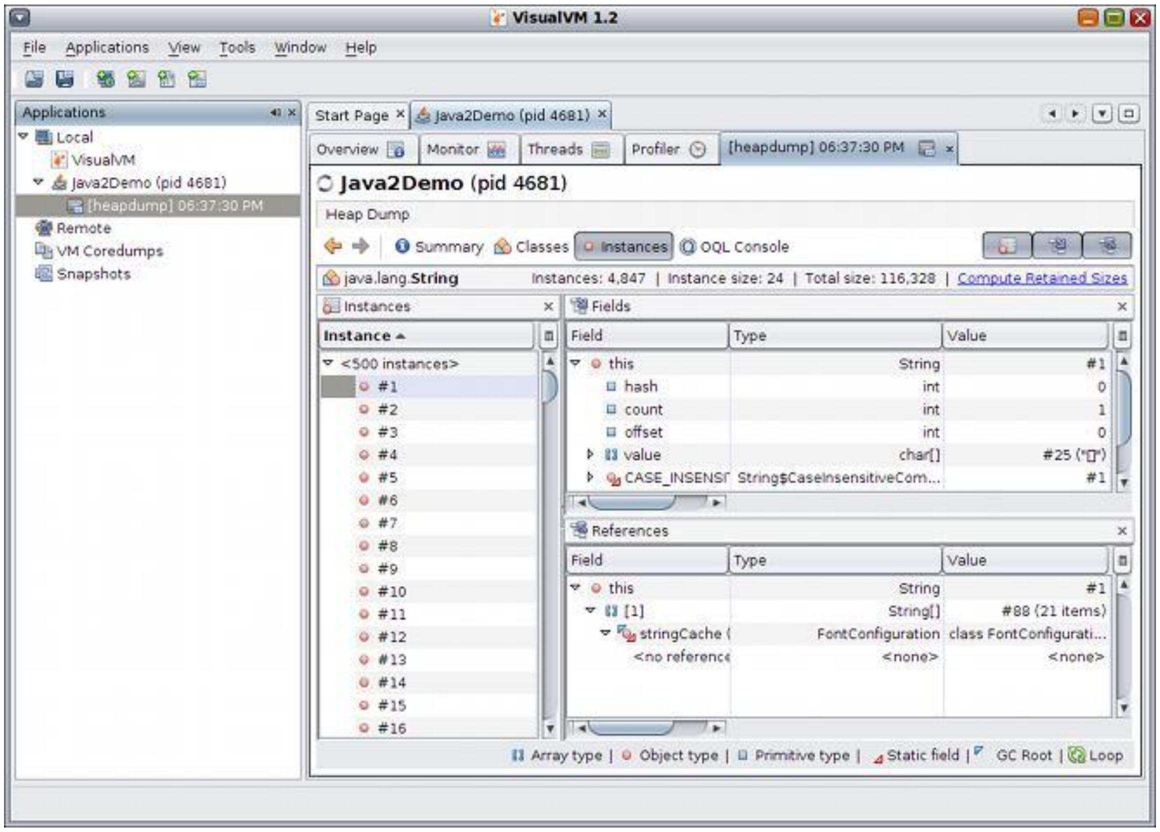


Status: profiling running (619 methods instrumented)

Hot Spots - Method	Self time [%]	Self time	Invocations
java2d.Intro$Surface$Scene.**pause** (Thread)		2321 ms (60.3%)	4
java2d.Intro$Surface.**paint** (java.awt.Graphics)		1071 ms (27.9%)	218
java2d.Intro$Surface$DdE.**render** (int, int, java...		91.4 ms (2.4%)	21
javax.swing.SystemEventQueueUtilities$Compone		89.4 ms (2.3%)	218
java2d.Java2Demo$J2DIcon.**paintIcon** (java.awt...		87.3 ms (2.3%)	1
java2d.Intro$Surface$GpE.**render** (int, int, java....		51.0 ms (1.3%)	66
java2d.Intro$Surface$TxE.**render** (int, int, java....		46.9 ms (1.2%)	66
java2d.Intro$Surface$Features.**render** (int, int, ...		34.3 ms (0.9%)	115
java2d.Intro$Surface$Temp.**render** (int, int, jav...		24.1 ms (0.6%)	145
java2d.Intro$Surface$Scene.**render** (int, int, ja...		13.3 ms (0.3%)	218
java2d.Intro$Surface$TxE.**step** (int, int)		5.14 ms (0.1%)	66
java2d.Intro$Surface$Scene.**step** (int, int)		3.90 ms (0.1%)	218
java2d.Intro$Surface$GpE.**step** (int, int)		1.29 ms (0%)	66

Figure 3.7: VisualVM – Profiler

- Recording and browsing heap dumps in order to analyze the application memory's contents or investigate possible memory leaks, VisualVM provides an embedded *HeapWalker* tool. Moreover, this tool is able to interpret files having the *hprof* format and to outline heap dumps created by the JVM on occurrence of an *OutOfMemoryException*. Figure 3.8 outlines heap dumps features available in VisualVM.



java.lang.String Instances: 4,847 | Instance size: 24 | Total size: 116,328 | Compute Retained Sizes

Instance
<500 instances>
#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
#11
#12
#13
#14
#15
#16

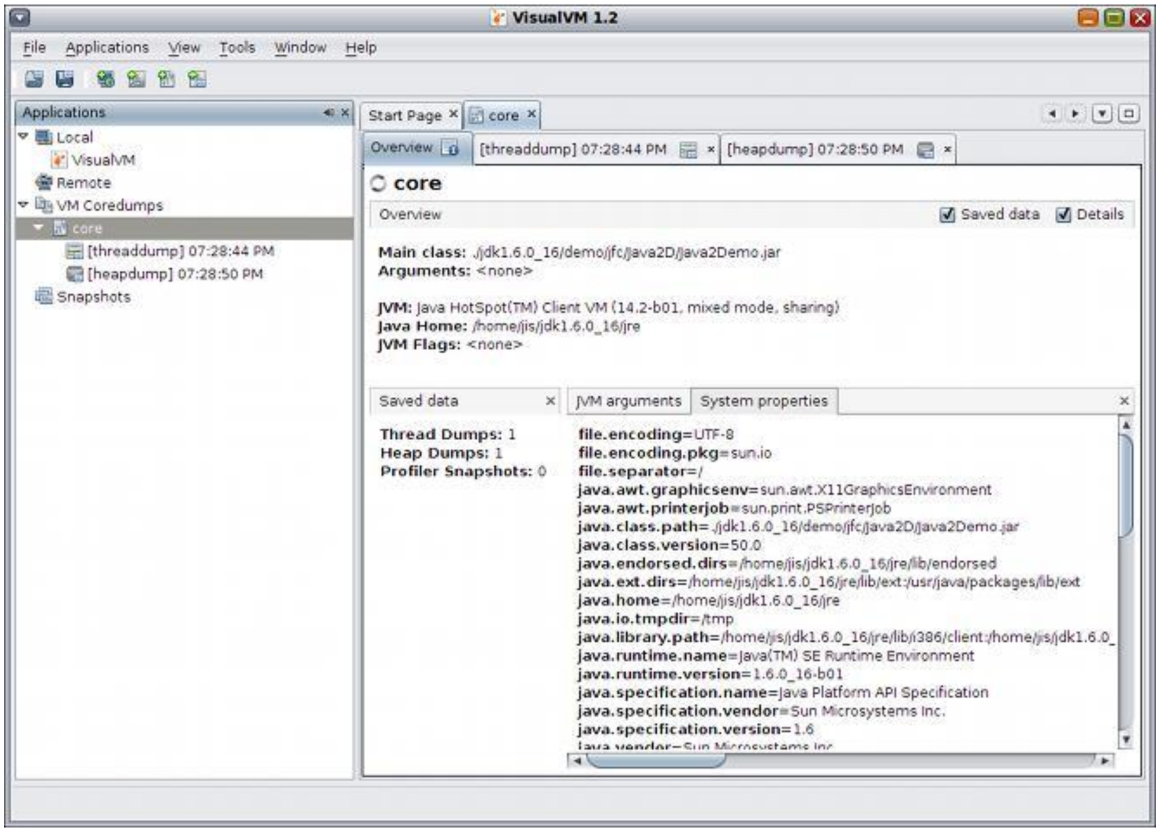
Field	Type	Value
this	String	#1
hash	int	0
count	int	1
offset	int	0
value	char[]	#25 ("□")
CASE_INSENSI	String$CaseInsensitiveCom...	#1

Field	Type	Value
this	String	#1
[1]	String[]	#88 (21 items)
stringCache (	FontConfiguration	class FontConfigurati...
<no reference	<none>	<none>

Figure 3.8: VisualVM – Record and visualize heap dumps

- Analysis of core dumps, very useful when a Java process suddenly stops working without any clue about the failure reason being available [23]. Specifically, in unfortunate cases when a failure occurs, a core dump may be generated by the JVM containing essential information regarding application status at that moment. Afterwards, VisualVM might be extremely useful as it offers an overview of both the application configuration and runtime environment, as illustrated in Figure 3.9. Additionally, VisualVM is able to extract thread and heap dumps from the initial core dump.

- Evaluation of applications offline by saving both application configuration and runtime environment together with the recorded thread dumps, heap dumps and profiler snapshots into a single application snapshot that might be further on analyzed offline. This feature is considered useful for bug reports in the context when users provide a single file containing the necessary information to investigate runtime environment and application state.



```
Main class: ./jdk1.6.0_16/demo/jfc/Java2D/Java2Demo.jar
Arguments: <none>

JVM: Java HotSpot(TM) Client VM (14.2-b01, mixed mode, sharing)
Java Home: /home/jis/jdk1.6.0_16/jre
JVM Flags: <none>

Thread Dumps: 1
Heap Dumps: 1
Profiler Snapshots: 0

file.encoding=UTF-8
file.encoding.pkg=sun.io
file.separator=/
java.awt.graphicsenv=sun.awt.X11GraphicsEnvironment
java.awt.printerjob=sun.print.PSPrinterJob
java.class.path=./jdk1.6.0_16/demo/jfc/Java2D/Java2Demo.jar
java.class.version=50.0
java.endorsed.dirs=/home/jis/jdk1.6.0_16/jre/lib/endorsed
java.ext.dirs=/home/jis/jdk1.6.0_16/jre/lib/ext:/usr/java/packages/lib/ext
java.home=/home/jis/jdk1.6.0_16/jre
java.io.tmpdir=/tmp
java.library.path=/home/jis/jdk1.6.0_16/jre/lib/i386/client:/home/jis/jdk1.6.0_
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.6.0_16-b01
java.specification.name=Java Platform API Specification
java.specification.vendor=Sun Microsystems Inc.
java.specification.version=1.6
java.vendor=Sun Microsystems Inc.
```

Figure 3.9: VisualVM – Evaluate core dump

3.2. Technologies used in License Thesis development

The license project, representing the starting point and fundamental basis for the current dissertation thesis research, was developed in Java programming language using Spring Web MVC as development framework and JPA for data persistence. Therefore, a brief overview of the main technologies used for the project's implementation, as well as their key characteristics are presented in this sub-chapter.

3.2.1. Spring Framework

Spring represents an open source application framework and a Java platform which provides infrastructure support for building Java applications. As Spring does not impose any programming model, it might be used by any Java application. Moreover, the framework provides extensions which allow developing web applications on top of Java EE platform, thus becoming a preferred alternative to the Enterprise Java Bean (EJB) model.

Before Spring release, enterprise applications were usually developed using the J2EE standards which allow the programmer to deploy an application on any J2EE application server, on any platform. Running an application on an application server provides several benefits and services, such as transaction management, messaging, mailing, a directory interface and other ones defined in the J2EE specifications. Unfortunately, as the usage of these standards was very complex, several problems arise. For instance, developing an Enterprise Java Bean required the programmer to write a set of xml files as deployment descriptors, home interfaces, remote/local interfaces, etc. Secondly, there was a 'look-up' issue concerning the dependencies among components. Therefore, when a component requires another component, the component itself was responsible for looking up the components it depends upon, this process happening by name, so the name of the dependency was hardcoded in the component. In this case, the communication between components from J2EE application servers of different vendors always represented a problem. Finally, in many situations, there were components that did not require all the services provided by the application server but, without any alternative, all the application's components were heavy weight, supporting all features even when didn't need them. Although the functionality came out of the box, the developers need to configure all the components, thus writing tons of xml configuration for unused functionalities.

All the above limitations of J2EE applications and many other determined programmers to prefer just Plain Old Java Objects (POJOs), thus removing the J2EE overhead. POJOs provide easier business logic, removing the need to implement any interfaces or extend other classes, thus being able to design and develop the domain classes using Java regular classes. However, this approach has one drawback: the programmer cannot take advantage of the services provided by the container such as transaction management, remoting, etc.

The Spring Framework is the one that removes this limitation, by providing a lightweight container for the POJOs to live in. As opposed to a heavyweight container, the components are now as light as possible, without the need to support unrequired services and their corresponding configuration. Moreover, no programming model is employed, so the developer might use POJOs and declaratively specify the needed services. By using POJOs to develop enterprise-class applications, Spring programmers do not need an EJB container product such as an application server, but only a robust servlet container such as Tomcat or similar commercial product.

Furthermore, Spring allows developers to focus only on the components they need and ignore the rest, thus being organized in a modular fashion. It uses the existing technologies like several ORM and logging frameworks, JEE, Quartz and JDK timers, as well as other view technologies. A definite enhancement brought by Spring is represented by its web extension, a well-designed web MVC framework, which provides a great alternative to other existing web frameworks such as Struts [25].

Spring is mainly characterized by its IoC (Inversion of Control) containers which tend to be lightweight as compared to EJB ones, allowing to develop and deploy applications on computers with limited memory and CPU resources. As an application development framework, Spring provides a consistent transaction management interface that may support a local transaction (using a single database, for instance) or a global transaction (using Java Transaction API, for instance).

Moreover, Spring focuses on exception handling, thus providing a consistent API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO) into consistent, unchecked exceptions. Finally, testing a Spring application is easy as environment-dependent code is moved into the framework. Also, using JavaBean-style POJOs, allows the programmers to use dependency injection for injecting test data.

3.2.1.1.Spring Framework architecture

The Spring framework has a layered architecture which consists of twenty modules that are built on the top of its core container. These modules provide everything needed in an application development, enabling the programmer to select the required features and remove the redundant ones [26]. Moreover, Spring's modular architecture assures a smooth integration with other frameworks. Figure 3.10 shows the division of the Spring Framework architecture among these modules.

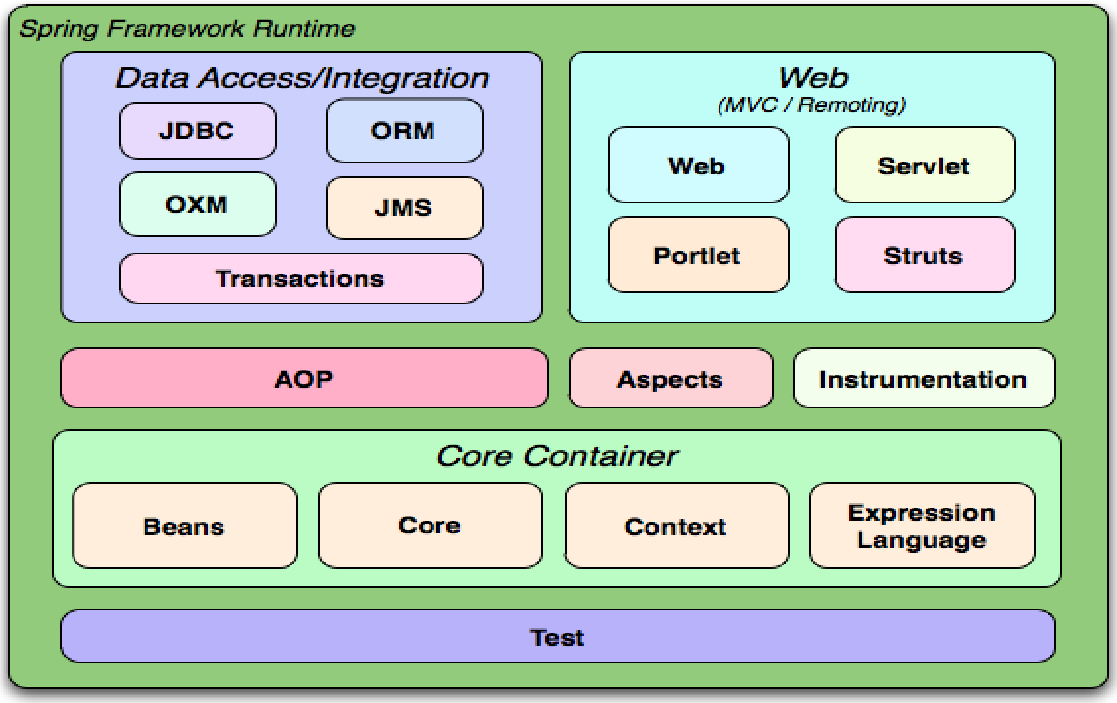


Figure 3.10: Spring Framework architecture

As this figure briefly shows, the Core Container represents one component of the Spring architecture. It consists of the Core, Beans, Context, and Expression Language described below:

- The Core module provides the fundamental components of the framework, including the Inversion of Control and Dependency Injection features.

- The Beans module provides a complex implementation of the factory design pattern, named BeanFactory.

- The Context module is built on the Core and Beans modules and represents a way to access the objects defined and configured. The main point of this module is represented by the ApplicationContext interface.

- The Expression Language module is intended to offer a consistent expression language for managing and querying objects at runtime.

Another layer of the Spring Framework architecture is the Data Access/Integration one which contains the JDBC, ORM, OXM, JMS and Transaction modules presented below:

- The JDBC module is intended to offer an abstraction layer which eliminates the need to write tiresome JDBC related code.

- The ORM module offers integration for commonly used object-relational mapping APIs such as JPA, JDO, Hibernate and iBatis.

- The OXM module contains an abstraction layer which offers Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

- The Java Messaging Service JMS module provides functionality for producing and consuming messages.

- The Transaction module offers programmatic and declarative transaction management for both POJO classes and those that implement special interfaces.

Furthermore, **t**he Web layer has as modules the Web, Web-Servlet, Web-Struts, and Web-Portlet ones whose description is given as follows:

- The Web module is intended to offer basic web-oriented integration functionalities such as multipart file-uploading as well as the initialization of the Inversion of Control container using a web-oriented application context and Servlet listeners.

- The Web-Servlet module is of high interest for the majority web applications because it contains the Spring's model-view-controller (MVC) implementation.

-The Web-Struts module allows the integration of a classic Struts web tier within a Spring application through its support classes.

-The Web-Portlet module enables the use of the MVC implementation in a portlet environment.

Moreover, there are few other important modules like AOP, Aspects, Instrumentation, and Test modules which bring crucial enhancements to Spring applications:

- The AOP module, through its aspect-oriented programming implementation, enables developers to define method-interceptors and pointcuts in order to detach code that implements functionality that should be isolated.

- The Aspects module ensures integration with AspectJ, another essential and powerful aspect oriented programming (AOP) framework.

- The Instrumentation module allows both infrastructure for class instrumentation and class loader implementations to be used in different application servers.

- The Test module focuses on testing Spring components dedicated frameworks such as JUnit or TestNG.

3.2.1.2. Inversion of Control and Dependency Injection (DI)

Inversion of Control and Dependency Injection represents a core design pattern of Spring framework. The Inversion of Control uses reflection to provide a consistent means of configuring Java objects, thus being responsible for managing object lifecycles: creating the objects, calling their initialization methods and configuring these objects by wiring them together. The container’s configuration might be performed in XML files or by detecting specific Java annotations on configuration classes. Objects can be obtained by means of dependency injection, a pattern in which the Inversion of Control container transmits Java objects by name to other objects, through constructors, properties, or even factory methods [25].

When developing a Java application, classes should be as independent as possible of other in order to provide reuse and perform independent unit testing. However, inside a complex Java application, there are dependencies between components, services, classes etc. Without Inversion of Control, the programmer would have to wire these together in order to build the dependencies [27]. This approach is not convenient because it forces the developer to change the code every time new artifacts or unit tests have to be created. Therefore, the solution to this problem is the Inversion of Control container which acts as an external party to manage all the dependencies. As its name briefly suggests, in the Inversion of Control pattern, the programmer allows someone from the outside to control how the application dependencies are wired together. In this context, the Dependency Injection, considered as the heart of the Spring Framework, helps in gluing these classes together and same time keeping them independent, as shown in Figure 3.11.

Dependency Injection might be seen as a pattern used to create instances of objects that other objects rely on, without the need to know at compile time which class will be used to provide that functionality. Inversion of Control relies on Dependency Injection because it acts as a mechanism which activates the components in order to provide the specific functionality. More precisely, Dependency Injection might occur by passing parameters to the constructor or by post-construction using setter methods. Moreover, by using Spring's main concepts, the dependencies of a particular object are not satisfied at compile time but at runtime. This way, the developer is not obliged to provide all the dependencies at compile time.

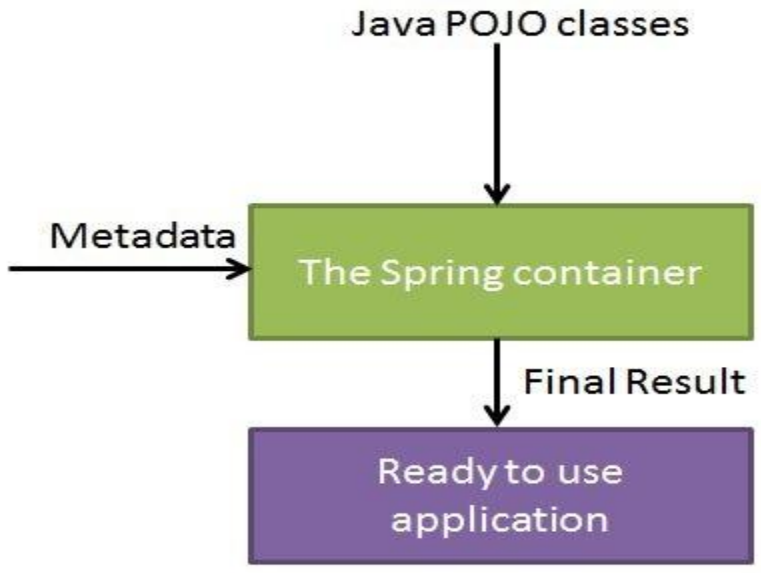


Figure 3.11: Dependency Injection (DI)

3.2.1.3. Aspect Oriented Programming (AOP)

One of the major features offered by the Spring Framework is the provision for separating the **cross-cutting concerns** in an application through the means of **Aspect Oriented Programming.**

In the case of most applications there are concerns that cut across different abstraction layers, logging being the typical example. For instance, the programmer might want to log in every method of the service layer, thus having a trace when the program enters and exits that specific method. Instead of writing log statements all over the application's service layer, the developer has the opportunity to use Aspect Oriented Programming (AOP) which considers logging as one concern separated from the business logic into a different entity [29].

In the AOP terminology a concern is similar to an advice, an entity like a class, which can be applied to certain pointcuts in the code. A pointcut represents a place in the code where the developer wants the crosscutting concern to be applied, for instance when entering or exiting a certain method [28]. An advice together with a pointcut is called an aspect, which is the unit of modularity in AOP, thus giving the name Aspect-Oriented Programming. There are various common examples of aspects including logging, declarative transactions, security, and caching, as illustrated in Figure 3.12.

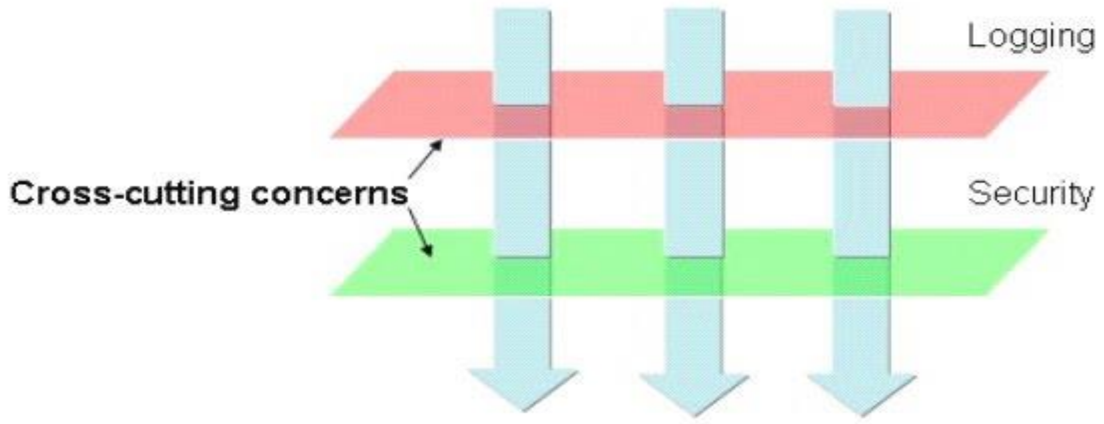


Figure 3.12: Aspect Oriented Programming (AOP)

As previously said, Dependency Injection provides decoupling among the application objects. On the other hand, AOP helps in separating cross-cutting concerns from the objects that they affect. Consequently, Spring's AOP module provides aspect-oriented programming implementation enabling programmers to specify method-interceptors and pointcuts in order to decouple code that implements functionality that should be separated.

3.2.1.4. Spring Web MVC

As the application was implemented using the Spring Web MVC Framework, a special attention has to be given to this topic. Therefore, the main actors of the Spring Web MVC architecture are the Model, the View and the Controller whose interconnection is shown in Figure 3.13. By using the Spring Web MVC Framework, the developers may take advantage of its architecture and build flexible and loosely coupled web applications. Therefore, different aspects of the application such as the input, the business logic and the user interface are separated, while a loose connection between these elements exists [28].

The main components of the Spring Web MVC Framework are described as follows:

- The Model is usually represented by Plain Old Java Objects (POJOs), thus comprising the application specific data.

- The View is responsible for displaying the application data, outputting HTML code which might be interpreted by the client's browser.
- The Controller is encharged with user requests management, appropriate creation and transmission of the model to the view for rendering.

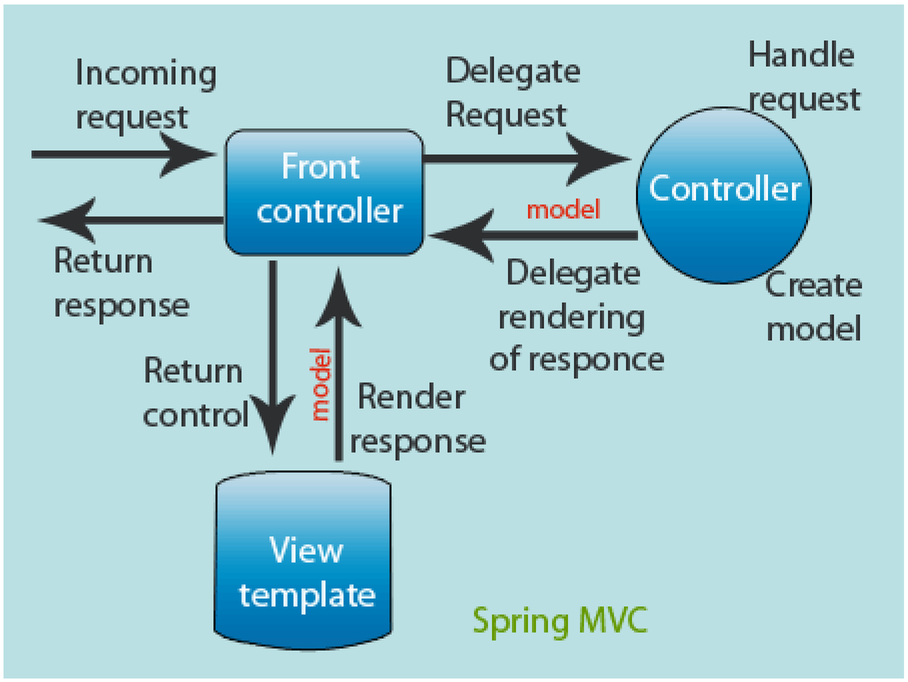


Figure 3.13: Spring Web MVC architecture

A noteworthy component of the Spring Web MVC Framework is represented by the DispatcherServlet encharged with handling all the HTTP requests and responses [30]. Figure 3.14 illustrates the request processing workflow of the Spring Web MVC.

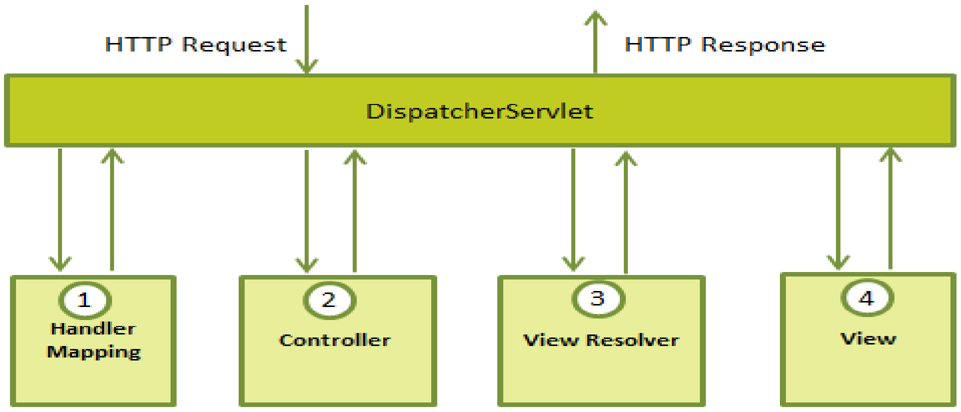


Figure 3.14: Request processing workflow of the Spring Web MVC DispatcherServlet

The main steps performed when an incoming HTTP request arrives to the DispatcherServlet are described as follows: when an HTTP request arrives, the DispatcherServlet consults the HandlerMapping in order to call the corresponding Controller [31]. That moment, the Controller receives the request and, based on the corresponding method type (GET or POST), calls the appropriate method from the service layer. Inside this service method, the model data will be set based on specified business logic and the view name is returned to the DispatcherServlet. Further on, the DispatcherServlet will use the ViewResolver to identify the defined view for the request and, when finalized, the DispatcherServlet transmits the model data to the view. The final step consists of rendering the model data to the client.

As far as the application concerns, it was structured so as to preserve the Model-View-Controller pattern. Therefore, we may identify: the Model – represented by classes from the **domain** package specific to the application domain, the View – mainly represented by the .JSP files from the **view** package and specific Spring MVC Front Controllers, and the Controller – described by the business logic implemented in the **service** and **persistence** packages.

3.2.2. Java Persistence API (JPA)

As briefly presented in this chapter, the Spring Framework has a layered architecture that provides support through its components, including the DAO (Data Access Object) one. Therefore, any of the technologies, including JDBC, Hibernate, JPA and iBATIS, can be used with the Spring Framework to access persisted data. Particularly, the application uses JPA for persistence support.

The Java Persistence API (JPA) provides Java programmers with an object relational mapping between Java objects and a relational database. Nowadays, JPA is considered the standard approach for Object to Relational Mapping (ORM) in the Java Industry.

Containing just a set of interfaces, without implementation, JPA represents only a specification, not a product, being unable to perform persistence or anything else by itself. The developers are free to choose among existing open-source and commercial JPA implementations and use any Java EE5 application server that would provide support for its use. Moreover, using JPA in an application requires a database to persist to.

More precisely, JPA allows Plain Old Java Objects (POJOs) to be easily persisted without requiring the application classes to implement any interfaces or methods. An object's object-relational mappings might be defined by using standard annotations or XML configuration, thus defining how a specific Java class maps to a relational database table. The processing of queries as well as the transaction on the objects against a database is employed with the aid of JPA's defined runtime EntityManager API. Furthermore, JPA defines an object-level query language, Java Persistence Query Language (JPQL), which allows querying of the objects from the database.

3.3. Technologies used in Dissertation Thesis development

The additional technologies used throughout the dissertation thesis' development are Apache Load Balancing and Eh-Cache with Spring and Hibernate. Consequently, a detailed overview of these technologies is presented in the following subsections.

3.3.1. Apache Load Balancing

In designing the architecture of a web application, an important aspect to consider is capacity planning which requires a decision of whether to scale the server farm vertically or horizontally [32]. As opposed to the vertical scaling which involves adding more resources to existing servers in order to fulfil changing requirements and needs, the horizontal one implies dynamic provisioning of a redundant pool of servers along with a load balancer. This last option represents the preferred approach as it helps in avoiding single points of failure in the system [33].

Basically, load balancers are mostly used to effectively distribute incoming requests to the application across a pool of servers. Nowadays intelligent mechanisms select the most appropriate server to handle individual client requests. This way, the configured load balancer fulfills the following three objectives:

- Guides users to the most suitable and best performing application server instance.
- Avoids server overloading by means of intelligent requests distribution.
- Enhances application traffic streams.

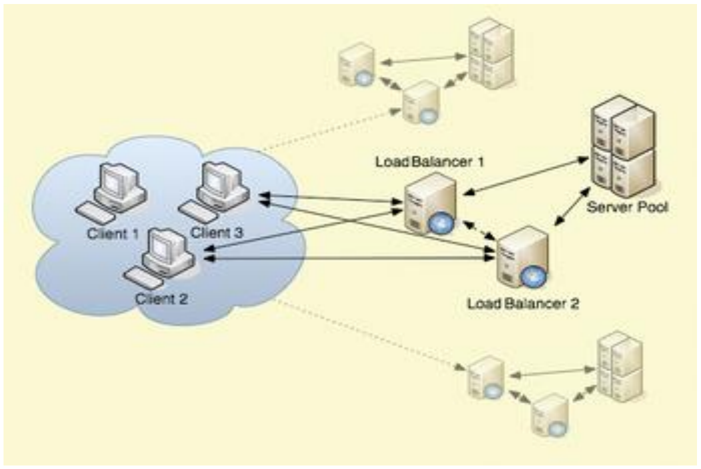
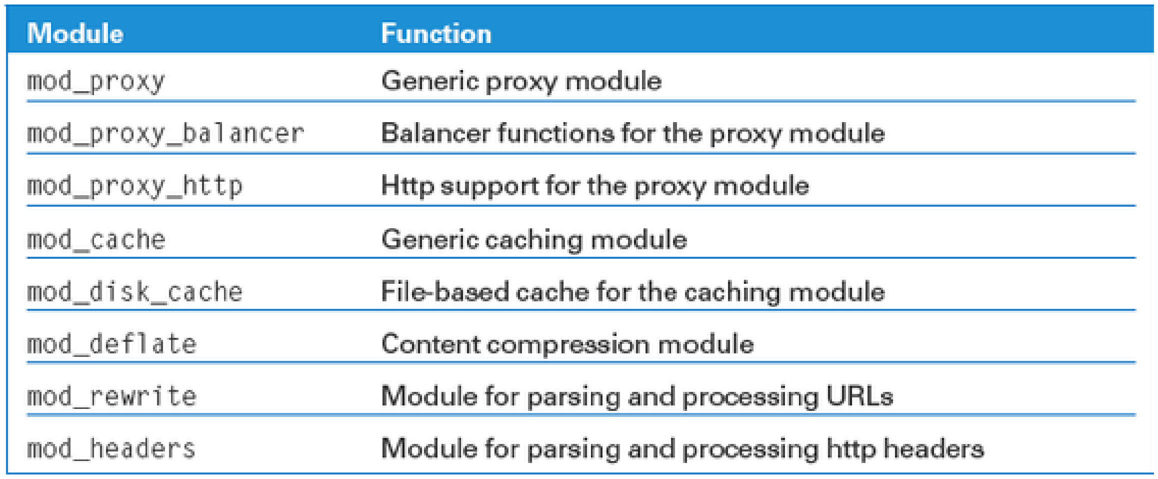


Figure 3.15: Overview of a http-based balancing system

Figure 3.15 outlines a software-based balancing system in which several front-end load balancers accept incoming client requests and distribute them to a pool of back-end servers based on some pre-established approach.

According to [34], Apache load balancer represents an open source server utilized in over 100,000 web sites to provide application traffic distribution. The most important feature, of providing high availability to web applications, it is realized through its customizability in being programmed to execute in different modes in order to meet specific environment requirements. These modes are configured using the MultiProcessing Modules (MPMs), of which many are intended to support load balancing. Figure 3.16 presents a list of Apache's basic load balancing modules, which include additional features such as caching, compression, URL rewriting, and header processing.



Module	Function
mod_proxy	Generic proxy module
mod_proxy_balancer	Balancer functions for the proxy module
mod_proxy_http	Http support for the proxy module
mod_cache	Generic caching module
mod_disk_cache	File-based cache for the caching module
mod_deflate	Content compression module
mod_rewrite	Module for parsing and processing URLs
mod_headers	Module for parsing and processing http headers

Figure 3.16: Apache modules required for load balancing

In order to use the desired Apache module, it has to be loaded (inside its specific configuration file) using the following syntax:

*LoadModule **xyz**_module modules/mod_**xyz**.so*

However, some modules might be already loaded, and, therefore, Apache's configuration file should be verified.

In case JServ-capable application servers, such as Apache Tomcat or Jetty, are used, the gateway might also use the Apache JServ Protocol (knows as *ajp*). In order to use the ajp protocol, the *mod_proxy_ajp* module will be loaded instead of *mod_proxy_http* and all URLs should be changed from http:// to ajp://. Despite of the advantages brought by this binary-format protocol concerning back-end connection performance and lower resource overheads, these enhancements are offered at the price of more permanent connections to the back-ends [35]. However, both ajp and http might be run back-ends as members of the same pool inside the same time instance.

In essence, all required configuration to set-up the cluster and customize the load balancer to fulfill the application's needs are performed in Apache's configuration files. One basic load balancing (cluster) configuration, using the Apache as balancer and a variable number of Tomcat instances as back-end servers, is illustrated in Figure 3.17.

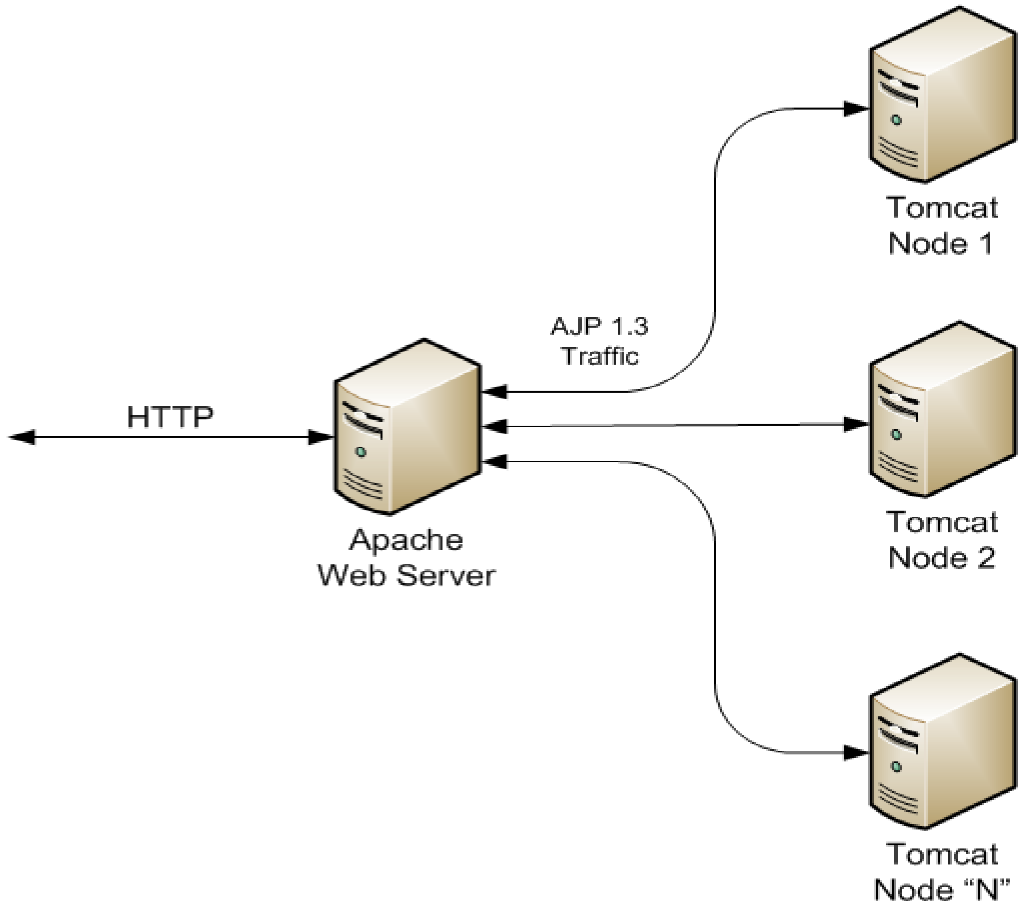


Figure 3.17: Apache load balancing mechanism

Apart from its high availability and nearly infinite flexibility, Apache http server owns extended features such as handling session management (through cookies, sticky sessions, etc.) and virtual hosts, as well as a balancer manager interface used for tracking the cluster' state [36].

As bottom line, Apache http server offers a highly efficient, elegant and scalable solution to configure a load balancing mechanism for particularized http environments. Therefore, Apache load balancing system represents a very sensible approach to popular commercial or open source alternatives.

3.3.2. Eh-Cache

Eh-Cache represents an open-source, standards-based cache for boosting performance, offloading the database, and simplifying scalability. Being a robust, proven, and full-featured solution, Eh-Cache is considered as a powerful-distributed catching framework as well as the most widely used Java-based cache. Eh-Cache might be used as a general-purpose cache or a second-level cache for Hibernate. Moreover, it supports integration with various frameworks and products such as Spring, ColdFusion and Google App Engine. Furthermore, Eh-Cache provides in-process cache that might be replicated across multiple nodes.

Currently owned by Terracotta.org, Eh-Cache resides at the core of *BigMemory Go* and *BigMemory Max*, additional Terracotta's commercial caching and in-memory data-storage products.

3.3.2.1. Spring Eh-Cache

Starting with version 3.1, Spring Framework offers support for adding caching into an existing software application developed based on this framework. Similar to the provided transaction support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the source code. Consequently, Eh-Cache integration with Spring is considered quite simple, as it consists of defining some properties in Spring's configuration XML file. Moreover, once configured, caching might be added to any method results through Spring annotations.

Figure 3.18: Spring and Eh-Cache

Implementing Spring Eh-Cache consists of the following steps:

- Add the required Maven dependencies for Eh-Cache to project's *pom.xml* file.
- Configure Eh-Cache in the Spring project's configuration file. The essential configuration consists of defining the CacheManager bean and specify its configuration location (*ehcache.xml* file) containing the definition of the particular Eh-Cache.
- Define the cache region(s) together with the properties customization in *ehcache.xml* file.
- Add *@Cacheable* annotation in Java source files to methods whose results should be cached, by indicating the name of the cache region defined in *ehcache.xml* file.

3.3.2.2. Hibernate Eh-Cache

Eh-Caching is usually implemented with Hibernate in order to offer performance optimization, as it resides between the software application and the database, in the context of avoiding the number of database hits as many as possible. This way, performance critical applications are highly optimized. Hibernate exhibits several caching schemes:

- *First-level Cache*: Represents the Session cache and a mandatory cache through which all application requests must pass. The Session object keeps an object under its own power before committing it to the database. In case multiple updates are performed for a certain object, Hibernate postpones the update as long as possible in order to reduce the number of update SQL statements executed against the database.

- *Second-level Cache*: Constitutes an optional cache that will be searched for an object occurrence only after a lookup into the first-level cache is performed. Being responsible for caching objects across sessions, the second-level cache may be configured on a per-class and per-collection basis.

- *Query-level Cache*: Defines a cache for query result sets and it easily integrates with the second-level cache. The query-level cache is perceived as an optional feature which requires two additional physical cache regions to be defined: one that holds the cached query results and another one that stores the timestamps when a table was last updated. Therefore, the query-level cache is mostly useful for queries that are executed frequently using the same parameters.

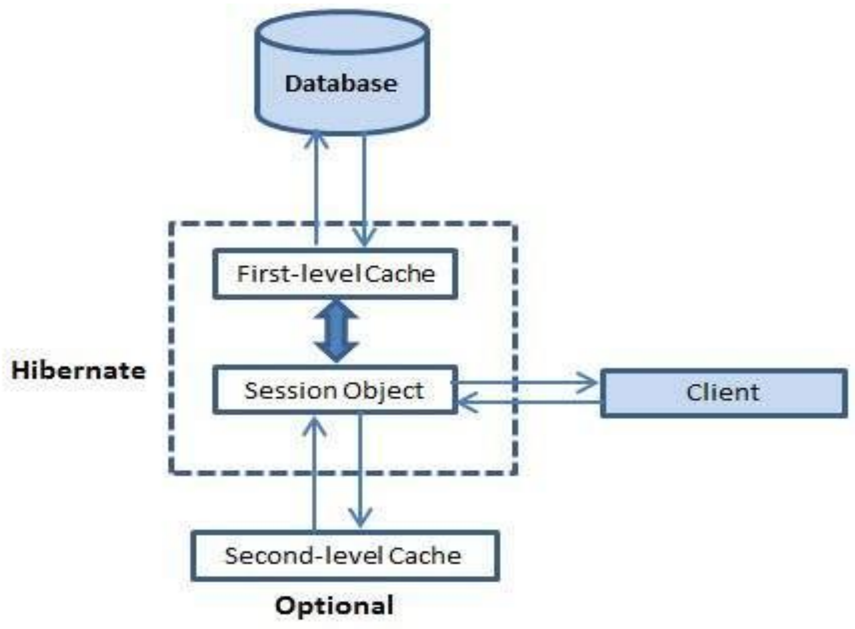


Figure 3.19: Hibernate Eh-Cache

Implementing *Second-Level Hibernate Eh-Cache* consists of the following steps:

- Add the required Maven dependencies for Hibernate-specific Eh-Cache library to project’s *pom.xml* file.
- Configure Hibernate for second-level caching and specify the second-level cache provider (in Hibernate’s configuration file).
- Eh-Cache creates by default separate cache regions for each entity configured for caching. In order to override the defaults for these regions, custom configuration has to be added to the *ehcache.xml*, placed in the project’s classpath.
- Configure entity objects to be cached by using either .xml – based definition or decorate the entity class with *@Cache* annotation. For both options, the caching strategy has to be specified as one of the following: none, read-only, read-write, nonstrict-read-write or transactional.

Moreover, enabling Query-level Hibernate Eh-Cache requires the subsequent phases:

- Activate the query-level cache by specifying its corresponding property in Hibernate’s configuration file.
- Take advantage of the query-cache in Java source code by setting the *cacheable* property for the SQL query to be executed. Moreover, the particular cache region, previously defined in *ehcache.xml* file might be set through *setCacheRegion()* method available for Java objects of type *Query*.

Chapter 4. Implementation Considerations and Experimental Results

As already mentioned in the current document, in the development of the dissertation thesis, the license thesis served as starting point. More precisely, the license thesis project was taken as initial fundament of the dissertation research, against which a suite of experiments, followed by code improvements as well as various enhancement techniques were executed. Furthermore, the obtained performance enhancements were monitored after each step using dedicated testing tools. Due to the particularity of the dissertation thesis, in which performance evaluation was systematically conducted, the analysis of experimental results was presented throughout this chapter and not in a separate one, as usual.

To begin, the initial phase in the conducted application performance optimization research, was to evaluate the state of the considered software application which was intended to be considerably improved. Therefore, the first step in the dissertation study consists of a set of JMeter experiments through which accurate information regarding performance capabilities of the monitored system were obtained. This step, as well as subsequent ones that followed in the thesis' research process are detailed in separate sub-chapters.

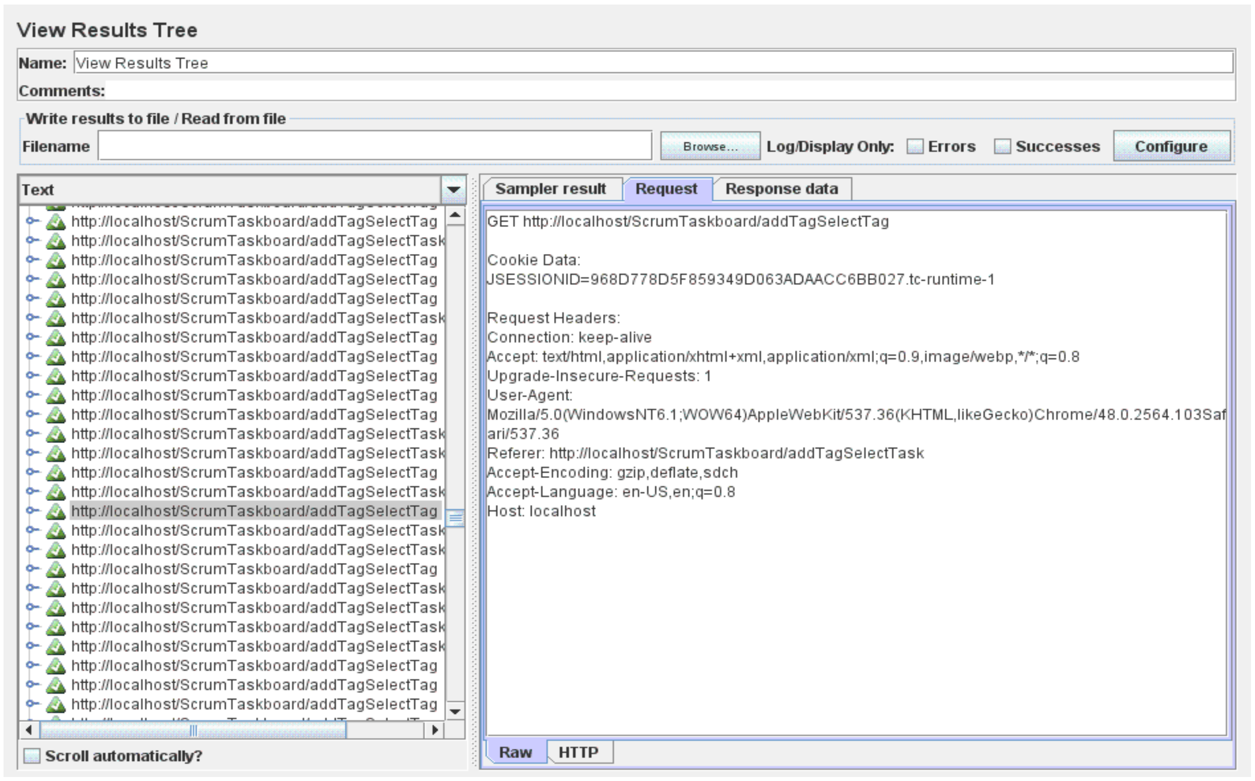
4.1. Evaluating the initial version of the dissertation thesis

JMeter tool was used to perform a series of measurements and evaluate different metrics for each use case scenario (application flow) that might be accomplished by a user of the ScrumTaskboard web application. In addition to a wide variety of monitoring utilities that are provided by JMeter, its main functionality is given by the virtual simulation of an established number of concurrent users. Therefore, several tests were conducted by suddenly changing the number of concurrent users (threads) of the web application. The results gathered from analyzing the tests' outcome were collected and further on used on subsequent phases of the dissertation thesis to serve in comparing the obtained improvements.

JMeter testplans offer a large variety of output listeners that provide a multitude of performance metrics and, therefore, allow for specific measurement of the desired system's capabilities. In order to assess the application's behavior from performance statistics point of view, a series of use cases were recorded and the corresponding testplans created. Afterwards, the following step was to add to the testplan the output listeners considered as appropriate for performance analysis viewpoint, and execute it in GUI mode. For each such use case the approach was to execute the testplan with a different number of concurrent users so that monitor the results based on this variable parameter.

When analyzing the testplans' reports for different output listeners, the overall result indicated poor performance capabilities for larger number of simultaneous users accessing the system. The main measurable attributes used as performance indicators were the reported error percentage, standard deviation, throughput, response time, number of KB/sec, latency and bandwidth.

In order to serve as sample for the previously stated facts, the main JMeter testplan listeners obtained for an initially considered use case scenario through the application are presented as follows. To be begin with, the considered use case scenario consisted of the following actions: log in as Scrum Master user, add a new task, add a tag for that task, visualize the taskboard and then log out. The corresponding testplan for this scenario was executed in the context of varying the number of simultaneous users from 100 to 300 and is being referred as **Login-Add Task-Add Tag for Task-Visualize Taskboard-Logout** throughout this paper.
The main screens obtained by executing the JMeter testcase with the configured number of 100 concurrent users correspond to the following output listeners: View Results Tree listener, Active Threads Over Time listener, Graph Results listener, Hits per Second listener, PerfMon (i.e. Performance Monitoring) Metrics Collector listener, Response Time Graph listener, Response Times Over Time listener, Summary Report listener, Transactions per Second listener and View Results in Table listener.


View Results Tree

Name: View Results Tree

Comments:

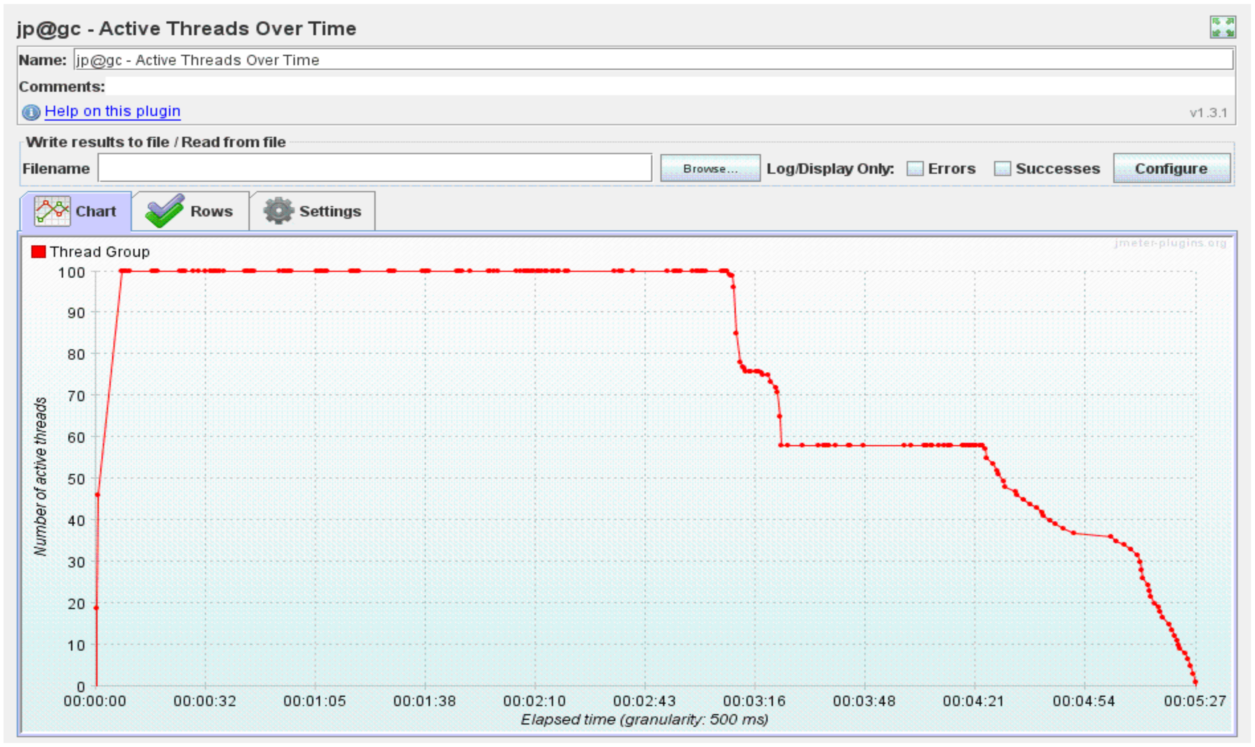
Write results to file / Read from file

Filename

```
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTag
```

```
GET http://localhost/ScrumTaskboard/addTagSelectTag

Cookie Data:
JSESSIONID=968D778D5F859349D063ADAACC6BB027.tc-runtime-1

Request Headers:
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent:
Mozilla/5.0(WindowsNT6.1;WOW64)AppleWebKit/537.36(KHTML,likeGecko)Chrome/48.0.2564.103Saf
ari/537.36
Referer: http://localhost/ScrumTaskboard/addTagSelectTask
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Host: localhost
```

Figure 4.1: View Results Tree listener – 100 threads


jp@gc - Active Threads Over Time

Name: jp@gc - Active Threads Over Time

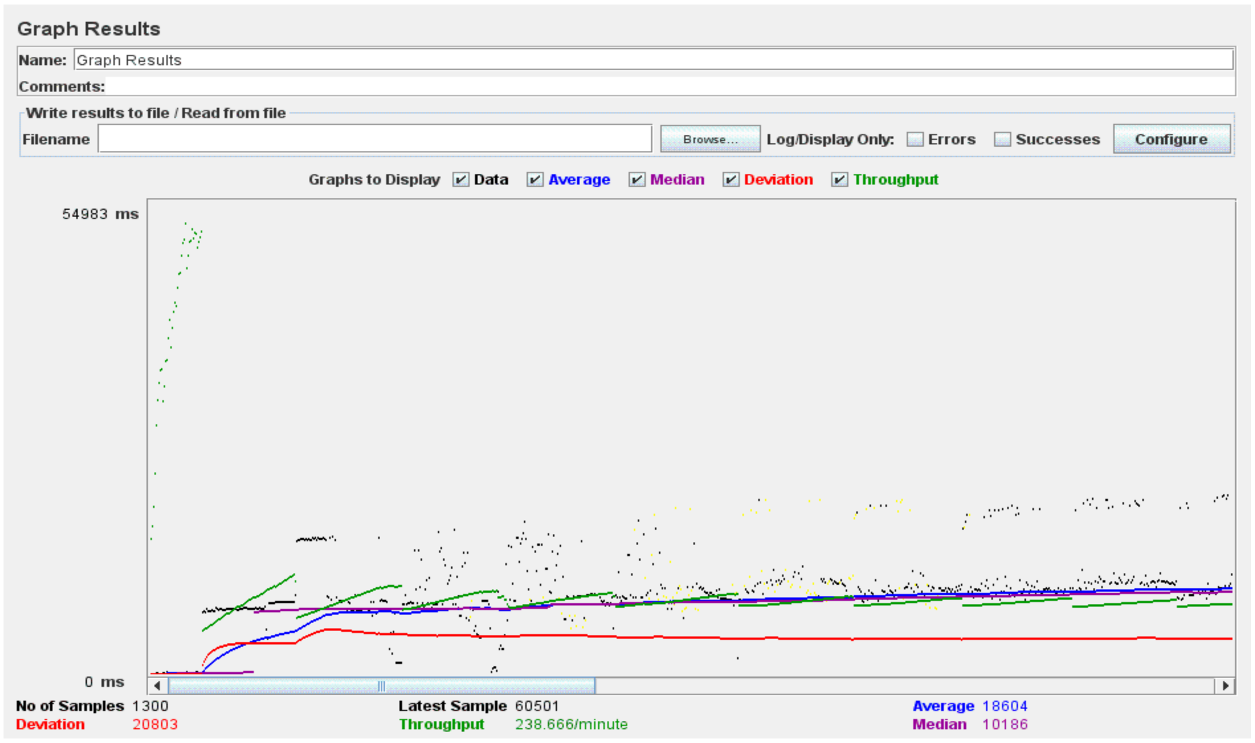
Comments:

Help on this plugin v1.3.1

Write results to file / Read from file

Filename Log/Display Only: Errors Successes

Figure 4.2: Active Threads Over Time listener – 100 threads


Graph Results

Name: Graph Results

Comments:

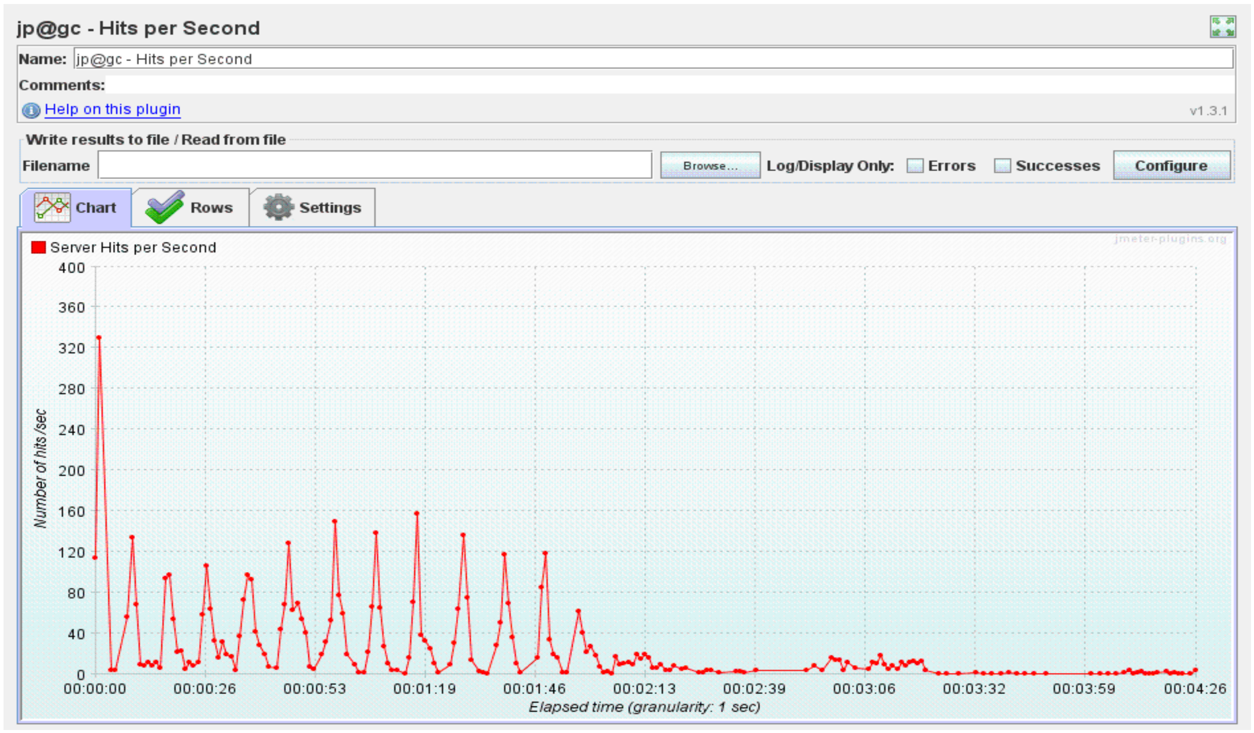
Write results to file / Read from file

Filename Log/Display Only: Errors Successes

Graphs to Display ✓ Data ✓ Average ✓ Median ✓ Deviation ✓ Throughput

No of Samples 1300
Deviation 20803
Latest Sample 60501
Throughput 238.666/minute
Average 18604
Median 10186

Figure 4.3: Graph Results listener – 100 threads


jp@gc - Hits per Second

Name: jp@gc - Hits per Second

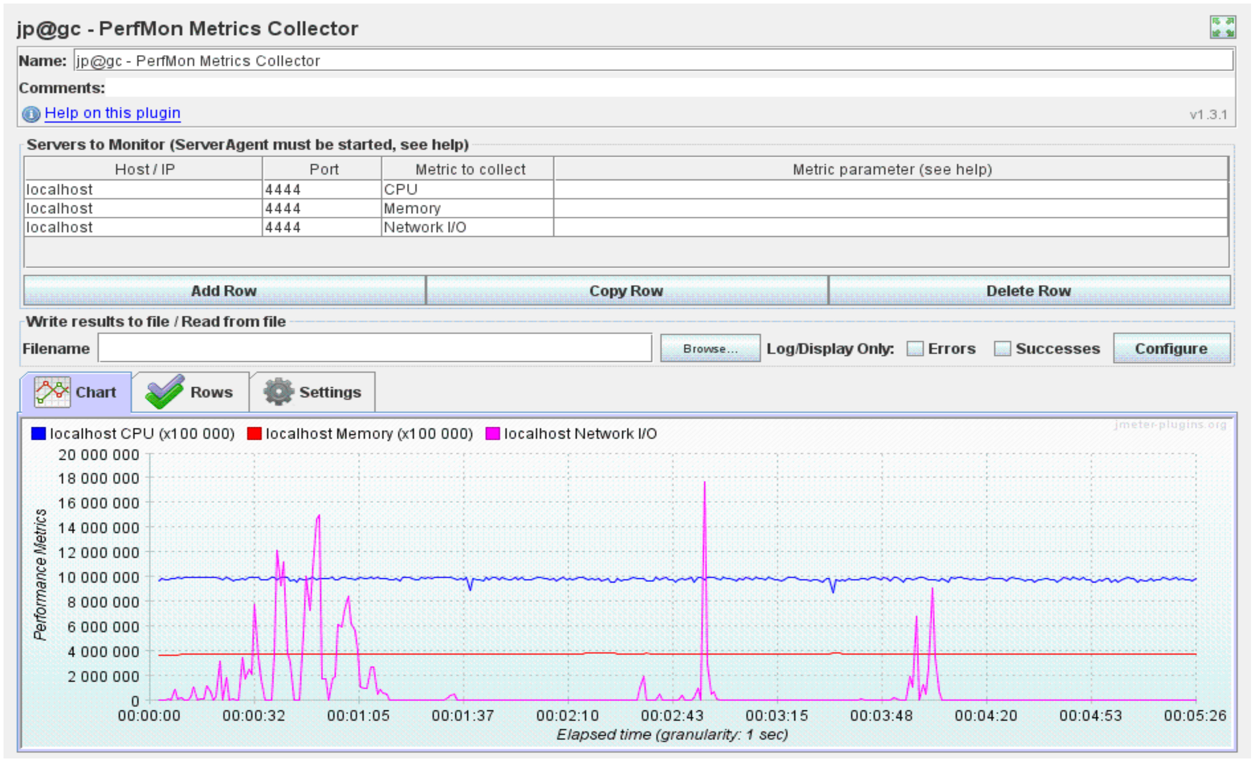
Comments:

Write results to file / Read from file

Filename

Log/Display Only: ☐ Errors ☐ Successes

Figure 4.4: Hits per Second listener – 100 threads


jp@gc - PerfMon Metrics Collector

Name: jp@gc - PerfMon Metrics Collector

Comments:

Servers to Monitor (ServerAgent must be started, see help)

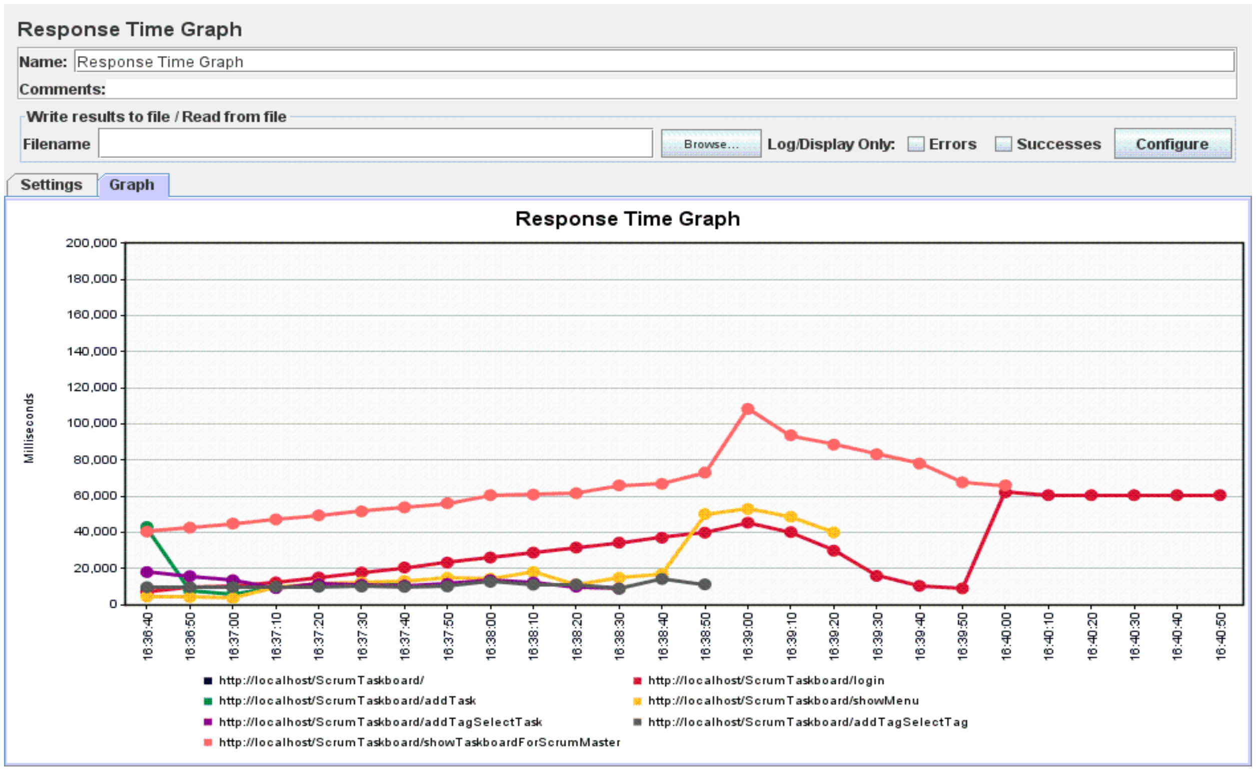
Host / IP	Port	Metric to collect	Metric parameter (see help)
localhost	4444	CPU	
localhost	4444	Memory	
localhost	4444	Network I/O	

Write results to file / Read from file

Filename

Log/Display Only: ☐ Errors ☐ Successes

Figure 4.5: PerfMon Metrics Collector listener – 100 threads


Response Time Graph

Name: Response Time Graph

Comments:

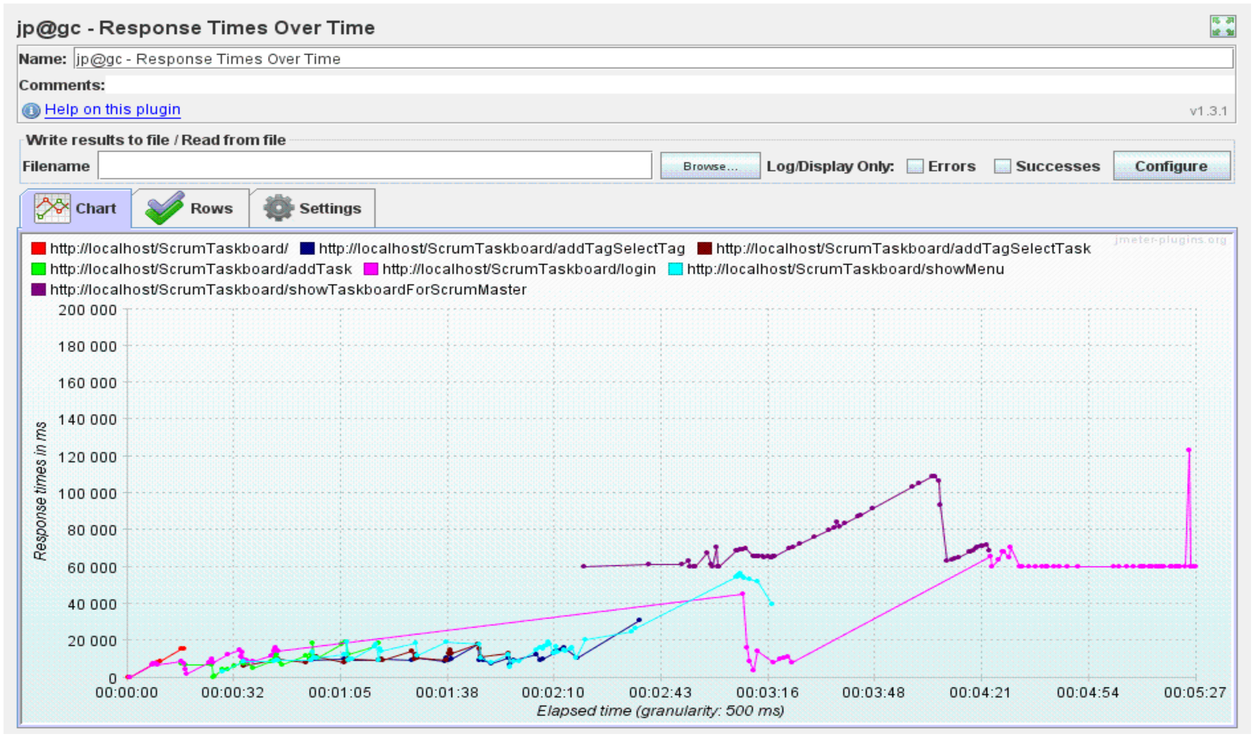
Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

Browse...
Configure
Settings
Graph
Response Time Graph
Milliseconds
■ http://localhost/ScrumTaskboard/
■ http://localhost/ScrumTaskboard/login
■ http://localhost/ScrumTaskboard/addTask
■ http://localhost/ScrumTaskboard/showMenu
■ http://localhost/ScrumTaskboard/addTagSelectTask
■ http://localhost/ScrumTaskboard/addTagSelectTag
■ http://localhost/ScrumTaskboard/showTaskboardForScrumMaster

Figure 4.6: Response Time Graph listener – 100 threads


jp@gc - Response Times Over Time

Name: jp@gc - Response Times Over Time

Comments:

Help on this plugin

v1.3.1

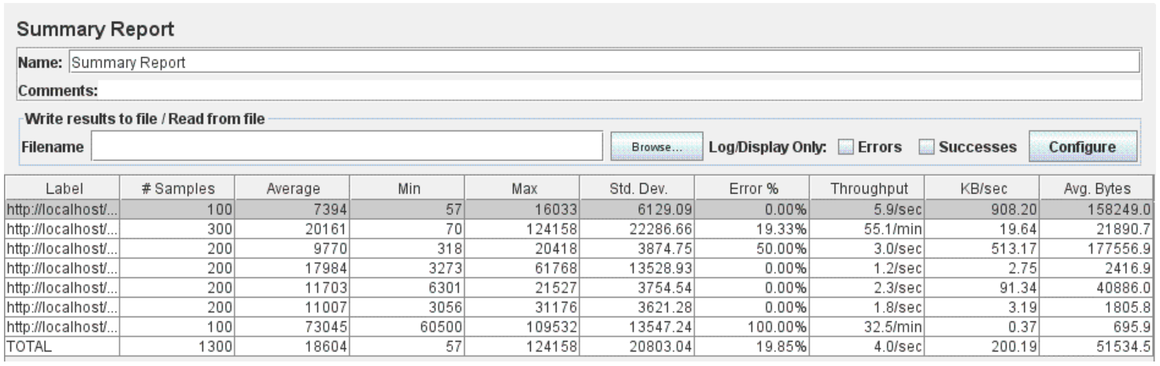
Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

Browse...
Configure
Chart
Rows
Settings
jmeter-plugins.org
http://localhost/ScrumTaskboard/
http://localhost/ScrumTaskboard/addTagSelectTag
http://localhost/ScrumTaskboard/addTagSelectTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
Response times in ms
Elapsed time (granularity: 500 ms)

Figure 4.7: Response Times Over Time listener – 100 threads



Summary Report

Name: Summary Report

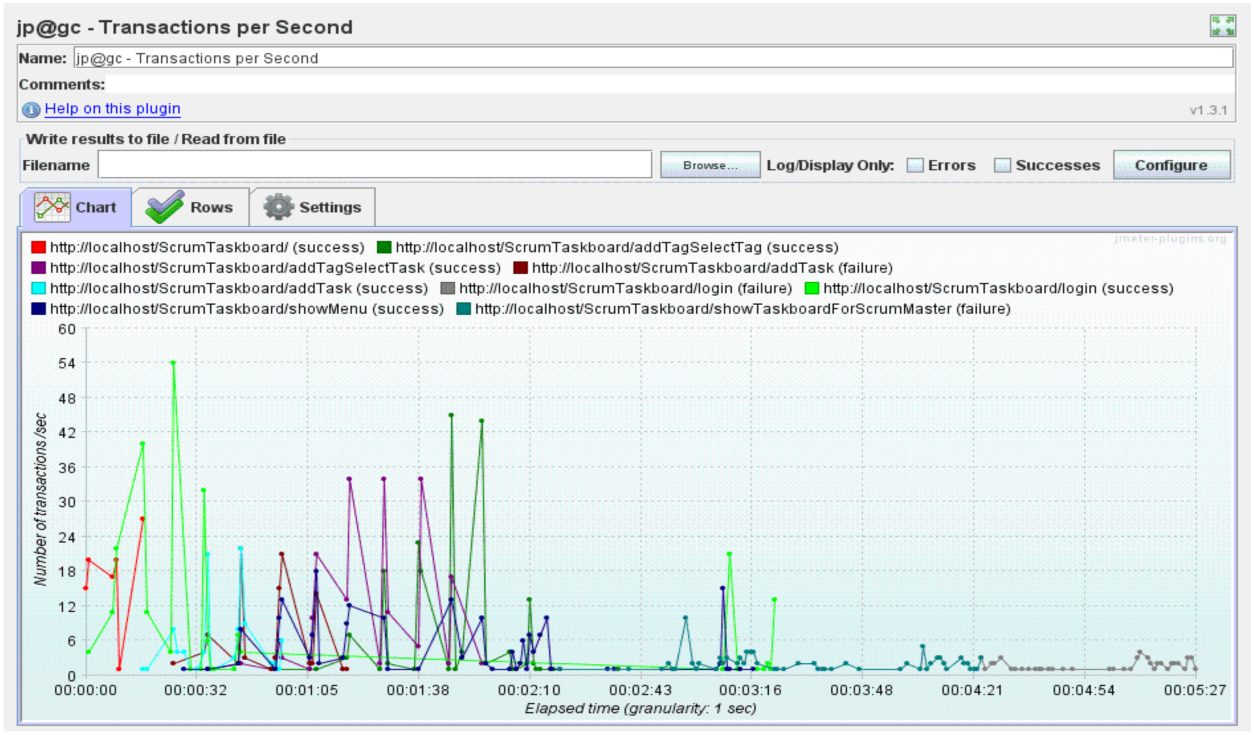
Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: ☐ Errors ☐ Successes | Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	100	7394	57	16033	6129.09	0.00%	5.9/sec	908.20	158249.0
http://localhost/...	300	20161	70	124158	22286.66	19.33%	55.1/min	19.64	21890.7
http://localhost/...	200	9770	318	20418	3874.75	50.00%	3.0/sec	513.17	177556.9
http://localhost/...	200	17984	3273	61768	13528.93	0.00%	1.2/sec	2.75	2416.9
http://localhost/...	200	11703	6301	21527	3754.54	0.00%	2.3/sec	91.34	40886.0
http://localhost/...	200	11007	3056	31176	3621.28	0.00%	1.8/sec	3.19	1805.8
http://localhost/...	100	73045	60500	109532	13547.24	100.00%	32.5/min	0.37	695.9
TOTAL	1300	18604	57	124158	20803.04	19.85%	4.0/sec	200.19	51534.5

Figure 4.8: Summary Report listener – 100 threads



jp@gc - Transactions per Second

Name: jp@gc - Transactions per Second

Comments:

Help on this plugin v1.3.1

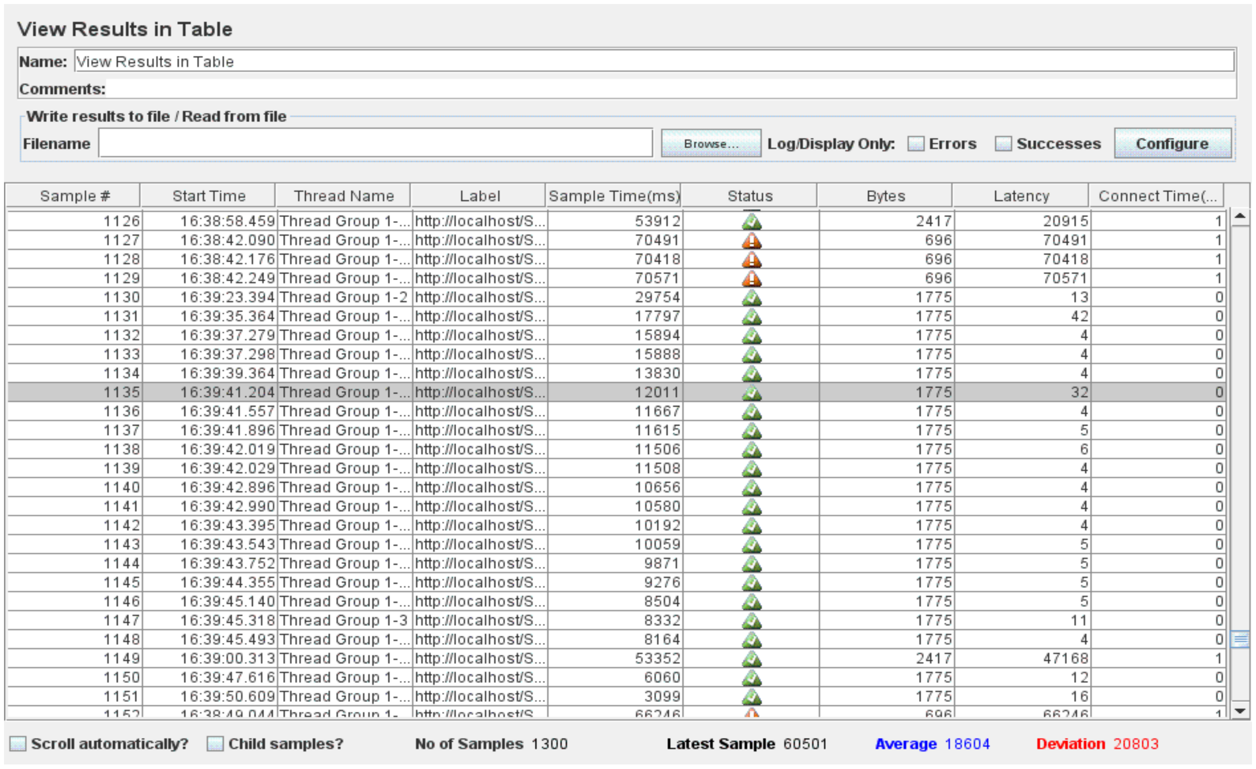
Write results to file / Read from file

Filename | Browse... | Log/Display Only: ☐ Errors ☐ Successes | Configure

Chart | Rows | Settings

🟥 http://localhost/ScrumTaskboard/ (success)
🟩 http://localhost/ScrumTaskboard/addTagSelectTag (success)
🟪 http://localhost/ScrumTaskboard/addTagSelectTask (success)
🟥 http://localhost/ScrumTaskboard/addTask (failure)
🟦 http://localhost/ScrumTaskboard/addTask (success)
⬜ http://localhost/ScrumTaskboard/login (failure)
🟩 http://localhost/ScrumTaskboard/login (success)
🟦 http://localhost/ScrumTaskboard/showMenu (success)
🟦 http://localhost/ScrumTaskboard/showTaskboardForScrumMaster (failure)

Figure 4.9: Transactions per Second listener – 100 threads



View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

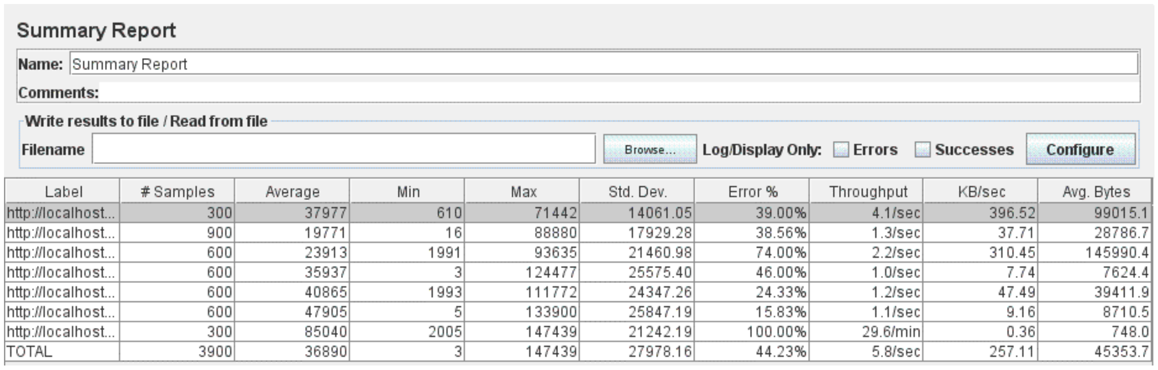
Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(...
1126	16:38:58.459	Thread Group 1-...	http://localhost/S...	53912	✓	2417	20915	1
1127	16:38:42.090	Thread Group 1-...	http://localhost/S...	70491	⚠	696	70491	1
1128	16:38:42.176	Thread Group 1-...	http://localhost/S...	70418	⚠	696	70418	1
1129	16:38:42.249	Thread Group 1-...	http://localhost/S...	70571	⚠	696	70571	1
1130	16:39:23.394	Thread Group 1-2	http://localhost/S...	29754	✓	1775	13	0
1131	16:39:35.364	Thread Group 1-...	http://localhost/S...	17797	✓	1775	42	0
1132	16:39:37.279	Thread Group 1-...	http://localhost/S...	15894	✓	1775	4	0
1133	16:39:37.298	Thread Group 1-...	http://localhost/S...	15888	✓	1775	4	0
1134	16:39:39.364	Thread Group 1-...	http://localhost/S...	13830	✓	1775	4	0
1135	16:39:41.204	Thread Group 1-...	http://localhost/S...	12011	✓	1775	32	0
1136	16:39:41.557	Thread Group 1-...	http://localhost/S...	11667	✓	1775	4	0
1137	16:39:41.896	Thread Group 1-...	http://localhost/S...	11615	✓	1775	5	0
1138	16:39:42.019	Thread Group 1-...	http://localhost/S...	11506	✓	1775	6	0
1139	16:39:42.029	Thread Group 1-...	http://localhost/S...	11508	✓	1775	4	0
1140	16:39:42.896	Thread Group 1-...	http://localhost/S...	10656	✓	1775	4	0
1141	16:39:42.990	Thread Group 1-...	http://localhost/S...	10580	✓	1775	4	0
1142	16:39:43.395	Thread Group 1-...	http://localhost/S...	10192	✓	1775	4	0
1143	16:39:43.543	Thread Group 1-...	http://localhost/S...	10059	✓	1775	5	0
1144	16:39:43.752	Thread Group 1-...	http://localhost/S...	9871	✓	1775	5	0
1145	16:39:44.355	Thread Group 1-...	http://localhost/S...	9276	✓	1775	5	0
1146	16:39:45.140	Thread Group 1-...	http://localhost/S...	8504	✓	1775	5	0
1147	16:39:45.318	Thread Group 1-3	http://localhost/S...	8332	✓	1775	11	0
1148	16:39:45.493	Thread Group 1-...	http://localhost/S...	8164	✓	1775	4	0
1149	16:39:00.313	Thread Group 1-...	http://localhost/S...	53352	✓	2417	47168	1
1150	16:39:47.616	Thread Group 1-...	http://localhost/S...	6060	✓	1775	12	0
1151	16:39:50.609	Thread Group 1-...	http://localhost/S...	3099	✓	1775	16	0
1152	16:38:49.044	Thread Group 1-...	http://localhost/S...	66246	⚠	696	66246	1

Scroll automatically? Child samples? **No of Samples** 1300 **Latest Sample** 60501 **Average** 18604 **Deviation** 20803

Figure 4.10: View Results in Table listener – 100 threads

By using them in association, these output results offer significant performance-oriented metrics which may be further on analyzed from different viewpoints. Among these output listeners, the ones mostly analyzed and used in monitoring various performance insights corresponding to the software application under test, were the View Results Tree listener, Graph Results listener, PerfMon (i.e. Performance Monitoring) Metrics Collector listener, Summary Report listener and View Results in Table listener.

For the particular use case scenario under test, the measured performance indicators were assessed using mainly the Summary Report listener as it offered concrete figures of interest for the conducted research. Furthermore, this listener was considered suitable in order to determine the performance degradation behavior when increasing the simulated number of concurrent workers from 100 to 300. More precisely, by executing the **Login-Add Task-Add Tag for Task-Visualize Taskboard-Logout** testplan with the configured number of 300 concurrent threads, the intent was to compare performance – oriented features with those obtained for the initial number of 100 users. In order to accomplish this, the Summary Report listener provided the necessary information so that to be able to conclude the performance decrease in the context of increasing the workers number. Figure 4.11 below illustrates the Summary Report listener results for a number of 300 concurrent users:


Summary Report
Name: Summary Report
Comments:
Write results to file / Read from file
Filename
Browse...
Log/Display Only: Errors Successes
Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost...	300	37977	610	71442	14061.05	39.00%	4.1/sec	396.52	99015.1
http://localhost...	900	19771	16	88880	17929.28	38.56%	1.3/sec	37.71	28786.7
http://localhost...	600	23913	1991	93635	21460.98	74.00%	2.2/sec	310.45	145990.4
http://localhost...	600	35937	3	124477	25575.40	46.00%	1.0/sec	7.74	7624.4
http://localhost...	600	40865	1993	111772	24347.26	24.33%	1.2/sec	47.49	39411.9
http://localhost...	600	47905	5	133900	25847.19	15.83%	1.1/sec	9.16	8710.5
http://localhost...	300	85040	2005	147439	21242.19	100.00%	29.6/min	0.36	748.0
TOTAL	3900	36890	3	147439	27978.16	44.23%	5.8/sec	257.11	45353.7

Figure 4.11: Summary Report listener – 300 threads

In the analyzed testcase listener, statistics related to the reported error for both 100 and 300 simulated users offered essential insights related to the system's performance capabilities. Particularly for the tested use case scenario, the reported error percentage increased from **19.85%** for 100 users to **44.23%** in case of 300 users, thus denoting a performance downgrade behavior in context of a higher workload performed onto the system.

Apart from this use case scenario, several others were considered appropriate in order to monitor performance – oriented concerns and therefore recorded so that to be further on monitored through application performance monitoring and testing tools such as JMeter. In all these cases, various output listeners were analyzed and their results compared by varying the number of configured simultaneous workers. The general behavior concluded based on the conducted performance analysis indicated a downgrade in system's non-functional, performance - oriented characteristics when increasing the users' workload.

As a result of this initial phase in the dissertation thesis development, essential insights concerning the application's performance features were gathered and next steps might be planned based on this information. Therefore, one of the major targets to be followed in the subsequent phases is to provide a higher serviceability for a larger number of concurrent users, i.e. enhance the operability of the system so that to successfully handle a number of requests in the range of thousands. This performance feature improvement is to be assessed by a considerable reduction in the reported error percentage for an increased number of concurrent workers. The idea behind this statement is that a certain percentage error indicates how many requests were not successfully handled by the application among all user requests oriented towards the system. The higher the reported error, the greater number of requests failed due to the system not being able to serve them. The reason of the system's disability in handling those user requests is perceived as its poor performance characteristics, and adequate approaches should be considered in order to solve this vulnerability.

Next sub-chapter focusses on the considered code improvements to be developed at source code level in order to enhance the overall system's performance. Moreover, it summarizes the main changes executed against the initial code base of the dissertation thesis so that to accomplish the desired goal.

4.2. Performing code improvements

Starting from the performance shortcomings concluded on behalf of the evaluations conducted at dissertation thesis' initial phase, the subsequent step aims to remove, or at least reduce, the threat affecting the application's operation when an increased number of concurrent users try to access the system. Consequently, the possible approaches to accomplish this target were considered and the most suitable one was established by taking into account the actual stage of the dissertation thesis development process.

More precisely, the selected solution in order to reduce the reported error percentage and, therefore determine the system to handle a higher number of simultaneous users, was to perform a review of the initial source code and try to identify possible performance vulnerabilities. This way, the encountered weaknesses might be re-considered and corrected by replacement of more suitable, error-free solutions.

In order to accomplish this phase, of conducting evaluations against the initial code base, two possible approaches were identified: starting to review the source code by inspecting each project file, or considering code review on a use case basis. After analyzing these possible working strategies, the first approach was considered inappropriate due to missing context when exhaustively evaluating the source code without any connection to the underlying functionality that it accomplishes. Therefore, the latter option was decided to be a valid one for our particular context because possible performance flaws are better determined by analyzing the code when following the acquired functionality through the project's architecture and layers.

Specifically, the followed approach to implement code enhancements was fulfilled by considering all possible use cases through the application and reviewing the associated source code which accomplished the underlying functionality. In completing a certain functionality, the flow through the project's layers is the following: starting from the storefront layer, represented by Java Controller classes, service layer methods are invoked in order to accomplish the business logic functionality, which, in turn, take advantage of persistence – level Data Access Object files responsible for accessing the database and update it accordingly. As a consequence, the source code responsible to accomplish the use case scenarios provided by the application under test, was reviewed by proceeding towards the project's layers, from Controller to Data Access Object files.

Evaluating project's source code based on the underlying completed functionality, involved identifying possible performance vulnerabilities that might affect the system's behavior as well as assessing development guidelines and best practices against the actual code base. For each potential threat that might affect various functional and non-functional characteristics of the system under test, the current solution was briefly understood and possible alternatives proposed. Afterwards, the most suitable approach was determined by taking into account the degree of complexity, efficiency and other features, as well as the code changes involved. This way, by assessing a suite of essential criteria against the potential alternative solutions, concrete evidence regarding the accuracy of the chosen solution is obtained.

Finally, the last step in the performing of code improvements stage is the actual development of the established, error – free approach, followed by the required developer tests. Furthermore, the necessary Unit and Integration, dedicated tests were developed in order to assess the accuracy and correctness of the implemented solution.

However, even if the previous, initial version of the dissertation thesis project was definitely of poor quality as compared to the actual, improved version, in which the targets were to provide an efficient, qualitative and error – free solution, there is no guarantee that the overall system's performance has improved. Consequently, the application's performance capabilities have to be evaluated and further on assessed, using similar dedicated tools as in the initial phase. As a result, the following sub-chapter is reserved for measuring whether the implemented code improvements brought significant performance enhancements.

4.3. Evaluating system's performance from JMeter

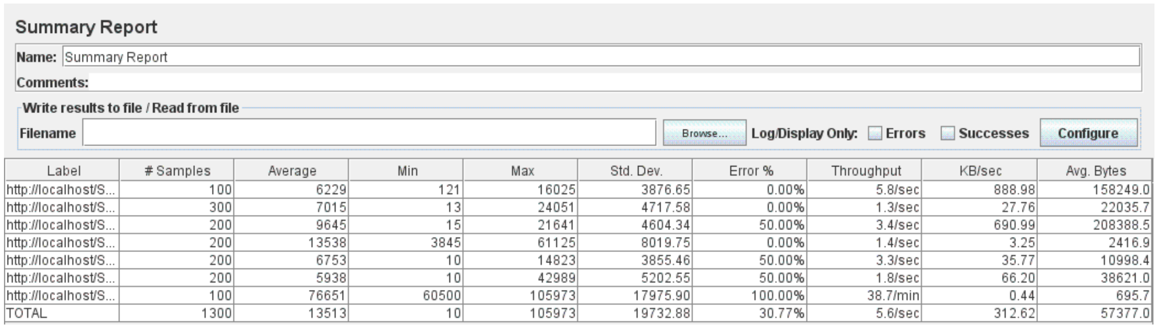
This sub-chapter presents the following phase in the dissertation thesis development, which is intended to evaluate the performance capabilities of the software application after the underlying code base passed through a complete revision followed by the implementation of clean-up, refactoring strategies and improvement solutions. In order to evaluate and monitor the system's performance characteristics, JMeter tool was used and similar measurements to the ones performed in the initial phase were conducted.

Several use case scenarios were recorded and dedicated measurements were conducted so that to obtain accurate information regarding the system's capabilities as compared to the initial version of the tested application and therefore, be able to conclude precise facts concerning the obtained performance enhancements. However, for the scope of this document, the sample use case considered in sub-chapter 4.1 is relevant and measurements executed on its behalf are detailed in the following paragraphs.

Therefore, dedicated JMeter performance measurements were executed for the use case scenario consisting of the following actions: log in as Scrum Master user, add a new task, add a tag for that task, visualize the taskboard and then log out. In order to assess the performance enhancement obtained as a result of the implemented code improvements, the previously created testplan for this scenario was re-executed. Moreover, the JMeter testplan was run under the same conditions, by varying the number of simultaneous users from 100 to 300.

As described in sub-chapter 4.1, there are a suite of output listeners that might be attached to a JMeter testplan in order to offer precise information regarding various performance characteristics, taken from different viewpoints. Among them, in order to accomplish the proposed performance evaluation, intended to further on serve in comparison with the initial application version, the Summary Report listener was considered the most appropriate. More precisely, the output gathered from this listener offered concrete figures of interest for the conducted research. Moreover, the precise data collected from the Summary Report listener was suitable to be compared with the figures obtained in first phase as they were of the same granularity and format. This way, accurate information is obtained regarding whether the implemented code improvements brought any performance enhancements and their magnitude.

Consequently, the **Login-Add Task-Add Tag for Task-Visualize Taskboard-Logout** testplan was executed with both configured 100 and 300 concurrent threads. Figures 4.12 and 4.13 below show the Summary Report listener results obtained by running the JMeter testplan with a simulated number of 100 and respectively 300 concurrent users:


Summary Report

Name: Summary Report

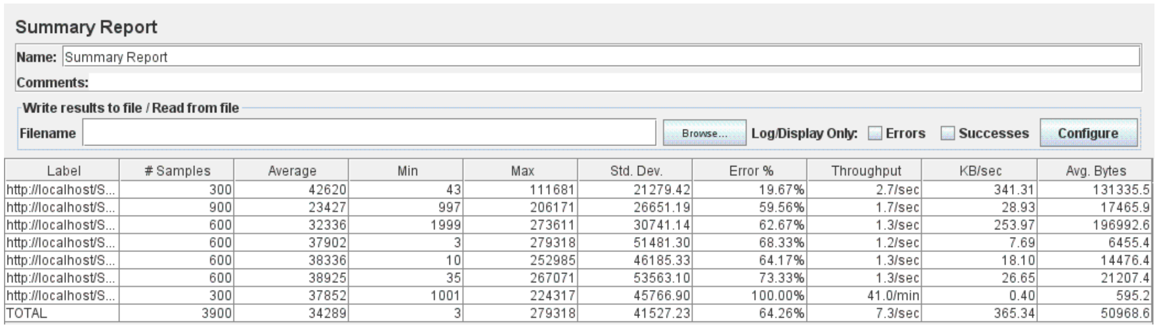
Comments:

Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/S...	100	6229	121	16025	3876.65	0.00%	5.8/sec	888.98	158249.0
http://localhost/S...	300	7015	13	24051	4717.58	0.00%	1.3/sec	27.76	22035.7
http://localhost/S...	200	9645	15	21641	4604.34	50.00%	3.4/sec	690.99	208388.5
http://localhost/S...	200	13538	3845	61125	8019.75	0.00%	1.4/sec	3.25	2416.9
http://localhost/S...	200	6753	10	14823	3855.46	50.00%	3.3/sec	35.77	10998.4
http://localhost/S...	200	5938	10	42989	5202.55	50.00%	1.8/sec	66.20	38621.0
http://localhost/S...	100	76651	60500	105973	17975.90	100.00%	38.7/min	0.44	695.7
TOTAL	1300	13513	10	105973	19732.88	30.77%	5.6/sec	312.62	57377.0

Figure 4.12: Summary Report listener – 100 threads


Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/S...	300	42620	43	111681	21279.42	19.67%	2.7/sec	341.31	131335.5
http://localhost/S...	900	23427	997	206171	26651.19	59.56%	1.7/sec	28.93	17465.9
http://localhost/S...	600	32336	1999	273611	30741.14	62.67%	1.3/sec	253.97	196992.6
http://localhost/S...	600	37902	3	279318	51481.30	68.33%	1.2/sec	7.69	6455.4
http://localhost/S...	600	38336	10	252985	46185.33	64.17%	1.3/sec	18.10	14476.4
http://localhost/S...	600	38925	35	267071	53563.10	73.33%	1.3/sec	26.65	21207.4
http://localhost/S...	300	37852	1001	224317	45766.90	100.00%	41.0/min	0.40	595.2
TOTAL	3900	34289	3	279318	41527.23	64.26%	7.3/sec	365.34	50968.6

Figure 4.13: Summary Report listener – 300 threads

The reports analysis was performed by comparing the collected information with the one obtained from the performance tests executed on behalf of the initial phase (presented in sub-chapter 4.1). More precisely, in both cases of 100 and 300 numbers of simulated users, statistics related to the reported error offered essential insights concerning the system's actual performance capabilities and allowed for further comparison with its initial state. Consequently, in the first case of 100 concurrent users, the reported error percentage was **30.77%** which is significantly worse as compared to the **19.85%** error obtained before the development of code improvements. This result reveals a severe, rather critical, situation due to the fact that the expected performance enhancement obtained as a result of implementing consistent code improvements did not occur and, even worse, a high performance downgrade was identified.

Unfortunately, the same worrying behavior was encountered in the case of 300 concurrent workers, when the reported error percentage was **64.26%**, a considerably worse result as compared to the **44.23%** one obtained for the initially executed performance tests.

Apart from this use case scenario, JMeter performance tests were executed for several others that were considered appropriate for gathering insights into the system's performance capabilities. In all these cases, the Summary Report listener was mainly used as it offered sufficient information on interest for the current research. Tragically, the general behavior concluded based on the conducted performance evaluation indicated a severe downgrade in the system's performance - oriented features as a result of implementing substantial code improvements.

To summarize the findings gathered in this sub-chapter, not only did the expected, targeted enhancement in the system's performance never occur, but also a larger error percentage as compared to the initial version was reported. Furthermore, a severe performance downgrade from the initial application version to the improved one was identified. Unfortunately, the reason behind this issue is unknown and, therefore, specialized research has to be conducted in this direction.

Next sub-chapter presents the investigation studies that were further on performed in order to encounter the root cause of the unexpected performance vulnerability reported for an enhanced code base version supporting the application's functionalities.

4.4. Evaluating system's performance by executing JMeter tests in console (Non-GUI) mode

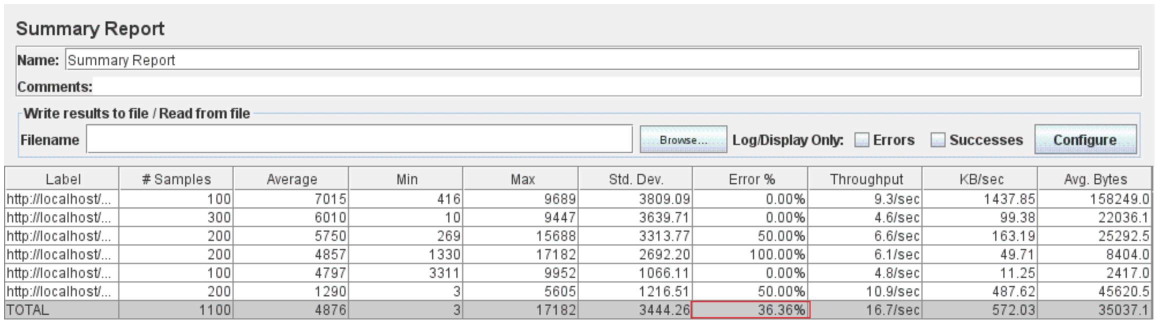
Following the worrying JMeter test results that indicated a severe deterioration of the software application's performance after its underlying source code was consistently improved, a suite of research and investigation activities were conducted in order to obtain a clear resolution of the root cause.

Consequently, according to [21], a system's performance downgrade might not be caused by the application itself, but by the monitoring tool used to record and analyze specific performance – oriented features. More precisely, studies show that JMeter is simply not designed to produce high loads in GUI mode because it might not only freeze, but also consume loads of resources and produce unreliable test results. Therefore, an essential recommendation is to never run a performance test using the JMeter GUI (Graphical User Interface). Alternatively, the most suitable approach is to execute JMeter tests using the command-line (console), while the GUI mode should be dedicated for test recording, development and debugging.

As a consequence of this guideline, the subsequent phase is to obey to the recommendation of executing JMeter performance tests in console mode. Therefore, this sub-chapter follows the execution of JMeter tests in non - GUI mode and their results analysis. For this purpose, another use case scenario through the application was recorded and a corresponding JMeter testplan created in order to serve for performance measurements. As opposed to the previously presented use case, the newly considered one is more appropriate for the investigations conducted throughout several sub-chapters of the thesis paper. The use case flow through the application consists of the following actions: login as a Scrum Master user, add a new developer, delete an old developer (from the team/project managed by the Scrum Master) and logout, being referred as **Login_Add New Developer_Delete Old Developer_Logout** testcase. The intent of this phase is to apply the previously described JMeter best practice and assess its results on behalf of the particular software application under test.

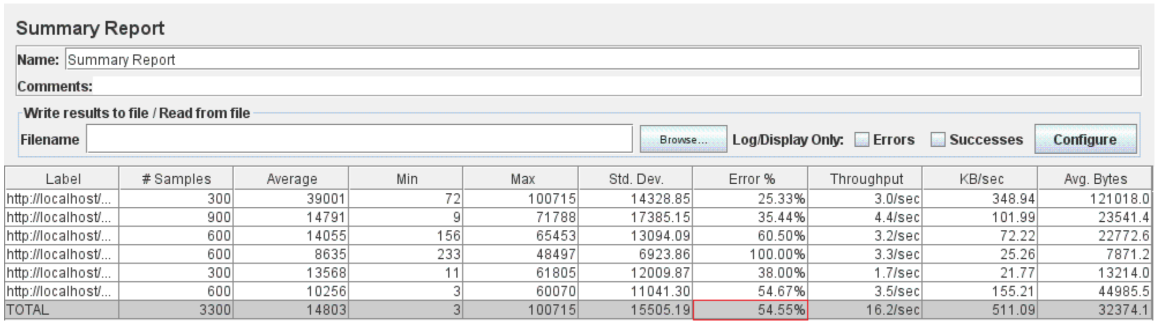
However, in order to evaluate whether executing the previously mentioned use case in JMeter's console mode really leads to better performance measurements results as compared to running the same testcase in GUI mode, the reports obtained in the latter case have to be collected. Therefore, the created JMeter testcase was executed in GUI mode and the results of output listeners were gathered. As for the scope of the intended performance comparison, only the records from the Summary Report listener are of high relevance, and therefore, presenting results from other listeners was skipped.

The records obtained from JMeter's Summary Report listener by running the **Login_Add New Developer_Delete Old Developer_Logout** testcase in GUI mode with a configured number of 100, 300 and 500 concurrent threads are illustrated in Figures 4.14, 4.15 and 4.16 as follows:



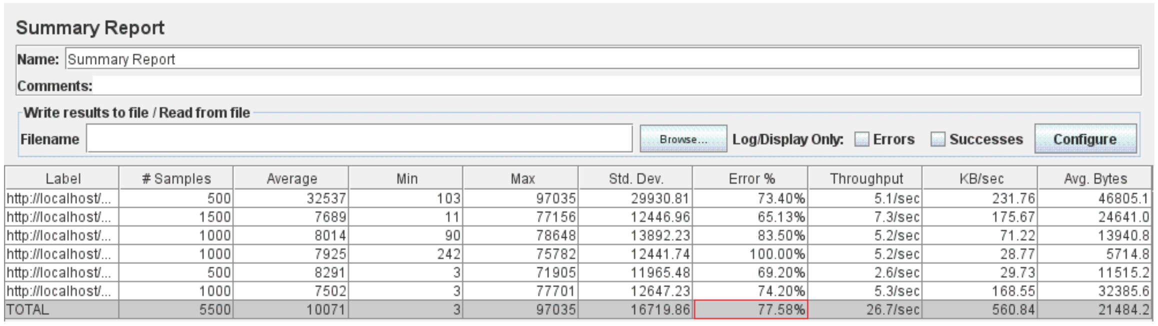
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	100	7015	416	9689	3809.09	0.00%	9.3/sec	1437.85	158249.0
http://localhost/...	300	6010	10	9447	3639.71	0.00%	4.6/sec	99.38	22036.1
http://localhost/...	200	5750	269	15688	3313.77	50.00%	6.6/sec	163.19	25292.5
http://localhost/...	200	4857	1330	17182	2692.20	100.00%	6.1/sec	49.71	8404.0
http://localhost/...	100	4797	3311	9952	1066.11	0.00%	4.8/sec	11.25	2417.0
http://localhost/...	200	1290	3	5605	1216.51	50.00%	10.9/sec	487.62	45620.5
TOTAL	1100	4876	3	17182	3444.26	36.36%	16.7/sec	572.03	35037.1

Figure 4.14: Summary Report listener – 100 threads



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	300	39001	72	100715	14328.85	25.33%	3.0/sec	348.94	121018.0
http://localhost/...	900	14791	9	71788	17385.15	35.44%	4.4/sec	101.99	23541.4
http://localhost/...	600	14055	156	65453	13094.09	60.50%	3.2/sec	72.22	22772.6
http://localhost/...	600	8635	233	48497	6923.86	100.00%	3.3/sec	25.26	7871.2
http://localhost/...	300	13568	11	61805	12009.87	38.00%	1.7/sec	21.77	13214.0
http://localhost/...	600	10256	3	60070	11041.30	54.67%	3.5/sec	155.21	44985.5
TOTAL	3300	14803	3	100715	15505.19	54.55%	16.2/sec	511.09	32374.1

Figure 4.15: Summary Report listener – 300 threads



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	500	32537	103	97035	29930.81	73.40%	5.1/sec	231.76	46805.1
http://localhost/...	1500	7689	11	77156	12446.96	65.13%	7.3/sec	175.67	24641.0
http://localhost/...	1000	8014	90	78648	13892.23	83.50%	5.2/sec	71.22	13940.8
http://localhost/...	1000	7925	242	75782	12441.74	100.00%	5.2/sec	28.77	5714.8
http://localhost/...	500	8291	3	71905	11965.48	69.20%	2.6/sec	29.73	11515.2
http://localhost/...	1000	7502	3	77701	12647.23	74.20%	5.3/sec	168.55	32385.6
TOTAL	5500	10071	3	97035	16719.86	77.58%	26.7/sec	560.84	21484.2

Figure 4.16: Summary Report listener – 500 threads

The performance measurements presented in these reports correspond to the enhanced version of the system, i.e. after implementing the code improvements. The following step is to execute the three JMeter testcases (for 100, 300 and 500 simultaneous users) in console mode and collect the results to be further on compared with the ones obtained from JMeter's GUI mode.

Therefore, in order to run the JMeter testplan in non-GUI mode, a console has to be opened in the tool's *bin* directory where the following command has to be specified:

```
jmeter -n -t testplan.jmx -l resultFile.jtl
```

In the previous command, the meaning of the parameters in bold is the following:

- **–n** specifies JMeter to run in non-GUI mode
- **-t** indicates that the following argument represents the name of JMX file that contains the testplan
- **-l** indicates that the next argument specifies the JTL file where the testplan's results will be logged

For the particular case of the considered scenario, the *testplan.jmx* file corresponds to the recorded testplan which has been configured with the desired number of concurrent threads. Therefore, in each of the three console executions (for 100, 300 and 500 configured workers), both *testplan.jmx* file and *resultFile.jtl* will be different as distinct testplan configurations are required and logs from all cases have to be collected.

In order to evaluate whether executing the JMeter testplan in non-GUI mode brought any performance improvement, both the information logged in the console and the output results file have to be analyzed. For this purpose, in the case of **100** configured users, the following logs were listed in the console while the testplan was running:

```
Creating summariser <summary>
Created the tree successfully using testplan.jmx

Starting the test @ Sat Mar 12 17:25:16 EET 2016 (1457796316464)
Waiting for possible shutdown message on port 4445

summary +    119 in    18s =    6.8/s Avg:  6997 Min:    50 Max:  8643 Err:     0 (0.00%)
Active: 100 Started: 100 Finished: 0

summary +    333 in    26s =   13.0/s Avg:  7664 Min:   244 Max: 18441 Err:    54
(16.22%) Active: 100 Started: 100 Finished: 0

summary =    452 in  43.3s =   10.4/s Avg:  7488 Min:    50 Max: 18441 Err:    54
(11.95%)

……………………………………………………………………………………………………………

summary +    167 in    10s =   17.0/s Avg:  7985 Min:     6 Max: 27722 Err:    77
(46.11%) Active: 0 Started: 100 Finished: 100

summary =   1100 in   113s =    9.7/s Avg:  9104 Min:     5 Max: 37799 Err:   400
(36.36%)

Tidying up ...    @ Sat Mar 12 17:27:09 EET 2016 (1457796429948)

... end of run
```

Apart from the information logged in the command line, the report for this testcase was generated in the specified .jtl file. As the console logs offered the information of interest for our research, the output file’s contents were not included in this document. Consequently, as the error percentage was used as performance indicator in the previous sub-chapter(s), the results analysis of testplan’s execution in console mode was also performed based on the reported error values.

In order to determine whether the expected performance enhancement brought by running the JMeter testplan in non-GUI mode was actually obtained, a comparison based on the error percentage was conducted among the two modes of JMeter testcase execution: GUI and console (non-GUI). As described in Figure 4.14, the error reported by executing the considered testcase with 100 threads in GUI mode was **36.36%**. Moreover, the console logs, corresponding to the testcase execution in non-GUI mode for 100 users, reported a total error of **36.36%**. Consequently, the same error percentage as the one reported from the testcase execution in GUI mode, was logged in console mode, thus no performance improvement was obtained.

Despite of the disappointment resulted from the fact that the expected performance enhancement did not occur, the considered testcase, configured once with 300 and once with 500 workers, was still executed in console mode and the logged results compared with the ones from GUI mode.

Therefore, in the case of **300** configured users, the following information was listed in the console log while the testplan was executing:

```
Creating summariser <summary>
Created the tree successfully using testplan.jmx
Starting the test @ Sat Mar 12 20:24:01 EET 2016 (1457807041938)
Waiting for possible shutdown message on port 4445

summary +     76 in     28s =     2.7/s Avg: 19843 Min:  2022 Max: 25883 Err:     42
(55.26%) Active: 300 Started: 300 Finished: 0

summary +    592 in   30.1s =    19.6/s Avg: 14754 Min:     4 Max: 57102 Err:    445
(75.17%) Active: 299 Started: 300 Finished: 1

summary =    668 in     58s =    11.5/s Avg: 15333 Min:     4 Max: 57102 Err:    487
(72.90%)

summary +    545 in     30s =    18.2/s Avg: 19637 Min:     4 Max: 77499 Err:    314
(57.61%) Active: 287 Started: 300 Finished: 13

…

summary +    351 in   30.1s =    11.7/s Avg: 16112 Min:    14 Max: 51011 Err:    143
(40.74%) Active: 57 Started: 300 Finished: 243
```

```
summary =   3180 in   298s =   10.7/s Avg: 22452 Min:     4 Max: 98274 Err:  1769 (55.63%)

summary +    120 in    12s =   10.4/s Avg:  6591 Min:     6 Max: 26216 Err:    61 (50.83%) Active: 0 Started: 300 Finished: 300

summary =   3300 in   309s =   10.7/s Avg: 21875 Min:     4 Max: 98274 Err:  1830 (55.45%)

Tidying up ...    @ Sat Mar 12 20:29:11 EET 2016 (1457807351628)

... end of run
```

Also for this case the information of interest is available in the console logs so analyzing the output file's contents might be skipped. Similar to the previous case (of 100 configured workers), the potential performance enhancement brought by running the JMeter testplan in non-GUI mode was determined by comparing the error values obtained for the two JMeter execution modes: GUI and console (non-GUI). Therefore, as illustrated in Figure 4.15, the testcase execution in GUI mode for 300 concurrent users reported an error of **54.55%**. Furthermore, the console logs, corresponding to the testcase execution in non-GUI mode for 300 users, reported a total error of **55.45%**. Consequently, greater error values were reported than the ones obtained when running the testcase from JMeter GUI, thus no performance enhancement was denoted.

Even if worst results were obtained when running JMeter in non-GUI mode, the research proceeded with the testcase execution for a number of **500** configured threads. The following logs were listed in the command line while the testplan was running:

```
Creating summariser <summary>
Created the tree successfully using testplan.jmx
Starting the test @ Sat Mar 12 20:38:47 EET 2016 (1457807927459)
Waiting for possible shutdown message on port 4445

summary +    533 in  12.3s =   43.4/s Avg:  2803 Min:  1994 Max: 10493 Err:   532 (99.81%) Active: 500 Started: 500 Finished: 0

summary +   1432 in    30s =   47.8/s Avg:  3870 Min:   344 Max: 39716 Err:  1421 (99.23%) Active: 423 Started: 500 Finished: 77

summary =   1965 in  42.3s =   46.5/s Avg:  3580 Min:   344 Max: 39716 Err:  1953 (99.39%)

summary +    880 in  31.3s =   28.1/s Avg: 13418 Min:     7 Max: 71453 Err:   809 (91.93%) Active: 380 Started: 500 Finished: 120

…
```

```
summary +    281 in  27.5s =   10.2/s Avg: 16046 Min:     7 Max: 50477 Err:   119
(42.35%) Active: 85 Started: 500 Finished: 415

summary =   5280 in   312s =   16.9/s Avg: 16670 Min:     6 Max: 118223 Err:  4219
(79.91%)

summary +    220 in    23s =    9.6/s Avg:  8202 Min:     3 Max: 33178 Err:   109
(49.55%) Active: 0 Started: 500 Finished: 500

summary =   5500 in   335s =   16.4/s Avg: 16331 Min:     3 Max: 118223 Err:  4328
(78.69%)

Tidying up ...    @ Sat Mar 12 20:44:22 EET 2016 (1457808262979)

... end of run
```

By following the same approach as in the previous cases (of 100 and 300 configured workers), the expected performance enhancement brought by running the JMeter testplan in non-GUI mode was determined by comparing the error values obtained for the two JMeter execution modes: GUI and console (non-GUI). Consequently, as presented in Figure 4.16, the testcase execution in GUI mode for 500 concurrent users reported an error of **77.58%**. Moreover, the console logs, corresponding to the testcase execution in non-GUI mode for a number of 500 users, reported a total error of **78.69%**. Therefore, greater error values were reported as compared to the ones obtained when running the testcase from JMeter GUI, thus no performance improvement was denoted. Even more, the percentage error increased when executing the testplan in non-GUI mode.

As a summary of the research conducted in this sub-chapter, some insights into the possible causes of software applications downgrade were gathered. Furthermore, literature shows that possible performance bottlenecks might not be caused by the application itself, but by the monitoring tool. More precisely, specialized studies state that running JMeter in GUI mode will consume loads of resources and produce unreliable test results. Therefore, an essential recommendation is to execute JMeter tests using the command-line (console) and reserve the GUI usage for test recording, development and debugging activities. As a result of this guideline, for a sample use case scenario, the JMeter tests were executed in both GUI and console modes and a performance comparison was performed in the end. However, for all of the three cases (with 100, 300 and 500 concurrent users), not only did the expected enhancement in the system's performance for the console mode execution never occur, but also larger error percentages as compared to the ones obtained from JMeter GUI mode were reported.

Next sub-chapter describes the further investigation performed in order to encounter the root cause of the system's performance vulnerability. More precisely, VisualVM tool is used to conduct advanced application monitoring and obtain accurate information regarding the system's bottleneck.

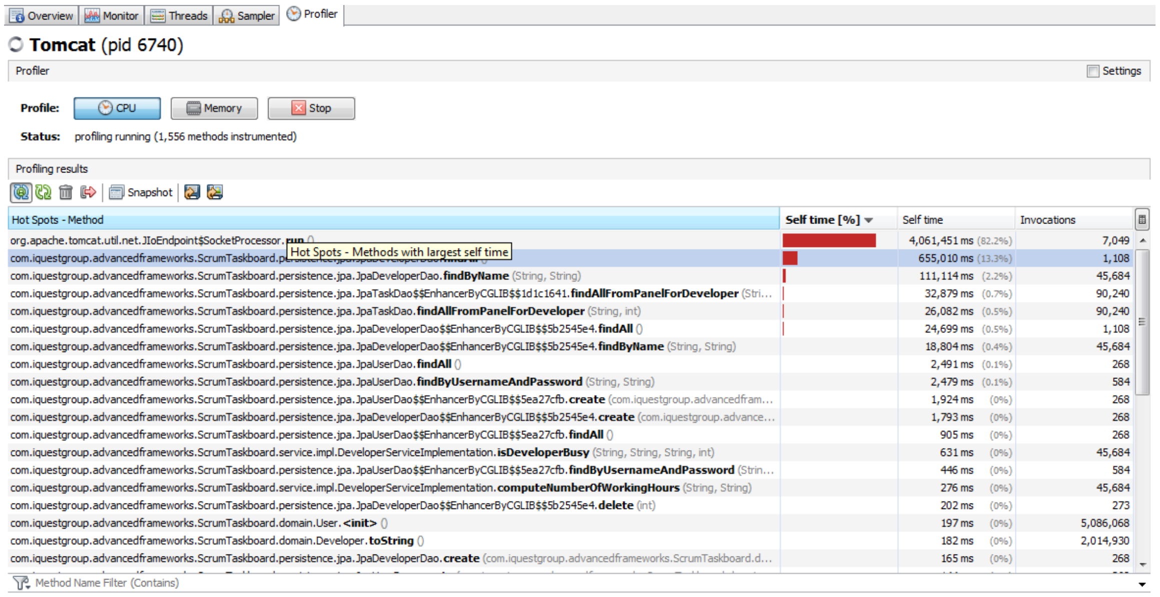
4.5. Extended monitoring of the system’s performance from VisualVM

The next step in dissertation thesis development is to successfully determine the root cause of the previously encountered performance issues so that to further on propose concrete solutions to be implemented in order to increase the overall performance of the application. To accomplish this, the simulated testcases executed by taking advantage of the functionalities offered by JMeter were associated with the ones provided by VisualVM performance monitoring tool.

VisualVM as an application performance monitoring tool which freely comes when installing Java JDK, offers several monitoring options, among which the profiling and sampling ones are of interest for our research. The profiler and sampler tools accessible from the VisualVM’s GUI are available for both CPU and memory load as performance indicators. However, for the particular scope of our investigation, monitoring the CPU load while sampling or profiling the tested application provides sufficient information regarding the system’s point of failure.

According to [23], by instrumenting a web application under test, profiling adds a constant amount of extra execution time to every single method call. Sometimes, this results in adding large amount of time to the execution which may even last hours. VisualVM’s sampler works by taking a dump of all of the threads of execution on a fairly regular basis, and uses this to determine how much CPU time each method spends. Usually, sampling takes a constant amount of time each second to record stack traces for each thread, thus only adding 5 – 10 minutes of execution time in total, while still succeeding to provide information regarding the source causes of potential performance vulnerabilities. However, sampling has the drawback that the number of invocations recorded is not necessarily accurate, since a short method could easily start and finish between stack dumps.

Therefore, the literature recommendation is to use sampling for recording the amount of CPU time or Memory consumption, rather than the number of invocations, thus having the chance to identify performance problems, much faster than the standard profiler.


Overview
Monitor
Threads
Sampler
Profiler
Settings
CPU
Memory
Stop
Snapshot
Method Name Filter (Contains)
Hot Spots - Methods with largest self time

Tomcat (pid 6740)

Profiler

Profile: CPU | Memory | Stop

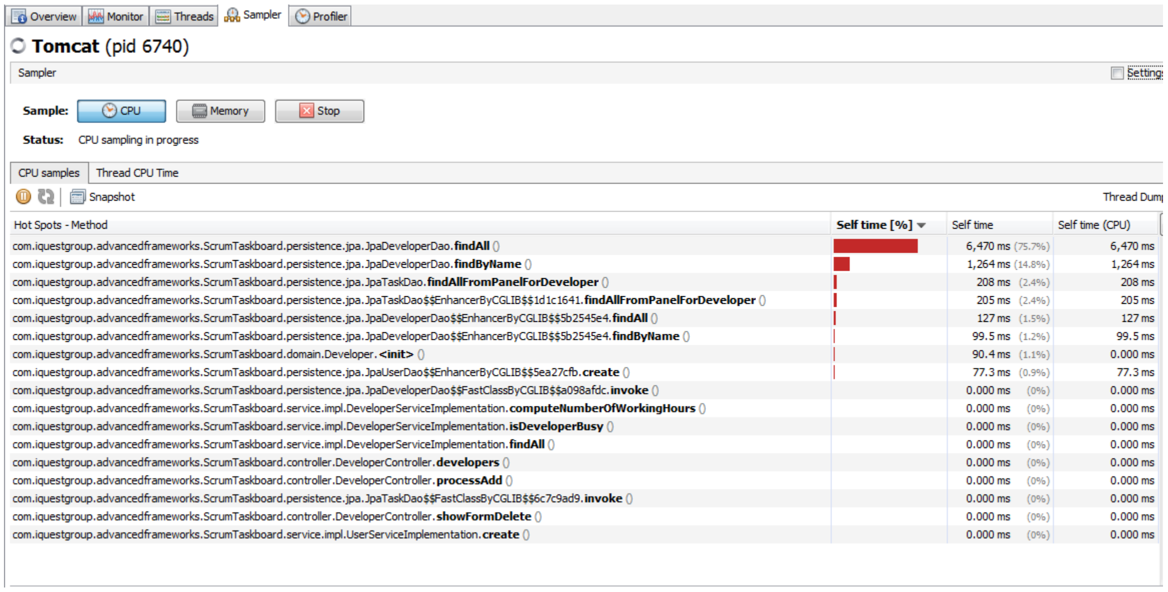
Status: profiling running (1,556 methods instrumented)

Profiling results

Hot Spots - Method	Self time [%]	Self time	Invocations
org.apache.tomcat.util.net.JIoEndpoint$SocketProcessor.**run** ()	🟥	4,061,451 ms (82.2%)	7,049
com.iquestgroup.advancedframeworks.ScrumTaskboard.p[illegible]	🟥	655,010 ms (13.3%)	1,108
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**findByName** (String, String)	🟥	111,114 ms (2.2%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao$$EnhancerByCGLIB$$1d1c1641.**findAllFromPanelForDeveloper** (Stri…	🟥	32,879 ms (0.7%)	90,240
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao.**findAllFromPanelForDeveloper** (String, int)	🟥	26,082 ms (0.5%)	90,240
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findAll** ()	🟥	24,699 ms (0.5%)	1,108
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findByName** (String, String)		18,804 ms (0.4%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao.**findAll** ()		2,491 ms (0.1%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao.**findByUsernameAndPassword** (String, String)		2,479 ms (0.1%)	584
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**create** (com.iquestgroup.advancedfram…		1,924 ms (0%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**create** (com.iquestgroup.advance…		1,793 ms (0%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**findAll** ()		905 ms (0%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**isDeveloperBusy** (String, String, String, int)		631 ms (0%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**findByUsernameAndPassword** (Strin…		446 ms (0%)	584
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**computeNumberOfWorkingHours** (String, String)		276 ms (0%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**delete** (int)		202 ms (0%)	273
com.iquestgroup.advancedframeworks.ScrumTaskboard.domain.User.**<init>** ()		197 ms (0%)	5,086,068
com.iquestgroup.advancedframeworks.ScrumTaskboard.domain.Developer.**toString** ()		182 ms (0%)	2,014,930
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**create** (com.iquestgroup.advancedframeworks.ScrumTaskboard.d…		165 ms (0%)	268

Figure 4.17: CPU usage information from VisualVM Profiler

As a result, for the previously presented use case scenario (login as a Scrum Master user, add a new developer, delete an old developer (from the team/project managed by the Scrum Master) and logout) executed in the context of 100 concurrent threads (configured in JMeter), the main bottleneck(s) of the system were determined by using both profiler and sampler tools. Figures 4.17 and 4.18 present CPU usage information when profiling and sampling the considered use case.



Tomcat (pid 6740)

Sampler

Sample: CPU | Memory | Stop

Status: CPU sampling in progress

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**findAll** ()	🟥	6,470 ms (75.7%)	6,470 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**findByName** ()	🟥	1,264 ms (14.8%)	1,264 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao.**findAllFromPanelForDeveloper** ()	🟥	208 ms (2.4%)	208 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao$$EnhancerByCGLIB$$1d1c1641.**findAllFromPanelForDeveloper** ()	🟥	205 ms (2.4%)	205 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findAll** ()	🟥	127 ms (1.5%)	127 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findByName** ()	🟥	99.5 ms (1.2%)	99.5 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.domain.Developer.**<init>** ()	🟥	90.4 ms (1.1%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**create** ()	🟥	77.3 ms (0.9%)	77.3 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$FastClassByCGLIB$$a098afdc.**invoke** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**computeNumberOfWorkingHours** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**isDeveloperBusy** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**findAll** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.controller.DeveloperController.**developers** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.controller.DeveloperController.**processAdd** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao$$FastClassByCGLIB$$6c7c9ad9.**invoke** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.controller.DeveloperController.**showFormDelete** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.UserServiceImplementation.**create** ()		0.000 ms (0%)	0.000 ms

Figure 4.18: CPU usage information from VisualVM Sampler

These two images offer information related to the bottleneck points of the application, by listing the specific functionalities that consume most of CPU's time. By taking advantage of these results, corrective, problem-oriented solution might be implemented in order to overcome the performance vulnerabilities faced by the application.

Starting from the results obtained by profiling and sampling activities (Figures 4.17 and 4.18) for the considered use case scenario, it can be easily determined that most of the CPU time is spent on calling functionality at the persistence layer. This fact is proved by the suite of database methods invocation that are reported in VisualVM's profiler and sampler as being the most CPU consuming spots. More precisely, the method call hierarchy is illustrated in both figures, ordered by the CPU time consumption when the considered use case scenario is performed: *findByName()*, *findAllFromPanelForDeveloper()*, *findAll()*, *findByUsernameAndPassword()*, *create()*, *delete()*. Therefore, the main CPU bottleneck is caused by methods from the persistence layer (when querying the database) and, thus, further improvements should be implemented at the database level.

To conclude this sub-chapter, the association between the JMeter and VisualVM tools provides accurate information regarding the precise source of failure for the web application under test. Furthermore, by analyzing the CPU usage – based results, specific research might be conducted and dedicated solutions designed and implemented.

The following sub-chapter focuses on exploring several techniques in order to remove the database bottleneck and selecting the appropriate solution to be further on implemented.

4.6. Performing improvements at database level (adding indexes) and monitoring the performance enhancements obtained

More precisely, for removing the database bottleneck, there are several techniques that may be explored and appropriate solutions implemented. By studying documentation related to database enhancements and methods to be applied on a web application to improve the persistence layer access, a series of rules and guidelines were considered. As one of the main strategy to be applied at persistence layer is adding database indexes where appropriate, this topic was studied in detail and best practices taken into account [37].

According to Singh [38], in order to fix the database bottleneck which determined the application to become unresponsive when a large number of users were trying to access it, several persistence layer strategies were applied where appropriate and the results analyzed after each step. Specifically for the described use case scenario, by means of SQL syntax, slow queries were determined and database indexes implemented in the adequate manner.

The main reason behind identifying slow SQL queries is the fact that one of the most important steps in optimizing and tuning MySQL databases is to identify the queries that are causing problems. Therefore, it must be determined what queries are taking the longest time to complete and which ones are slowing down the MySQL server. The recommendation is to select the most 10 problematic queries and try to optimize them properly in hope of an immediate improvement of the overall database performance [39].

In the particular context of our research, the first step in removing the database bottleneck is to identify the SQL queries reported as being slow. Afterwards, design and analysis activities have to be conducted in order to establish the proper indexes to be added. The following step consists of the implementation of the database indexes, followed by a performance testing using JMeter. In case performance improvements are obtained by database level optimization, the added indexes will be kept into the code base, otherwise they will be removed.

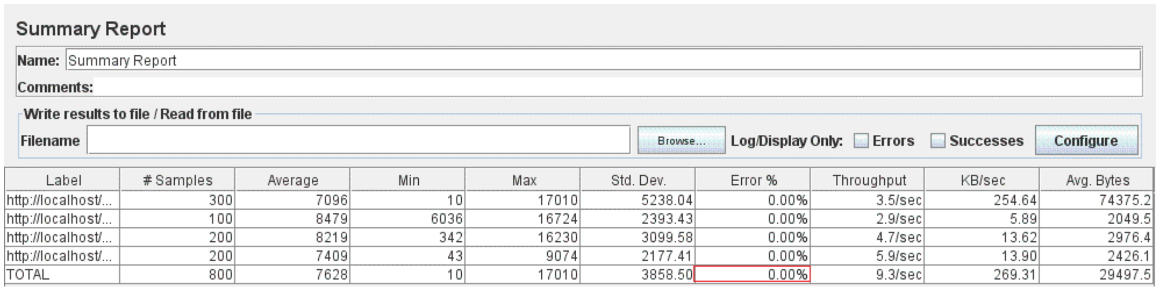
This approach was executed for all use cases identified through the application under test in order to enhance the overall performance by removing database – level bottlenecks. However, as for the scope of this sub-chapter, the process of adding database indexes was presented only for the most relevant use case scenarios. Therefore, the considered scenario for this purpose consists of the following actions: login as a Scrum Master user, visualize all proposed upgrades for various skills owned by developers (belonging to the project/team managed by this user), select the skill to be upgraded for a certain developer, repeat this process until done (for other skills and developers), and logout. The algorithm which determines these upgrades takes into account the actual skills possessed by a certain developer (described by a percentage value associated to each skill), as well as the tasks that were completed by him/her and proposes a newly computed percentage value. This use case scenario will be referred throughout this paper as **Login_Skill Upgrades for developer_Logout** testcase.

By following the previously mentioned approach for implementing database indexes, the first step was to take advantage of the functionalities offered by MySQL in order to identify the SQL queries reported as being slow for this use case. As the majority of encountered slow queries were the ones searching the database for a certain user based on its username and password, the initial solution was to add an index for the *username* attribute of the *User* entity model. This was performed from Java code as follows:

```
@NotEmpty
@Column(name="username")
@Index(name = "usernameIDX")
private String username;
```

After adding this index, the corresponding SQL query was not logged anymore as being slow, fact which denotes a performance enhancement brought by this solution. Even more, dedicated performance tests were executed both before and after the index addition in order to establish whether any performance improvement was obtained and the order of its magnitude. These tests were performed using JMeter tool and were executed for 100, 300 and 500 concurrent users.

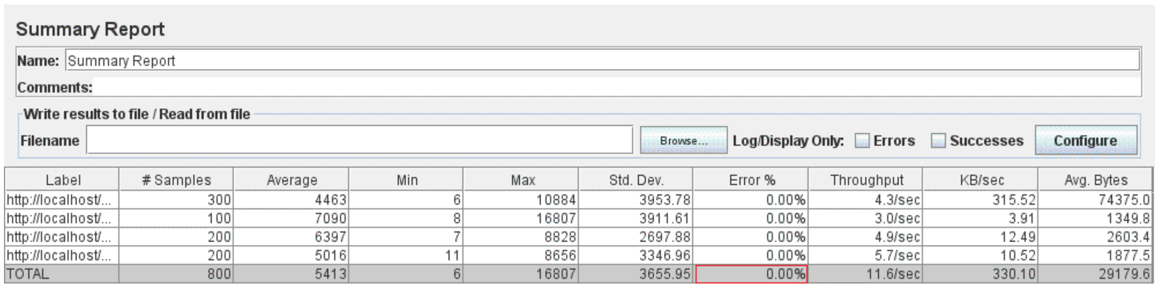
Therefore, before implementing the database index, for a number of **100** concurrent threads, the reported percentage error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **0%**. Figure 4.19 below illustrates the output obtained from the Summary Report listener.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	300	7096	10	17010	5238.04	0.00%	3.5/sec	254.64	74375.2
http://localhost/...	100	8479	6036	16724	2393.43	0.00%	2.9/sec	5.89	2049.5
http://localhost/...	200	8219	342	16230	3099.58	0.00%	4.7/sec	13.62	2976.4
http://localhost/...	200	7409	43	9074	2177.41	0.00%	5.9/sec	13.90	2426.1
TOTAL	800	7628	10	17010	3858.50	0.00%	9.3/sec	269.31	29497.5

Figure 4.19: Summary Report listener – 100 threads – before adding index

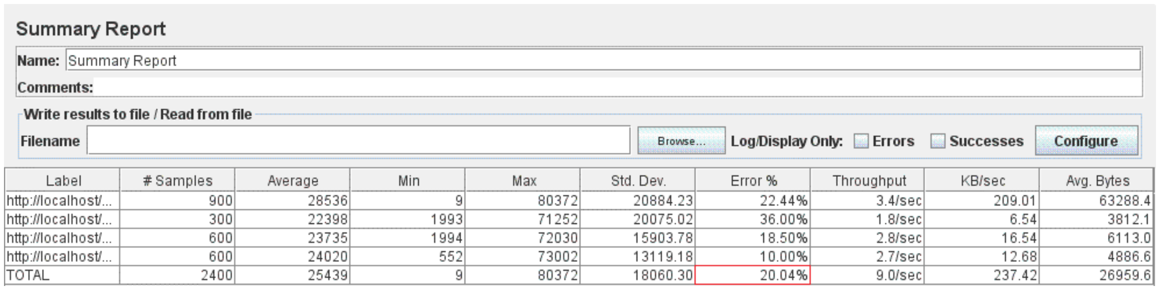
After the index addition, this **0%** error was preserved, therefore no performance downgrade occurred (Figure 4.20).



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	300	4463	6	10884	3953.78	0.00%	4.3/sec	315.52	74375.0
http://localhost/...	100	7090	8	16807	3911.61	0.00%	3.0/sec	3.91	1349.8
http://localhost/...	200	6397	7	8828	2697.88	0.00%	4.9/sec	12.49	2603.4
http://localhost/...	200	5016	11	8656	3346.96	0.00%	5.7/sec	10.52	1877.5
TOTAL	800	5413	6	16807	3655.95	0.00%	11.6/sec	330.10	29179.6

Figure 4.20: Summary Report listener – 100 threads – after adding index

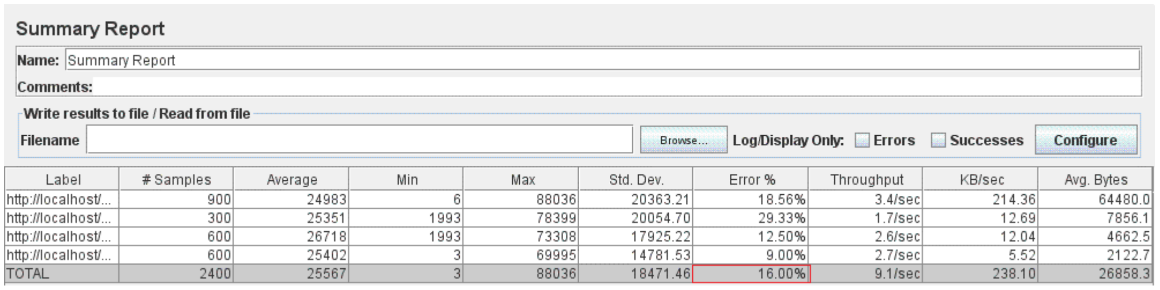
For a number of **300** concurrent users, before implementing the database index, the reported percentage error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **20.04**%. Figure 4.21 presents the output obtained from JMeter's Summary Report listener.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	900	28536	9	80372	20884.23	22.44%	3.4/sec	209.01	63288.4
http://localhost/...	300	22398	1993	71252	20075.02	36.00%	1.8/sec	6.54	3812.1
http://localhost/...	600	23735	1994	72030	15903.78	18.50%	2.8/sec	16.54	6113.0
http://localhost/...	600	24020	552	73002	13119.18	10.00%	2.7/sec	12.68	4886.6
TOTAL	2400	25439	9	80372	18060.30	20.04%	9.0/sec	237.42	26959.6

Figure 4.21: Summary Report listener – 300 threads – before adding index

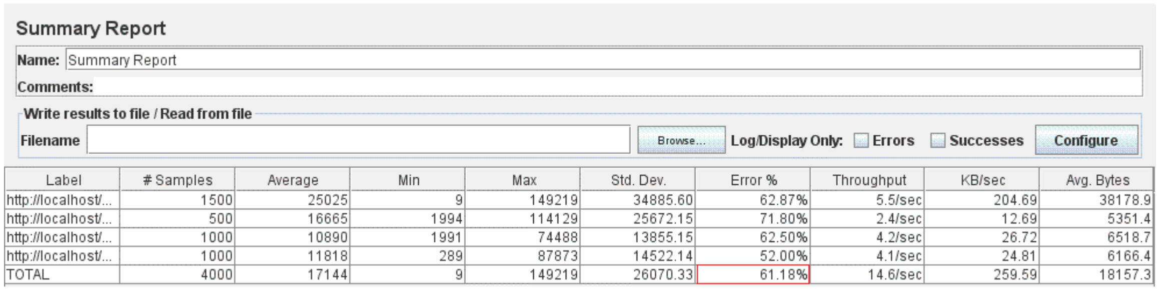
The newly obtained error percentage after implementing the database index for the *username* attribute of the *User* entity model decreased to **16.00%** (Figure 4.22).



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	900	24983	6	88036	20363.21	18.56%	3.4/sec	214.36	64480.0
http://localhost/...	300	25351	1993	78399	20054.70	29.33%	1.7/sec	12.69	7856.1
http://localhost/...	600	26718	1993	73308	17925.22	12.50%	2.6/sec	12.04	4662.5
http://localhost/...	600	25402	3	69995	14781.53	9.00%	2.7/sec	5.52	2122.7
TOTAL	2400	25567	3	88036	18471.46	16.00%	9.1/sec	238.10	26858.3

Figure 4.22: Summary Report listener – 300 threads – after adding index

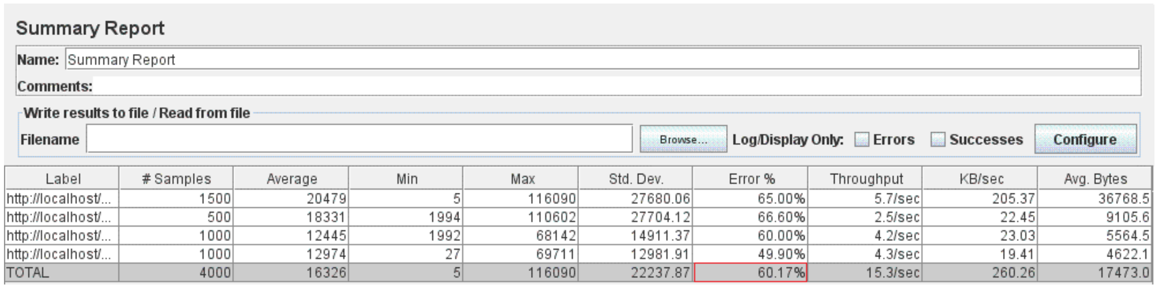
For a number of **500** concurrent users, before adding the database index, the reported error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **61.18**%. Figure 4.23 listed below illustrates the results obtained from JMeter's Summary Report listener.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	1500	25025	9	149219	34885.60	62.87%	5.5/sec	204.69	38178.9
http://localhost/...	500	16665	1994	114129	25672.15	71.80%	2.4/sec	12.69	5351.4
http://localhost/...	1000	10890	1991	74488	13855.15	62.50%	4.2/sec	26.72	6518.7
http://localhost/...	1000	11818	289	87873	14522.14	52.00%	4.1/sec	24.81	6166.4
TOTAL	4000	17144	9	149219	26070.33	61.18%	14.6/sec	259.59	18157.3

Figure 4.23: Summary Report listener – 500 threads – before adding index

As a result of adding the database index, the error percentage for the testcase execution with **500** concurrent threads decreased to **60.17%** (Figure 4.24).



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	1500	20479	5	116090	27680.06	65.00%	5.7/sec	205.37	36768.5
http://localhost/...	500	18331	1994	110602	27704.12	66.60%	2.5/sec	22.45	9105.6
http://localhost/...	1000	12445	1992	68142	14911.37	60.00%	4.2/sec	23.03	5564.5
http://localhost/...	1000	12974	27	69711	12981.91	49.90%	4.3/sec	19.41	4622.1
TOTAL	4000	16326	5	116090	22237.87	60.17%	15.3/sec	260.26	17473.0

Figure 4.24: Summary Report listener – 500 threads – after adding index

The conclusions gathered from the previous investigations are summarized below:

- For 100 concurrent users executing the considered testcase, the satisfactory result of 0% error is preserved after adding the database index.
- In case of 300 concurrent threads executing the considered testcase, the initially obtained 20.04% error is reduced to 16.00% after implementing the database index for the *username* attribute of the *User* entity model.
- For 500 users that simultaneously execute the sample testcase, adding the database index determined a decrease in the percentage error from 61.18% to 60.17%.

Consequently, by implementing the database index for the *username* attribute of the *User* entity model, the system's performance has been improved or at least preserved in all of the three cases (for 100, 300 and 500 concurrent threads). Even if positive results were obtained after this first solution, of developing a database index for the *username* attribute of the *User* entity model, the investigations for removing the database bottleneck proceed with alternative approaches that might lead to better performance enhancements.

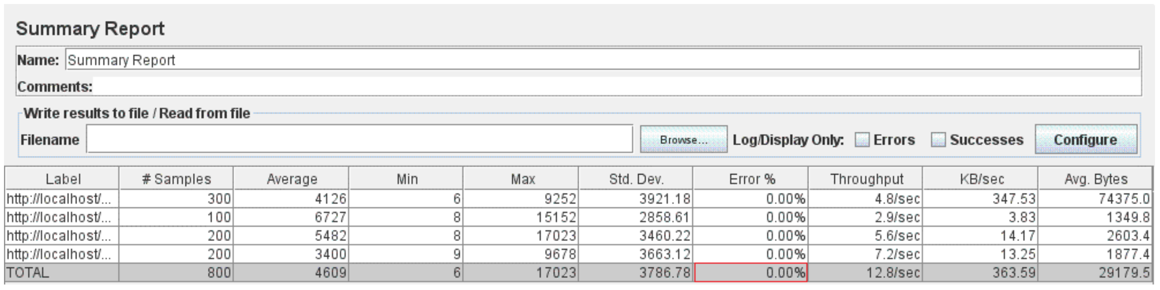
As a result, the next step was to reanalyze the list of SQL queries reported as being slow and select another approach that might be higher scored than the previous one in terms of performance improvement. Therefore, as the initially selected slow query was searching the database for a user based on username and password, the subsequent solution consists of removing the previously implemented index for the username attribute and adding a new one for the *password* attribute of the *User* entity. Finally, the same JMeter performance tests offered a precise resolution regarding the suitability of this solution.

Consequently, the index for the username attribute was removed, and another one for the *password* attribute of the *User* entity was added from Java code, as illustrated below:

```
@NotEmpty
@Column(name="password")
@Index(name = "passwordIDX")
private String password;
```

After the addition of this database index, the corresponding SQL query was not logged anymore as being slow, fact which denotes a performance enhancement brought by this approach. The same behavior was encountered in the previous solution, when the *username* attribute was indexed. Moreover, by running JMeter performance tests, a precise performance analysis might be conducted in order to establish whether this solution brought any improvement. Similar to the previous case, the performance tests were executed for 100, 300 and 500 concurrent users.

Therefore, before adding the database index, for a number of **100** concurrent threads, the reported error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **0%**. The output obtained from the Summary Report listener has already been illustrated in Figure 4.19. Fortunately, after the index addition, the **0%** error was preserved, therefore no performance downgrade occurred (Figure 4.25).


Summary Report

Name: Summary Report

Comments:

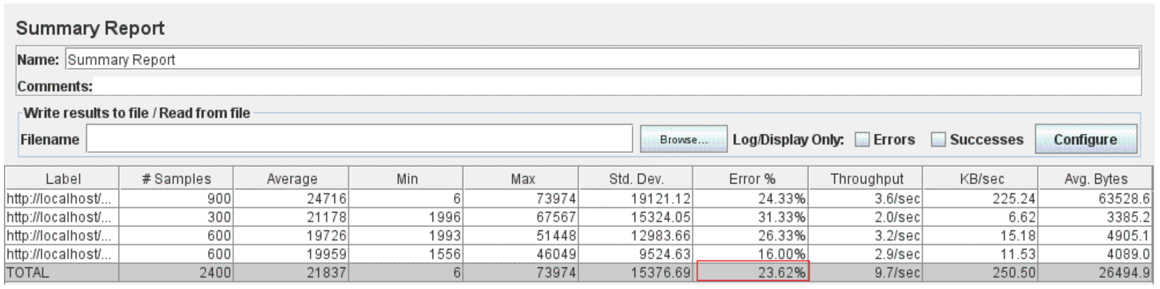
Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	300	4126	6	9252	3921.18	0.00%	4.8/sec	347.53	74375.0
http://localhost/...	100	6727	8	15152	2858.61	0.00%	2.9/sec	3.83	1349.8
http://localhost/...	200	5482	8	17023	3460.22	0.00%	5.6/sec	14.17	2603.4
http://localhost/...	200	3400	9	9678	3663.12	0.00%	7.2/sec	13.25	1877.4
TOTAL	800	4609	6	17023	3786.78	0.00%	12.8/sec	363.59	29179.5

Figure 4.25: Summary Report listener – 100 threads – after adding index

For a number of **300** concurrent users, before implementing any database index, the reported percentage error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **20.04**%. The result from JMeter's Summary Report listener was already presented in Figure 4.21. Surprisingly, the newly obtained error percentage after implementing the database index for the *password* attribute of the *User* entity model increased to **23.62%** (Figure 4.26).


Summary Report

Name: Summary Report

Comments:

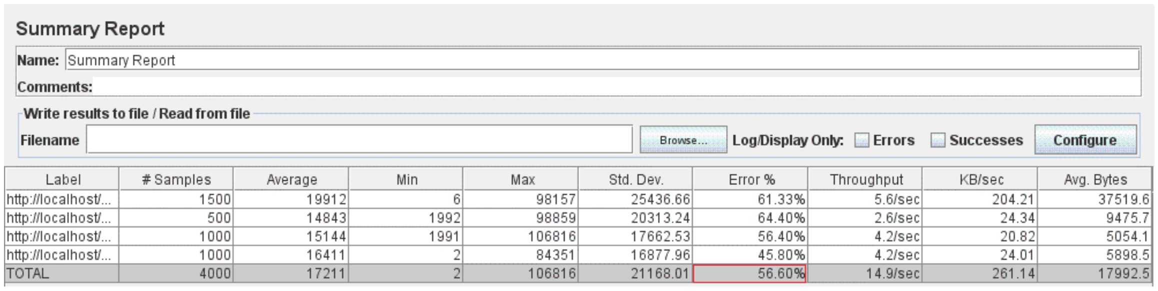
Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	900	24716	6	73974	19121.12	24.33%	3.6/sec	225.24	63528.6
http://localhost/...	300	21178	1996	67567	15324.05	31.33%	2.0/sec	6.62	3385.2
http://localhost/...	600	19726	1993	51448	12983.66	26.33%	3.2/sec	15.18	4905.1
http://localhost/...	600	19959	1556	46049	9524.63	16.00%	2.9/sec	11.53	4089.0
TOTAL	2400	21837	6	73974	15376.69	23.62%	9.7/sec	250.50	26494.9

Figure 4.26: Summary Report listener – 300 threads – after adding index

For a number of **500** concurrent users, before adding the database index, the reported error percentage corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **61.18**%. Figure 4.23 previously presented illustrates the results obtained from JMeter's Summary Report listener. As a result of adding the database index, the error percentage for the testcase execution with **500** concurrent threads decreased to **56.60%** (Figure 4.27).



Name: Summary Report
Comments:
Write results to file / Read from file
Filename
Log/Display Only: ☐ Errors ☐ Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	1500	19912	6	98157	25436.66	61.33%	5.6/sec	204.21	37519.6
http://localhost/...	500	14843	1992	98859	20313.24	64.40%	2.6/sec	24.34	9475.7
http://localhost/...	1000	15144	1991	106816	17662.53	56.40%	4.2/sec	20.82	5054.1
http://localhost/...	1000	16411	2	84351	16877.96	45.80%	4.2/sec	24.01	5898.5
TOTAL	4000	17211	2	106816	21168.01	56.60%	14.9/sec	261.14	17992.5

Figure 4.27: Summary Report listener – 500 threads – after adding index

The conclusions gathered from the previous investigations are summarized as follows:

- For 100 concurrent users executing the considered testcase, the satisfactory result of 0% error is preserved after adding the database index for the *password* attribute.
- In case of 300 concurrent threads executing the considered testcase, the initially obtained 20.04% error is increased to 23.62% after implementing the database index.
- For 500 users that simultaneously execute the sample testcase, adding the database index for the *password* attribute of the *User* entity model determined a decrease in the percentage error from 61.18% to 56.60%.

Consequently, the implemented database index for the *password* attribute exhibits different behavior depending on the number of concurrent users. More precisely, a performance improvement might be denoted for 100 and 500 threads executing the considered testcase, while for 300 concurrent users the system's performance is negatively affected by the added database index.

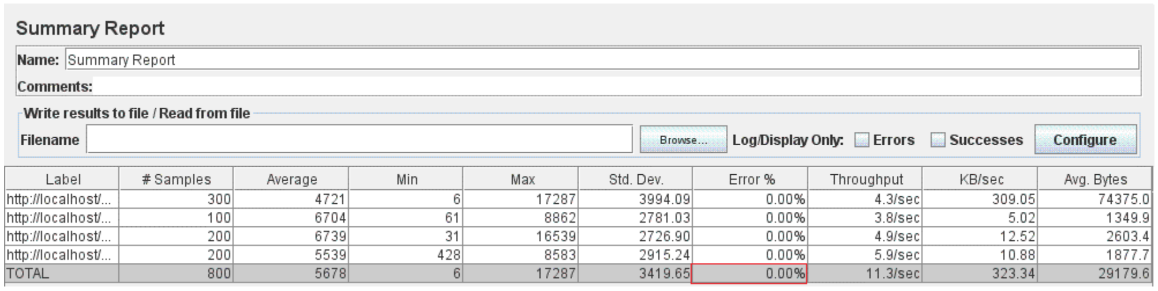
Despite of the overall positive performance results obtained in the two cases of database indexing, the conducted research did not complete, as several other approaches were considered valuable. As a consequence, the following approach was to preserve the last index added for the *password* attribute and recreate the one initially added for the *username* attribute of the *User* entity. Afterwards, same JMeter measurements were executed in order to assess whether these two separate database indexes added for the attributes used in the *WHERE* clause of the SQL query (for searching the user in the database) brought significant performance enhancements.

Therefore, the index for the *password* attribute of the *User* entity was preserved and the one corresponding to the *username* attribute was recreated (through Java code, as well):

```
@NotEmpty
@Column(name="username")
@Index(name = "usernameIDX")
private String username;

@NotEmpty
@Column(name="password")
@Index(name = "passwordIDX")
private String password;
```

After the addition of these two database indexes, the corresponding SQL query was not logged anymore as being slow, fact which denotes a performance enhancement brought by the approach of adding two separate indexes, for the *username* and *password* attributes of the User entity. The same behavior was encountered in the previous cases, when the two attributes were indexed one at a time, initially the *username* attribute and afterwards the *password* one. Moreover, by running JMeter performance tests, an accurate comparison might be conducted among the performance obtained when using only one index and the one reported when adding two separate database indexes. Similar to the previous cases, JMeter performance tests were executed for 100, 300 and 500 concurrent threads.

Therefore, after the two indexes addition, for a number of **100** concurrent threads, the reported error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was also **0%**, the same as in the previous cases: when no index was added (Figure 4.19.) or when only one index was created (Figures 4.20 and Figure 4.25). Figure 4.28 below illustrates the results obtained from JMeter's Summary Report listener.


Summary Report

Name: Summary Report

Comments:

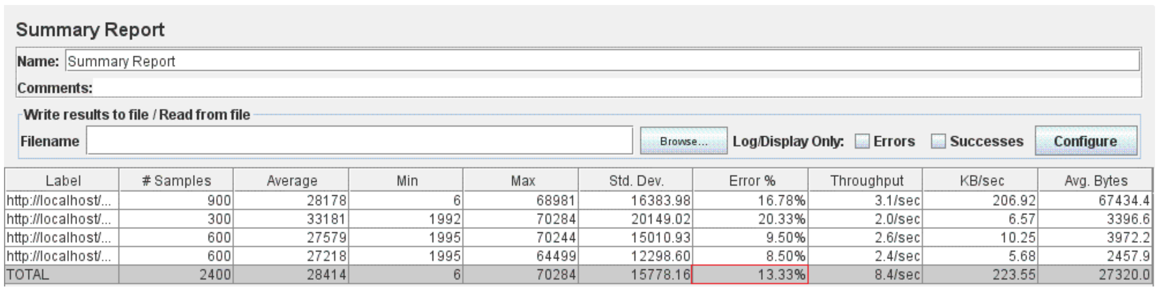
Write results to file / Read from file

Filename Log/Display Only: Errors Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	300	4721	6	17287	3994.09	0.00%	4.3/sec	309.05	74375.0
http://localhost/...	100	6704	61	8862	2781.03	0.00%	3.8/sec	5.02	1349.9
http://localhost/...	200	6739	31	16539	2726.90	0.00%	4.9/sec	12.52	2603.4
http://localhost/...	200	5539	428	8583	2915.24	0.00%	5.9/sec	10.88	1877.7
TOTAL	800	5678	6	17287	3419.65	0.00%	11.3/sec	323.34	29179.6

Figure 4.28: Summary Report listener – 100 threads – after adding index

For a number of **300** concurrent users, after adding the two database indexes, the reported percentage error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **13.33%**. The result from JMeter's Summary Report listener is presented in Figure 4.29.


Summary Report

Name: Summary Report

Comments:

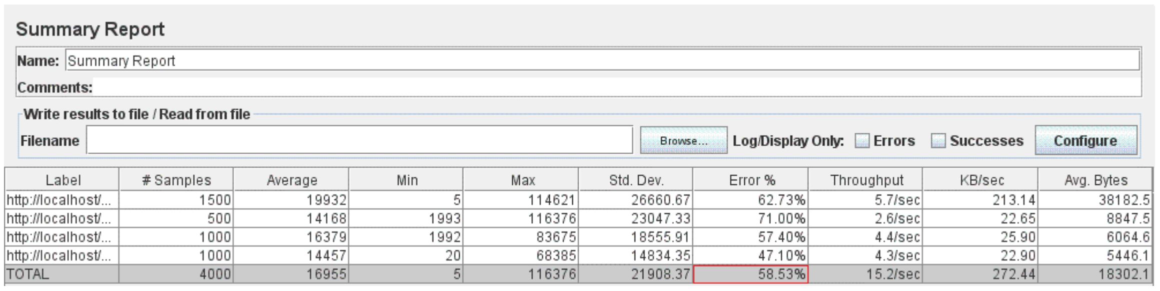
Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	900	28178	6	68981	16383.98	16.78%	3.1/sec	206.92	67434.4
http://localhost/...	300	33181	1992	70284	20149.02	20.33%	2.0/sec	6.57	3396.6
http://localhost/...	600	27579	1995	70244	15010.93	9.50%	2.6/sec	10.25	3972.2
http://localhost/...	600	27218	1995	64499	12298.60	8.50%	2.4/sec	5.68	2457.9
TOTAL	2400	28414	6	70284	15778.16	13.33%	8.4/sec	223.55	27320.0

Figure 4.29: Summary Report listener – 300 threads – after adding index

After adding the two database indexes, the reported error corresponding to the execution of **Login_Skill Upgrades for developer_Logout** testcase for a number of **500** threads, was of **58.53%**. The output obtained from JMeter's Summary Report listener is presented in Figure 4.30 below.


Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	1500	19932	5	114621	26660.67	62.73%	5.7/sec	213.14	38182.5
http://localhost/...	500	14168	1993	116376	23047.33	71.00%	2.6/sec	22.65	8847.5
http://localhost/...	1000	16379	1992	83675	18555.91	57.40%	4.4/sec	25.90	6064.6
http://localhost/...	1000	14457	20	68385	14834.35	47.10%	4.3/sec	22.90	5446.1
TOTAL	4000	16955	5	116376	21908.37	58.53%	15.2/sec	272.44	18302.1

Figure 4.30: Summary Report listener – 500 threads – after adding index

The conclusions gathered from the previous investigations are summarized as follows:

- For 100 concurrent users executing the considered testcase, the successful result of 0% error is preserved after adding the two separate database indexes for the *username* and *password* attributes.
- In case of 300 concurrent threads executing the considered testcase, after adding the two database indexes, the error is decreased to 13.33%.
- For 500 users that simultaneously execute the sample testcase, after adding two separate database indexes for the *username* and *password* attributes of the *User* entity model an error of 58.53% was obtained.

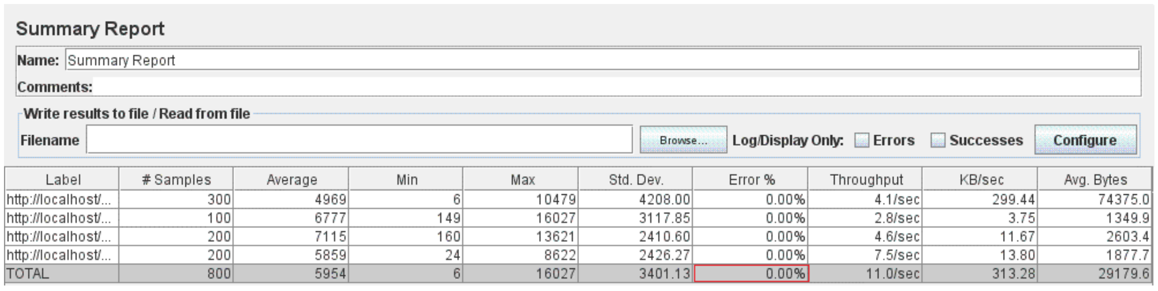
Consequently, the two database indexes added for the *username* and *password* attributes brought a performance improvement in all of the three situations (for 100, 300 and 500 concurrent users) as compared to the initial case when no index was used and to the ones when a single database index was created.

The research conducted towards removing the database bottleneck completes with the solution of adding a composite database index. More precisely, the last indexes added were removed and a composite one, made of both attributes (*username* and *password*) of the *User* entity, was created. The composite index was added to the Java entity class it belongs to, i.e. *User*:

```
@Entity
@Table(name="userList",
uniqueConstraints=@UniqueConstraint(columnNames={"username", "password"}))
public class User {
        ...
        @NotEmpty
        @Column(name="username")
        private String username;

        @NotEmpty
        @Column(name="password")
        private String password;

    ...

}
```

After the addition of the composite database index, the corresponding SQL query was not logged anymore as being slow, fact which denotes a performance enhancement brought by this approach. The same behavior was encountered in all the previous cases when database indexes were added. Further on, by running JMeter performance tests, an accurate comparison might be conducted among the performance obtained for the current solution and for the previous ones. Also for this case, JMeter performance tests were executed for 100, 300 and 500 concurrent users.

Therefore, after the composite index addition, for a number of **100** concurrent threads, the reported error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was also **0%**, the same as in all previous cases: when no index was added (Figure 4.19.), when only one index was created (Figures 4.20 and Figure 4.25) or when two separate indexes were added (Figure 4.28). Figure 4.31 outlines the results obtained from JMeter's Summary Report listener.



Summary Report

Name: Summary Report

Comments:

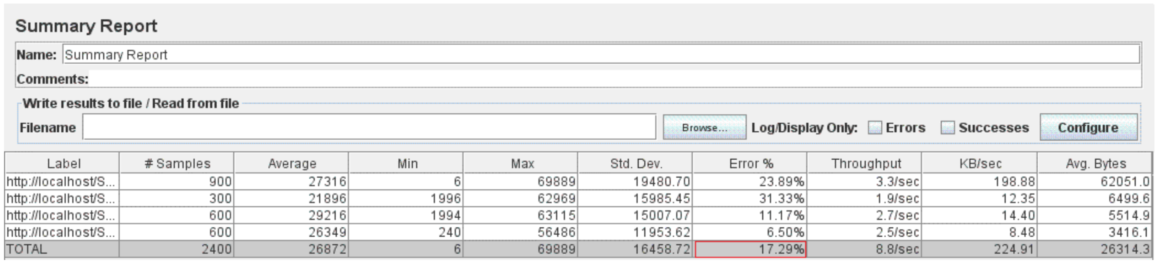
Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/...	300	4969	6	10479	4208.00	0.00%	4.1/sec	299.44	74375.0
http://localhost/...	100	6777	149	16027	3117.85	0.00%	2.8/sec	3.75	1349.9
http://localhost/...	200	7115	160	13621	2410.60	0.00%	4.6/sec	11.67	2603.4
http://localhost/...	200	5859	24	8622	2426.27	0.00%	7.5/sec	13.80	1877.7
TOTAL	800	5954	6	16027	3401.13	0.00%	11.0/sec	313.28	29179.6

Figure 4.31: Summary Report listener – 100 threads – after adding index

For a number of **300** concurrent users, after adding the composite database index, the reported percentage error corresponding to the **Login_Skill Upgrades for developer_Logout** testcase was **17.29%**. The result from JMeter's Summary Report listener is shown in Figure 4.32 below.



Summary Report

Name: Summary Report

Comments:

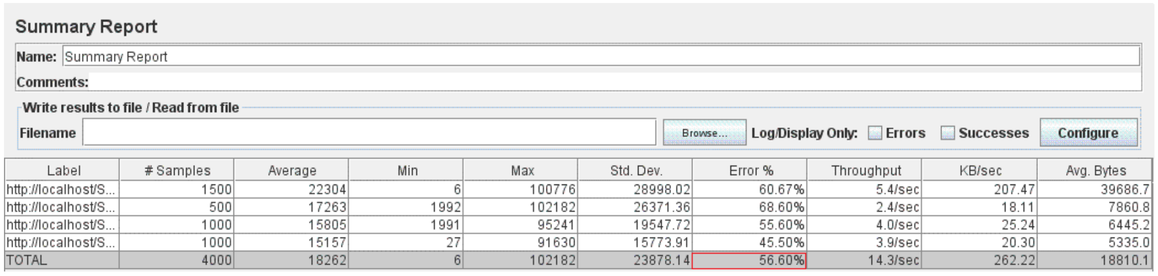
Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/S...	900	27316	6	69889	19480.70	23.89%	3.3/sec	198.88	62051.0
http://localhost/S...	300	21896	1996	62969	15985.45	31.33%	1.9/sec	12.35	6499.6
http://localhost/S...	600	29216	1994	63115	15007.07	11.17%	2.7/sec	14.40	5514.9
http://localhost/S...	600	26349	240	56486	11953.62	6.50%	2.5/sec	8.48	3416.1
TOTAL	2400	26872	6	69889	16458.72	17.29%	8.8/sec	224.91	26314.3

Figure 4.32: Summary Report listener – 300 threads – after adding index

After adding the composite indexes, the reported error corresponding to the execution of **Login_Skill Upgrades for developer_Logout** testcase for a number of **500** threads, was of **56.60%**. The output obtained from JMeter's Summary Report listener is illustrated in Figure 4.33.



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/S...	1500	22304	6	100776	28998.02	60.67%	5.4/sec	207.47	39686.7
http://localhost/S...	500	17263	1992	102182	26371.36	68.60%	2.4/sec	18.11	7860.8
http://localhost/S...	1000	15805	1991	95241	19547.72	55.60%	4.0/sec	25.24	6445.2
http://localhost/S...	1000	15157	27	91630	15773.91	45.50%	3.9/sec	20.30	5335.0
TOTAL	4000	18262	6	102182	23878.14	56.60%	14.3/sec	262.22	18810.1

Figure 4.33: Summary Report listener – 500 threads – after adding index

The conclusions resulted from the previous research are summarized below:

- For 100 concurrent users executing the considered testcase, the successful result of 0% error is preserved after adding the composite database index, consisting of the *username* and *password* attributes.

- In case of 300 concurrent threads executing the considered testcase, after adding the composite database index, the reported error is 17.29%.
- For 500 users that simultaneously execute the sample testcase, after adding the composite database index, consisting of the *username* and *password* attributes belonging the *User* entity model, an error of 56.60% was reported.

Consequently, the composite database index, consisting of the *username* and *password* attributes does not offer a performance improvement in all of the three situations (for 100, 300 and 500 concurrent users). More precisely, for a number of 300 concurrent workers, the solution of adding a single index for the *username* attribute and the one of creating two separate indexes (one for the *username* and another one for the *password* attribute of the *User* entity), offered higher scored results from system's performance viewpoint. For the other cases, the successful performance capabilities were preserved for 100 concurrent threads, and even enhanced, by reducing the percentage error, in case of 500 users.

Finally, the table below summarizes the results of JMeter performance tests, corresponding to all solutions implemented for removing the database bottleneck and having the error percentage as indicator:

	100 threads	**300 threads**	**500 threads**
No Index	0%	20.04%	61.18%
Index for *username*	0%	16.00%	60.17%
Index for *password*	0%	23.62%	56.60%
Index for *username* + Index for *password*	0%	13.33%	58.53%
Composite index from *username* and *password*	0%	17.29%	56.60%

Table 4.1: Summary of error values corresponding to all investigated database indexing configurations

As a result of the experimental study related to the methodology of determining the most suitable database indexing approach for persisted entities, it might be concluded that, in majority of the cases, indexing improves the application's performance capabilities. Even more, starting from the SQL queries reported as being slow, the recommendation is to add indexes for the attribute(s) that appear in the *WHERE* clause. In the case of several attributes used in results filtering (which appear in *WHERE* clause of the *SELECT* query), the performance is maximized when either creating a separate index for each attribute or only a single one, selected thought dedicated performance tests. Surprisingly, creating a composite index might bring overhead, while offering only a slightly enhanced performance as compared to using a separate index for each attribute or for a single attribute, whose selection is dependent on use case. However, the most appropriate database indexing approach has to be determined for each particular case through experimental tests.

As bottom line for the **Login_Skill Upgrades for developer_Logout** testcase, the performance experiments' results determined the usage of two separate indexes (one for the *username* and another one for the *password* attribute of the *User* entity) which will maximize the overall use case performance.

Another use case scenario relevant for the database indexing scope is the one already presented in sub-chapter 4.4 and referred as **Login_Add New Developer_Delete Old Developer_Logout** testcase. Its use case flow through the application consists of the following actions: login as a Scrum Master user, add a new developer, delete an old developer (from the team/project managed by the Scrum Master) and logout. Similar to the previous use case, the same approach to determine the most suitable database indexing scheme was followed.

The first step was to analyze the SQL queries logged as being slow and, therefore, the ***SELECT * FROM Developer d WHERE d.firstName = '… ' and d.lastName ='…';*** result was obtained.

By taking into account the results obtained for the **Login_Skill Upgrades for developer_Logout** testcase, where the results filtering was also performed after two attributes (*username* and *password*), the same database indexing approach was selected. Consequently, in case of the **Login_Add New Developer_Delete Old Developer_Logout** testcase, two separate indexes were created for the attributes used in results filtering (which appear in *WHERE* clause of the *SELECT* query) corresponding to the *Developer* entity model, namely *firstName* and *lastName*. This was performed from Java code as follows:

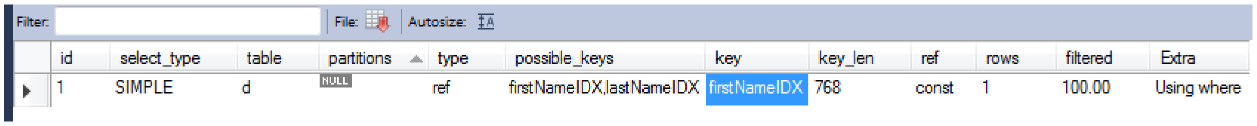
```
@Column(name="firstName")
@Index(name = "firstNameIDX")
private String firstName;

@Column(name="lastName")
@Index(name = "lastNameIDX")
private String lastName;
```

The performance brought by implementing this database indexing approach was denoted by analyzing the MySQL slow queries log. More precisely, before adding the indexes, the number of occurrences of ***SELECT * FROM Developer d WHERE d.firstName = '… ' and d.lastName ='…';*** query inside the slow queries log file was 240. After adding the database indexes and executing the use case under the same conditions, this query was not included anymore in the slow queries log.

Furthermore, in order to assess the indexes usage accuracy when executing the SQL query, the following command was run from MySQL Workbench tool:

EXPLAIN EXTENDED SELECT * FROM Developer d WHERE d.firstName = '…' and d.lastName = '…';.

The result, obtained in a table format, clearly denotes the correct usage of the two database indexes in the *SELECT* clause:



Filter: File: Autosize:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	d	NULL	ref	firstNameIDX,lastNameIDX	firstNameIDX	768	const	1	100.00	Using where

Figure 4.34: MySQL output for the ***EXPLAIN EXTENDED SELECT*** query

As final conclusion of this sub-chapter, different database indexes approaches were detailed followed by dedicated performance experiments intended to determine their suitability for each particular use case scenario. Moreover, a summary of the conclusions gathered was presented and some general recommendations formulated.

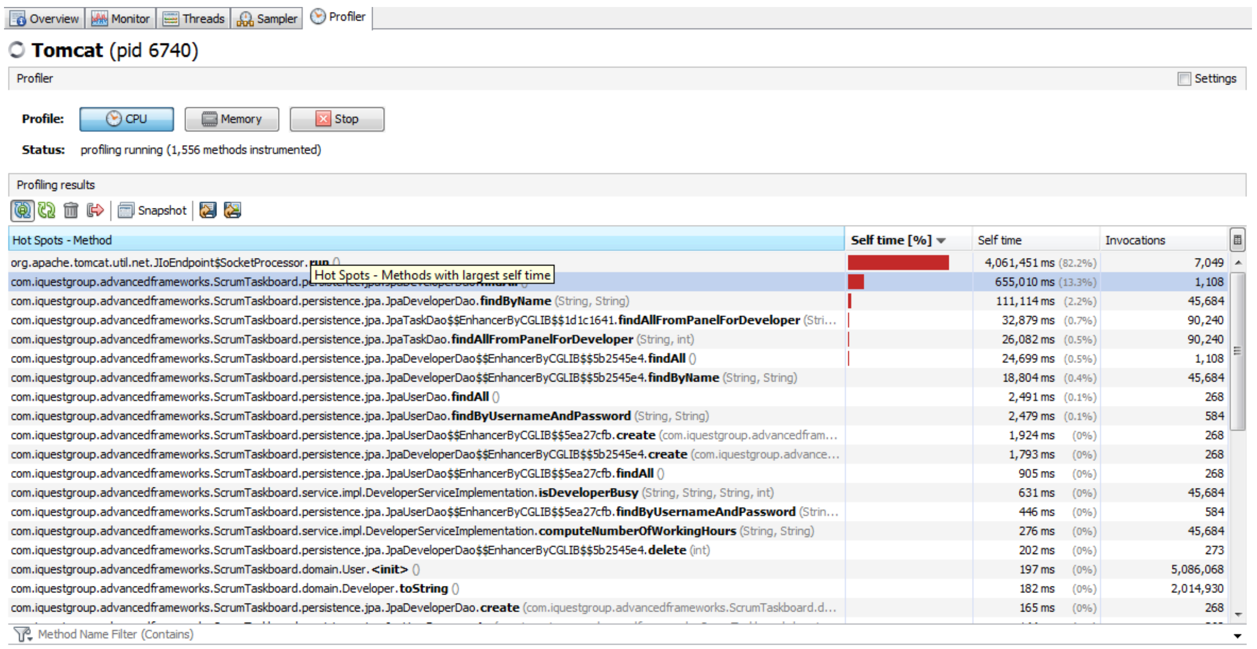
The following sub-chapter focuses on studying and developing caching using **Spring Eh-Cache** and assessing the performance enhancements gathered.

4.7. Implementing caching using Spring Eh-Cache and monitoring the performance enhancements obtained

The next step in the dissertation thesis research is to add caching for the web application under test and afterwards evaluate the performance improvements obtained. Two caching methodologies were considered for the scope of the dissertation thesis research, Spring Eh-Cache and Hibernate Eh-Cache. This sub-chapters focuses on briefly presenting the applicability of Spring Eh-Cache on the system and presents the experimental results obtained from performance tests execution.

Therefore, the initial phase of the approach followed was to gather insights regarding Spring Eh-Cache's operation and features. Following this phase, came the effective development of the established Spring Eh-Cache solution. Finally, the overall performance enhancement brought by Spring Eh-Cache addition is evaluated through dedicated experiments and results analysis concludes its suitability for our particular case scenario.

The selected use case for the purpose of the current research (of adding caching using Spring Eh-Cache) was **Login_Add New Developer_Delete Old Developer_Logout**, whose performance was initially monitored from VisualVM's profiler and sampler tools (Figures 4.35 and 4.36).


Overview | Monitor | Threads | Sampler | Profiler

Tomcat (pid 6740)

Profiler

Settings

Profile: CPU | Memory | Stop

Status: profiling running (1,556 methods instrumented)

Profiling results

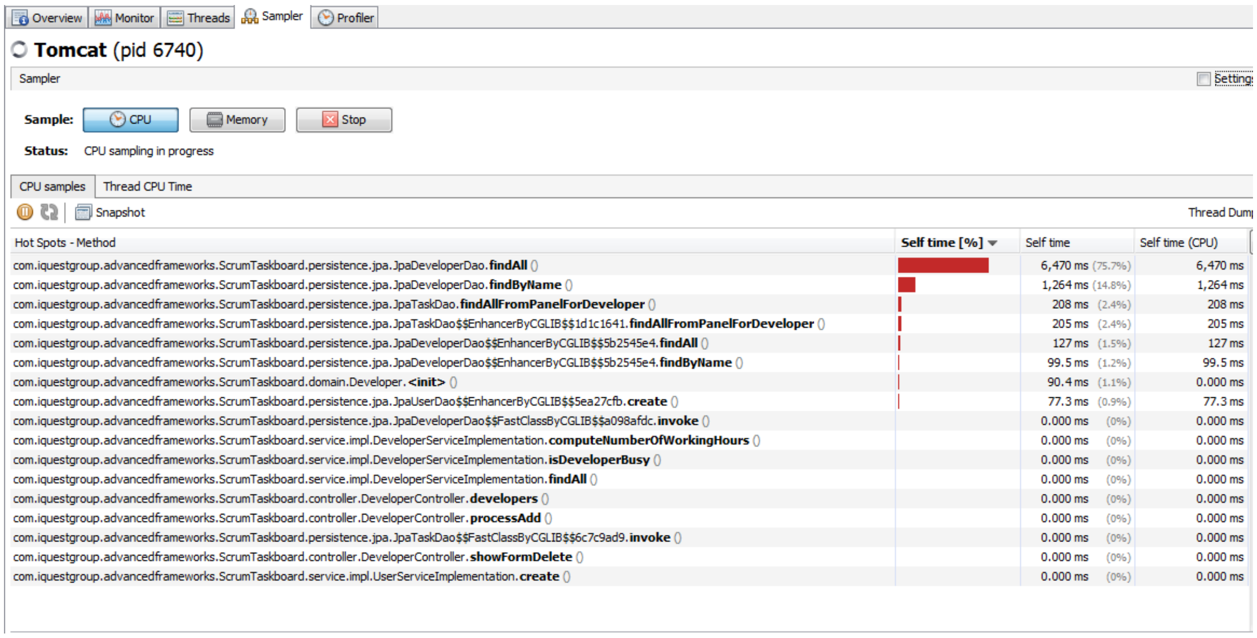
Snapshot

Hot Spots - Methods with largest self time

Hot Spots - Method	Self time [%]	Self time	Invocations
org.apache.tomcat.util.net.JIoEndpoint$SocketProcessor.**run** ()	🟥	4,061,451 ms (82.2%)	7,049
com.iquestgroup.advancedframeworks.ScrumTaskboard.p	🟥	655,010 ms (13.3%)	1,108
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**findByName** (String, String)	🟥	111,114 ms (2.2%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao$$EnhancerByCGLIB$$1d1c1641.**findAllFromPanelForDeveloper** (Stri...	🟥	32,879 ms (0.7%)	90,240
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao.**findAllFromPanelForDeveloper** (String, int)	🟥	26,082 ms (0.5%)	90,240
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findAll** ()	🟥	24,699 ms (0.5%)	1,108
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findByName** (String, String)		18,804 ms (0.4%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao.**findAll** ()		2,491 ms (0.1%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao.**findByUsernameAndPassword** (String, String)		2,479 ms (0.1%)	584
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**create** (com.iquestgroup.advancedfram...		1,924 ms (0%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**create** (com.iquestgroup.advance...		1,793 ms (0%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**findAll** ()		905 ms (0%)	268
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**isDeveloperBusy** (String, String, String, int)		631 ms (0%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**findByUsernameAndPassword** (Strin...		446 ms (0%)	584
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**computeNumberOfWorkingHours** (String, String)		276 ms (0%)	45,684
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**delete** (int)		202 ms (0%)	273
com.iquestgroup.advancedframeworks.ScrumTaskboard.domain.User.**<init>** ()		197 ms (0%)	5,086,068
com.iquestgroup.advancedframeworks.ScrumTaskboard.domain.Developer.**toString** ()		182 ms (0%)	2,014,930
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**create** (com.iquestgroup.advancedframeworks.ScrumTaskboard.d...		165 ms (0%)	268

Method Name Filter (Contains)

Figure 4.35: CPU usage information from VisualVM Profiler


Overview | Monitor | Threads | Sampler | Profiler

Tomcat (pid 6740)

Sampler

Settings

Sample: CPU | Memory | Stop

Status: CPU sampling in progress

CPU samples | Thread CPU Time

Snapshot

Thread Dump

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**findAll** ()	🟥	6,470 ms (75.7%)	6,470 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao.**findByName** ()	🟥	1,264 ms (14.8%)	1,264 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao.**findAllFromPanelForDeveloper** ()	🟥	208 ms (2.4%)	208 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao$$EnhancerByCGLIB$$1d1c1641.**findAllFromPanelForDeveloper** ()	🟥	205 ms (2.4%)	205 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findAll** ()	🟥	127 ms (1.5%)	127 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$EnhancerByCGLIB$$5b2545e4.**findByName** ()	🟥	99.5 ms (1.2%)	99.5 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.domain.Developer.**<init>** ()	🟥	90.4 ms (1.1%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaUserDao$$EnhancerByCGLIB$$5ea27cfb.**create** ()	🟥	77.3 ms (0.9%)	77.3 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaDeveloperDao$$FastClassByCGLIB$$a098afdc.**invoke** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**computeNumberOfWorkingHours** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**isDeveloperBusy** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.DeveloperServiceImplementation.**findAll** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.controller.DeveloperController.**developers** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.controller.DeveloperController.**processAdd** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.persistence.jpa.JpaTaskDao$$FastClassByCGLIB$$6c7c9ad9.**invoke** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.controller.DeveloperController.**showFormDelete** ()		0.000 ms (0%)	0.000 ms
com.iquestgroup.advancedframeworks.ScrumTaskboard.service.impl.UserServiceImplementation.**create** ()		0.000 ms (0%)	0.000 ms

Figure 4.36: CPU usage information from VisualVM Sampler

Despite of the considered use case, monitoring the application performance, through VisualVM profiling/sampling activities, provided precise information regarding the roots of system's CPU bottleneck. More precisely, by analyzing VisualVM's results, it might be determined that the overall system performance is affected by methods from the persistence layer (when retrieving results from the database). As this result indicates that database improvements are required at persistence level, these concerns were investigated in detail throughout the previous sub-chapter and corresponding measures undertaken.

Therefore, after the appropriate improvements for the persistence layer have been implemented, the current step considers the development of caching for the most CPU consuming functionality. In case of the sample use case scenario, referred as **Login_Add New Developer_Delete Old Developer_Logout**, Figures 4.35 and 4.36 indicate that most of the CPU time is spent on *JpaDeveloperDao.findAll ()*. As a consequence, Spring Eh-Cache was configured and implemented for the service procedure which calls this persistence-layer method, namely *DeveloperServiceImplementation.findAll()*. The main steps involved in developing this Spring Eh-Cache solution are:

- Add Spring Eh-Cache required dependencies in ***pom.xml*** file:

```
<properties>
        <!-- Eh-cache  -->
        <ehcache.version>2.10.0</ehcache.version>
</properties>

<dependencies>
        <dependency>
                <groupId>net.sf.ehcache</groupId>
                <artifactId>ehcache</artifactId>
                <version>${ehcache.version}</version>
        </dependency>

        <!-- Spring caching framework inside this -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>${spring-framework.version}</version>
        </dependency>

        <!-- Support for Ehcache and others -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context-support</artifactId>
                <version>${spring-framework.version}</version>
        </dependency>
</dependencies>
```

- Create ***ehcache.xml*** file under project's *resources* directory in order to tell Eh-cache how and where to cache the data:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
        updateCheck="true"
        monitoring="autodetect"
        dynamicConfig="true">

        <cache name="findAllDevelopersCache"
                maxEntriesLocalHeap="1000"
                maxEntriesLocalDisk="1000"
                eternal="false"
                timeToIdleSeconds="300" timeToLiveSeconds="600"
                memoryStoreEvictionPolicy="LFU"
                transactionalMode="off">
                <persistence strategy="localTempSwap" />
        </cache>
</ehcache>
```

- Add ***@Cacheable*** annotation on the method you want to cache (***findAll()*** from ***DeveloperServiceImplementation.java*** class) :

```
        @Cacheable("findAllDevelopersCache")
        public List<Developer> findAll() {
                ...
        }
```

- Configure and enable caching inside Spring’s configuration file (***application-config.xml***) :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:p="http://www.springframework.org/schema/p"
        ...
        xmlns:cache="http://www.springframework.org/schema/cache"
        xsi:schemaLocation="...
 http://www.springframework.org/schema/cache
 http://www.springframework.org/schema/cache/spring-cache.xsd ">
<cache:annotation-driven />

<bean                                                  id="cacheManager"
class="org.springframework.cache.ehcache.EhCacheCacheManager"
p:cacheManager-ref="ehcache" />
```

```
<bean id="ehcache"
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
p:configLocation="classpath:ehcache.xml" p:shared="true" />

</beans>
```

Another aspect that has to be taken into account when introducing caching to a web application concerns the possible data inconsistencies that might happen. More precisely, performing updates on the cached data might offer misleading information to the end user. For the particular Spring Eh-cache solution implemented for the *findAll()* method from ***DeveloperServiceImplementation.java*** class, in case the Scrum Master performs some modification on the list of developers (by adding or removing developers), data inconsistencies, and even errors, will occur on application usage:

- If new developers were added to the application, they won't be displayed in the web page until the cache has expired (after 10 minutes, as configured for this cache region in *ehcache.xml* file).
- If developers were removed from the application while they are still being cached, an error is thrown because the database search is performed with a NULL entity.

These critical issues were solved by manually invalidating the cache after performing updates on the developer list. This was performed explicitly through Java code by retrieving the specific cache (determined by its name) from a new *CacheManager* instance and invoking the *removeAll()* method:

```
private void invalidateCache(String cacheToClear) {
        CacheManager cacheManager = CacheManager.create();
        Ehcache ehcache = cacheManager.getEhcache(cacheToClear);
        ehcache.removeAll();
}
```

The cache invalidate functionality is invoked after operating (on successful addition or removal) on developers data, with the cache region's name to be invalidated given as parameter:

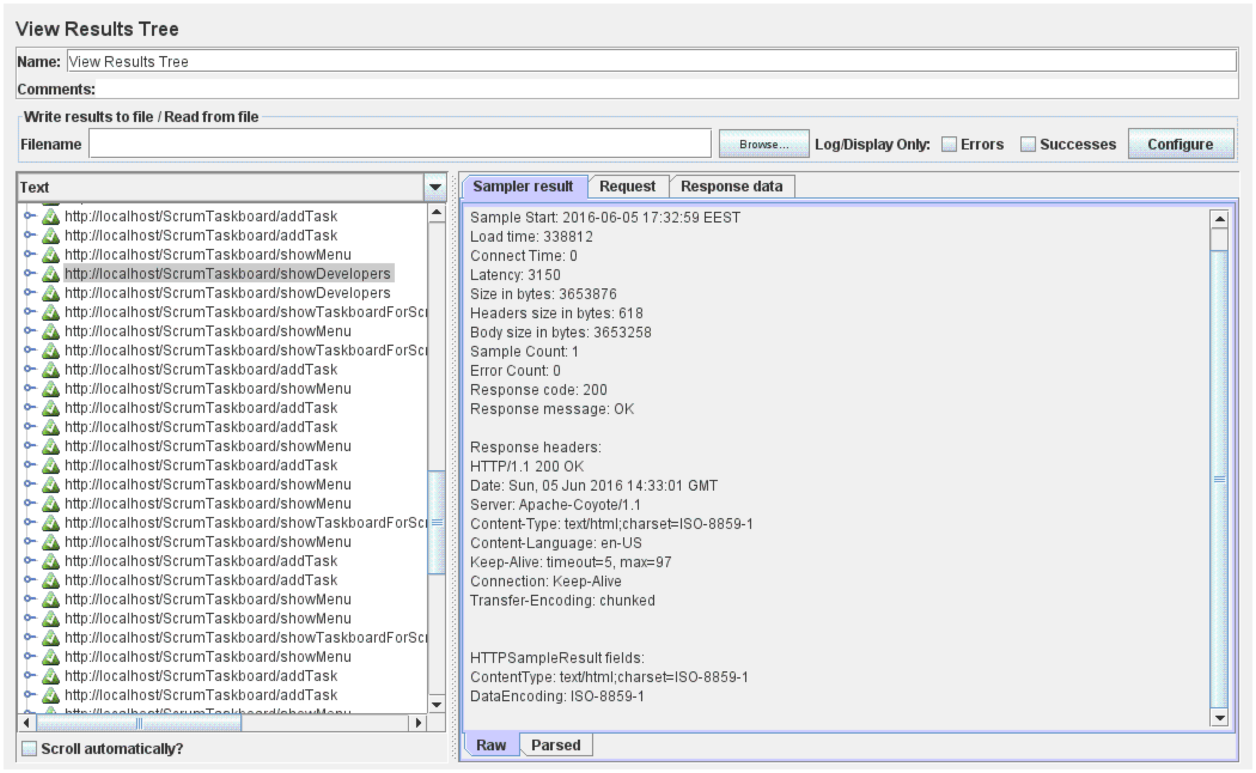
```
public void delete(int developerId) {

        try {

                ...  // perform developer removal

                invalidateCache("findAllDevelopersCache");

        } catch (...) {

                ...
        }
    }
```

The subsequent step after implementing the previously described Spring Eh-Cache solution is to assess the performance improvements obtained. In this context, the **Login_Add New Developer_Delete Old Developer_Logout** testcase is not a suitable case scenario for evaluating performance enhancements offered by Spring Eh-Cache, because it consists of actions that execute updates on the cached data (list of developers): additions and removals of developers from the database. More precisely, as we have implemented caching for the list of developers, executing this use case (or any other one that performs manipulations of the data being cached), would cause data inconsistencies and even errors (as described in a previous paragraph). Therefore, we may conclude this scenario is not suitable for testing performance improvements brought by implementing Spring Eh-Caching.

Oppositely, the system's performance capabilities have to be assessed on behalf of a use case scenario through which the developers are retrieved several times from the database. In this context, the usage of cached developers will take advantage of the Spring Eh-Cache implemented, fact validated through performance measurements. Consequently, the use case created for evaluating this concern is **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout**, which is performed by an administrator user (say Scrum Master) who logins into the application, visualizes all developers, then visualizes the taskboard, adds a new task (together with the required information for it), visits again the list of developers and the logs out from the application.

In order to evaluate the performance enhancements obtained, JMeter tool will be used to simulate the use case execution for many concurrent users retrieving the developers. The desired behavior for the implemented cache is to access the database only once (on first call) and afterwards retrieve the developers' information from the cache (for successive calls to find developers functionality).

The **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** testcase was executed from JMeter ***<u>before</u>*** implementing Spring Eh-Caching with the intent to collect actual performance measurements results. In order to see considerable performance improvements offered by the implemented Spring Eh-Cache solution, the data being cached (list of developers) has to be of large amount. Therefore, all performance tests were executed in the context of 1000 developers existent in the database. However, as this list is retrieved twice for each thread executing the selected use case scenario (**Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout**), the simulated number of users (threads) from JMeter cannot be very large (like hundreds or thousands) because this consumes all of the CPU's activity and takes a large amount of time to complete. As a result, the tested use case was executed in the context of 10 and 50 concurrent users and the output listeners' results were collected. However, it is not within the scope of this paper to present all these test results but only the ones providing essential information for the actual research. Consequently, for both cases of 10 and 50 users configured, *View Results Tree* and *Summary Report* listeners present information of interest for our investigation, so their output is presented in Figures 4.37 – 4.40 below.


View Results Tree

Name: View Results Tree

Comments:

Write results to file / Read from file

Filename

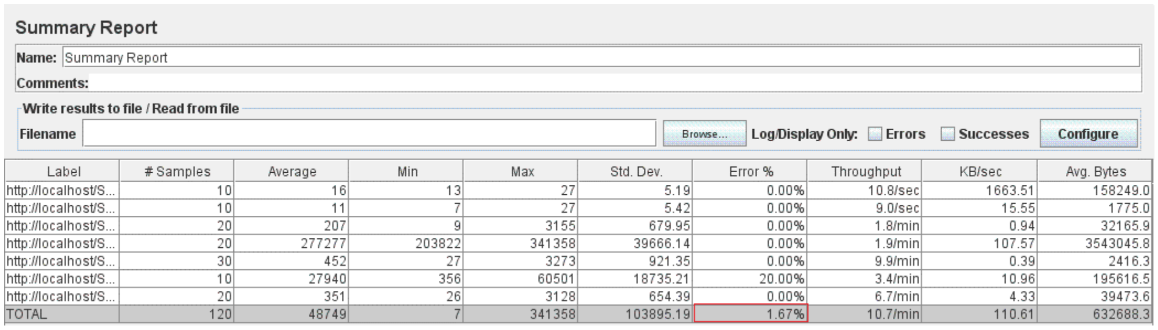
Log/Display Only: Errors Successes

```
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showTaskboardForScı
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScı
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScı
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScı
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
```

```
Sample Start: 2016-06-05 17:32:59 EEST
Load time: 338812
Connect Time: 0
Latency: 3150
Size in bytes: 3653876
Headers size in bytes: 618
Body size in bytes: 3653258
Sample Count: 1
Error Count: 0
Response code: 200
Response message: OK

Response headers:
HTTP/1.1 200 OK
Date: Sun, 05 Jun 2016 14:33:01 GMT
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=ISO-8859-1
Content-Language: en-US
Keep-Alive: timeout=5, max=97
Connection: Keep-Alive
Transfer-Encoding: chunked


HTTPSampleResult fields:
ContentType: text/html;charset=ISO-8859-1
DataEncoding: ISO-8859-1
```

Scroll automatically?

Figure 4.37: View Results Tree listener– 10 threads


Summary Report

Name: Summary Report

Comments:

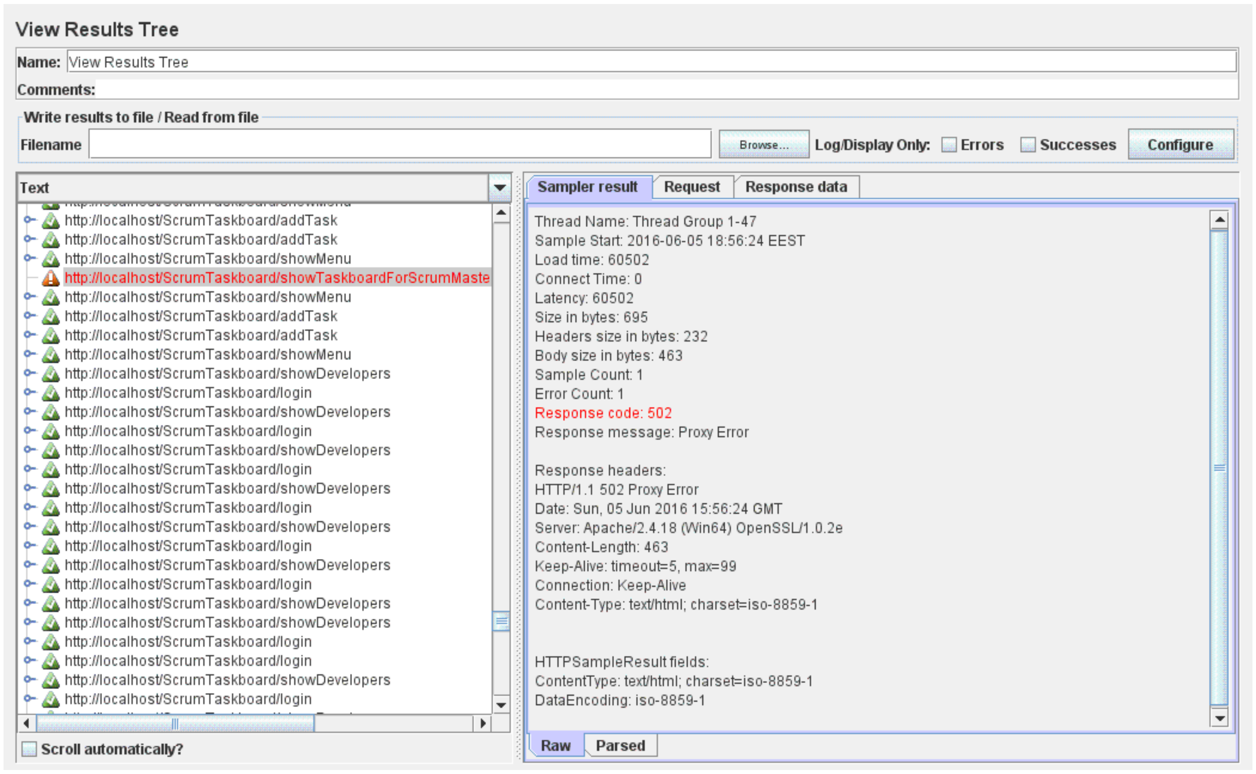
Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/S...	10	16	13	27	5.19	0.00%	10.8/sec	1663.51	158249.0
http://localhost/S...	10	11	7	27	5.42	0.00%	9.0/sec	15.55	1775.0
http://localhost/S...	20	207	9	3155	679.95	0.00%	1.8/min	0.94	32165.9
http://localhost/S...	20	277277	203822	341358	39666.14	0.00%	1.9/min	107.57	3543045.8
http://localhost/S...	30	452	27	3273	921.35	0.00%	9.9/min	0.39	2416.3
http://localhost/S...	10	27940	356	60501	18735.21	20.00%	3.4/min	10.96	195616.5
http://localhost/S...	20	351	26	3128	654.39	0.00%	6.7/min	4.33	39473.6
TOTAL	120	48749	7	341358	103895.19	1.67%	10.7/min	110.61	632688.3

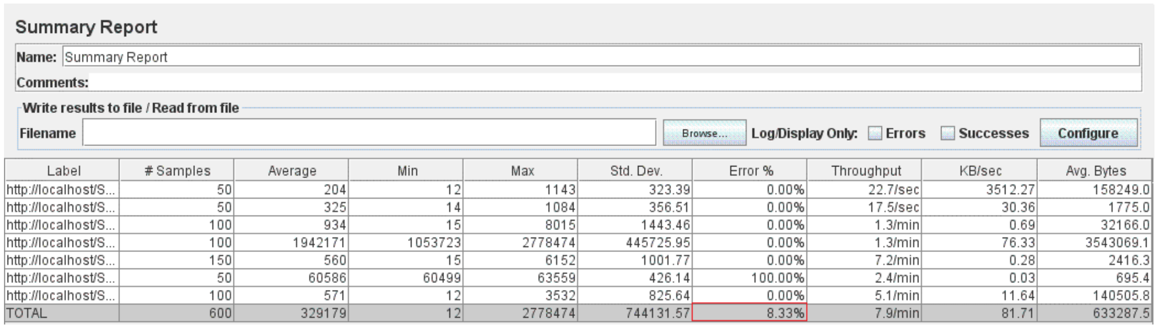
Figure 4.38: Summary Report listener – 10 threads



```
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaste
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/login
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/login
```

```
Thread Name: Thread Group 1-47
Sample Start: 2016-06-05 18:56:24 EEST
Load time: 60502
Connect Time: 0
Latency: 60502
Size in bytes: 695
Headers size in bytes: 232
Body size in bytes: 463
Sample Count: 1
Error Count: 1
Response code: 502
Response message: Proxy Error

Response headers:
HTTP/1.1 502 Proxy Error
Date: Sun, 05 Jun 2016 15:56:24 GMT
Server: Apache/2.4.18 (Win64) OpenSSL/1.0.2e
Content-Length: 463
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1


HTTPSampleResult fields:
ContentType: text/html; charset=iso-8859-1
DataEncoding: iso-8859-1
```

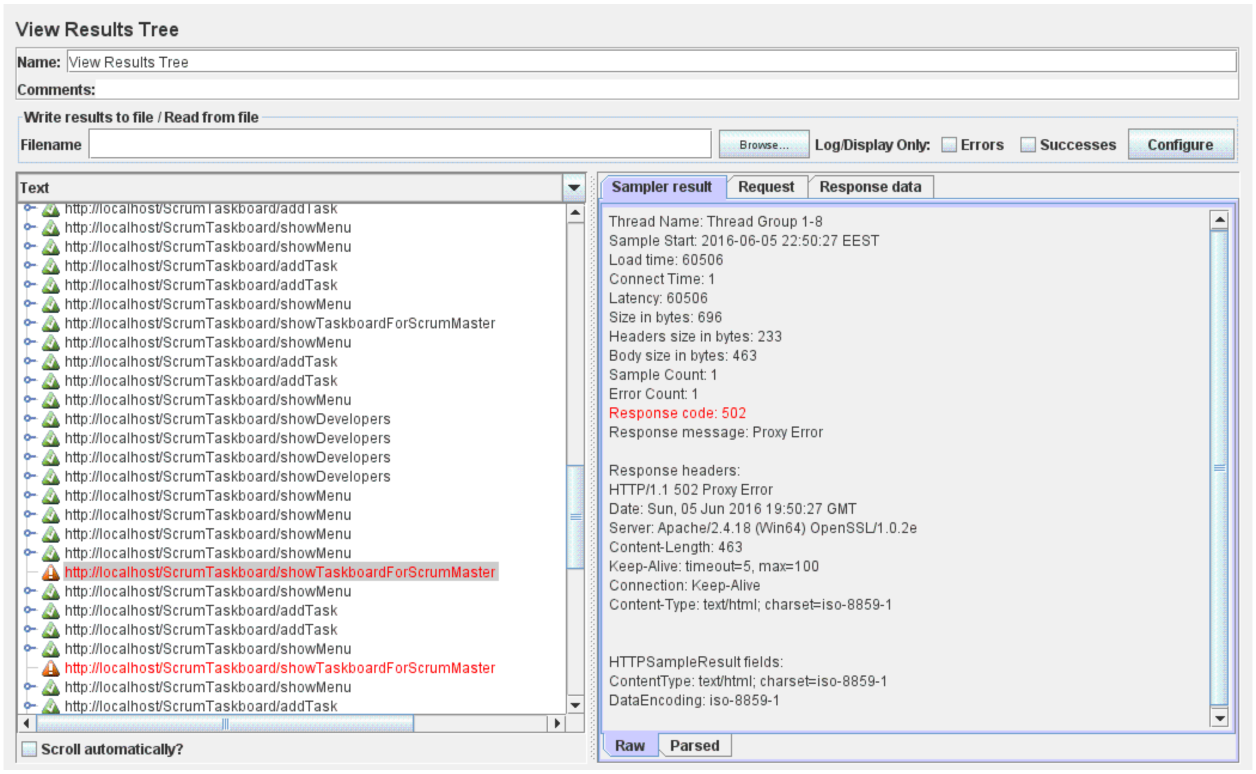
Figure 4.39: View Results Tree listener– 50 threads



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/S...	50	204	12	1143	323.39	0.00%	22.7/sec	3512.27	158249.0
http://localhost/S...	50	325	14	1084	356.51	0.00%	17.5/sec	30.36	1775.0
http://localhost/S...	100	934	15	8015	1443.46	0.00%	1.3/min	0.69	32166.0
http://localhost/S...	100	1942171	1053723	2778474	445725.95	0.00%	1.3/min	76.33	3543069.1
http://localhost/S...	150	560	15	6152	1001.77	0.00%	7.2/min	0.28	2416.3
http://localhost/S...	50	60586	60499	63559	426.14	100.00%	2.4/min	0.03	695.4
http://localhost/S...	100	571	12	3532	825.64	0.00%	5.1/min	11.64	140505.8
TOTAL	600	329179	12	2778474	744131.57	8.33%	7.9/min	81.71	633287.5

Figure 4.40: Summary Report listener – 50 threads

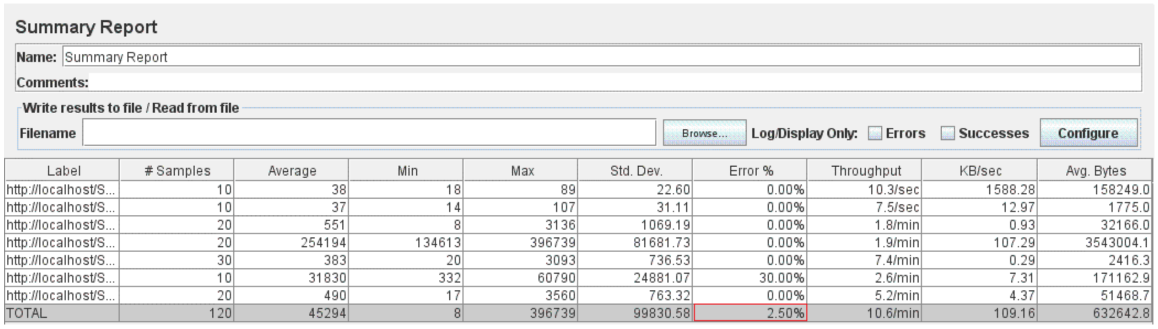
<u>After</u> implementing the previously detailed Spring Eh-Cache, the **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** testcase was executed from JMeter under the same conditions: having 1000 developers existent in the database and by simulating two different numbers of concurrent threads performing the use case (10 and 50). The results obtained from JMeter's View *Results Tree* and *Summary Report* listeners are illustrated in Figures 4.41 – 4.44 as follows.



```
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
```

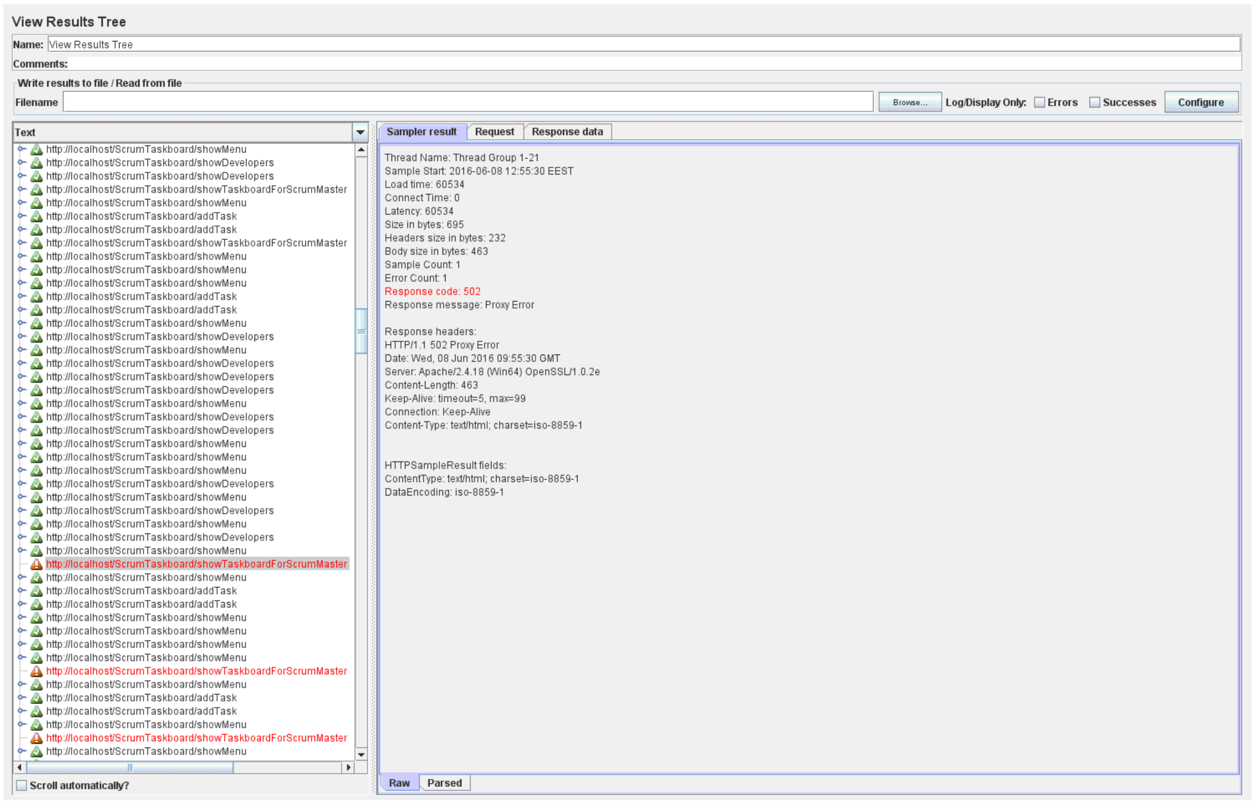
```
Thread Name: Thread Group 1-8
Sample Start: 2016-06-05 22:50:27 EEST
Load time: 60506
Connect Time: 1
Latency: 60506
Size in bytes: 696
Headers size in bytes: 233
Body size in bytes: 463
Sample Count: 1
Error Count: 1
Response code: 502
Response message: Proxy Error

Response headers:
HTTP/1.1 502 Proxy Error
Date: Sun, 05 Jun 2016 19:50:27 GMT
Server: Apache/2.4.18 (Win64) OpenSSL/1.0.2e
Content-Length: 463
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1


HTTPSampleResult fields:
ContentType: text/html; charset=iso-8859-1
DataEncoding: iso-8859-1
```

Figure 4.41: View Results Tree listener– 10 threads



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
http://localhost/S...	10	38	18	89	22.60	0.00%	10.3/sec	1588.28	158249.0
http://localhost/S...	10	37	14	107	31.11	0.00%	7.5/sec	12.97	1775.0
http://localhost/S...	20	551	8	3136	1069.19	0.00%	1.8/min	0.93	32166.0
http://localhost/S...	20	254194	134613	396739	81681.73	0.00%	1.9/min	107.29	3543004.1
http://localhost/S...	30	383	20	3093	736.53	0.00%	7.4/min	0.29	2416.3
http://localhost/S...	10	31830	332	60790	24881.07	30.00%	2.6/min	7.31	171162.9
http://localhost/S...	20	490	17	3560	763.32	0.00%	5.2/min	4.37	51468.7
TOTAL	120	45294	8	396739	99830.58	2.50%	10.6/min	109.16	632642.8

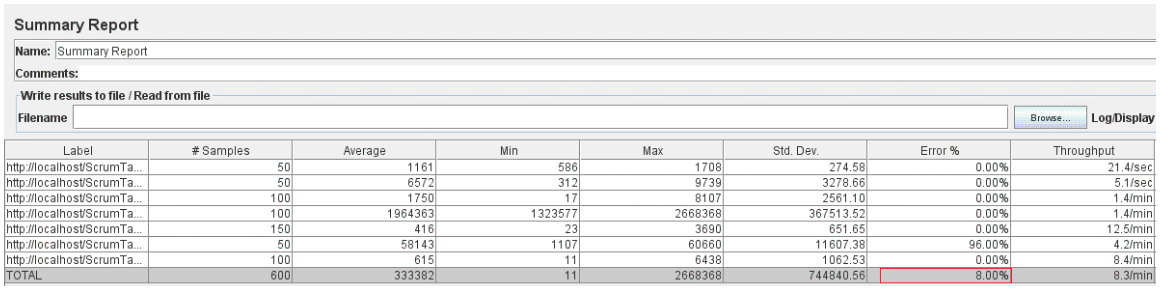
Figure 4.42: Summary Report listener – 10 threads



```
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showDevelopers
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/addTask
http://localhost/ScrumTaskboard/showMenu
http://localhost/ScrumTaskboard/showTaskboardForScrumMaster
http://localhost/ScrumTaskboard/showMenu
```

```
Thread Name: Thread Group 1-21
Sample Start: 2016-06-08 12:55:30 EEST
Load time: 60534
Connect Time: 0
Latency: 60534
Size in bytes: 695
Headers size in bytes: 232
Body size in bytes: 463
Sample Count: 1
Error Count: 1
Response code: 502
Response message: Proxy Error

Response headers:
HTTP/1.1 502 Proxy Error
Date: Wed, 08 Jun 2016 09:55:30 GMT
Server: Apache/2.4.18 (Win64) OpenSSL/1.0.2e
Content-Length: 463
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1


HTTPSampleResult fields:
ContentType: text/html; charset=iso-8859-1
DataEncoding: iso-8859-1
```

Figure 4.43: View Results Tree listener– 50 threads



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
http://localhost/ScrumTa...	50	1161	586	1708	274.58	0.00%	21.4/sec
http://localhost/ScrumTa...	50	6572	312	9739	3278.66	0.00%	5.1/sec
http://localhost/ScrumTa...	100	1750	17	8107	2561.10	0.00%	1.4/min
http://localhost/ScrumTa...	100	1964363	1323577	2668368	367513.52	0.00%	1.4/min
http://localhost/ScrumTa...	150	416	23	3690	651.65	0.00%	12.5/min
http://localhost/ScrumTa...	50	58143	1107	60660	11607.38	96.00%	4.2/min
http://localhost/ScrumTa...	100	615	11	6438	1062.53	0.00%	8.4/min
TOTAL	600	333382	11	2668368	744840.56	8.00%	8.3/min

Figure 4.44: Summary Report listener – 50 threads

After collecting performance measurements from JMeter’s listeners, both before and after implementing Spring Eh-Cache for the considered use case scenario, the subsequent step consisted of analyzing these results in a comparative manner. Therefore, by exploring Summary Report listener’s results in the context of 10 concurrent threads executing the use case (Figures 4.38 and 4.42), it might be concluded that the error percentage has increased after implementing Spring Eh-Cache (from **1.67%** to **2.50%**). Furthermore, in case of 50 workers configured, the Summary Report listener’s results (Figures 4.40 and 4.44) indicate an insignificant performance improvement obtained by developing Spring

Eh-Cache. This result is indicated by an error reduction from **8.33%** to **8.00%**, as a consequence of adding caching to the considered use case.

As almost no performance enhancement was brought by developing Spring Eh-Cache, some research was conducted in order to determine the root cause of reported errors. By analyzing the View Results Tree listener's results both before and after the Spring Eh-Cache addition, it might be determined that the reported errors are of type **502: Proxy Error**. Therefore, the next phase was to determine the root cause of these Proxy Errors and develop a solution to fix it. This way, the reported error percentages would decrease and, therefore other performance indicators might be analyzed in order to determine the performance improvements offered by Spring Eh-Cache.

Further on, a suite of investigations were conducted in order to determine the Proxy Errors' cause and the resolution obtained indicated that some configurations have to be performed at Apache server's level. More precisely, the application was deployed on a Tomcat application-server embedded into an Apache web-server through which all requests pass. Therefore, the Apache web server was heavy loaded by a multitude of requests when executing the testcase, which caused the Proxy Errors.

After performing some research, a proper solution was designed and implemented at Apache server's level in order to fix these errors. Specifically, some properties were updated in Apache's configuration file (*httpd.conf*) intended to increase timeout specific settings. Further on, the JMeter testcase was executed again and the reported error was **0%**, for both 10 and 50 configured threads. In this context, the evaluation of performance enhancements obtained after implementing Spring Eh-Caching might not be performed anymore based on error as a performance indicator.

Consequently, the use case's response time was selected to serve as indicator for assessing the system's performance capabilities as well as possible performance improvements. In order to evaluate such performance concerns corresponding to the project's version after implementing Spring Eh-Cache, the **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** testcase was also executed with 100 users and the total response time to serve all requests took about ***3 hours.*** In addition, the reported error was also **0%**.

Furthermore, apart from the 3 hours response time, which represent an unsatisfactory, too large amount, an "*OutOfMemory: Java Heap space*" error was reported in JMeter's logs when running the testcase with 100 users. This error indicates the existence of errors inside JMeter's configuration, which might be the source cause of the huge response time required for executing the testcase. As a consequence, the following phase consisted on a suite of investigation activities oriented towards JMeter configuration. More precisely, several solutions were considered in order to properly customize JMeter's parameters to allocate greater memory.

As a result of various approaches followed in order to remove JMeter's "*OutOfMemory: Java Heap space"* error obtained when running the testcase with 100 threads and prove the performance improvements brought by Spring Eh-Cache (by obtaining a significant reduction in use case's execution time as compared to the non-cached version), the following conclusions might be stated:

- The **"***OutOfMemory: Java Heap space"* error was completely removed.
- However, none of the configurations performed were of help in reducing the testplan's execution time. Therefore, for 100 concurrent threads, the testplan's execution takes between 2 and 3 hours.

As no valuable solutions to reduce the use case's response time amount were encountered, the effort to prove the performance improvements brought by the implemented Spring Eh-Cache was *resumed*. Therefore, the research conducted towards monitoring the performance enhancements obtained by implementing Spring Eh-Cache completes with negative results.

The following sub-chapter explores **Second-Level Hibernate Eh-Cache** and evaluates the magnitude of performance improvements obtained.

4.8. Implementing caching using Second-Level Hibernate Eh-Cache and monitoring the performance enhancements obtained

The following phase in the dissertation thesis research was to implement caching using **Hibernate Eh-Cache** for a Java entity model class and test the performance obtained through dedicated measurements.

To begin, implementing **Hibernate Eh-Cache** was performed on behalf of the same scenario used when developing Spring Eh-Cache: **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout**. This use case is performed by an administrator user (say Scrum Master) who logins into the application, visualizes all developers, then visualizes the taskboard, adds a new task (together with the required information for it), visits again the list of developers and then logs out from the application.

In order to evaluate the performance enhancements obtained through a series of metrics, JMeter tool was also used to simulate the execution of this scenario for many concurrent users retrieving the developers. The desired behavior for the implemented cache is to access the database only once (on first call) and then retrieve the developers' information from the cache for successive calls to the find developers functionality.

In essence, the main purpose of the developed Hibernate Eh-Cache was to improve the application's performance when executing the **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** testcase in the context of a larger number of concurrent workers. More precisely, as no enhancement in the testcase's execution time was obtained from the solution implemented on behalf of Spring Eh-Caching (corresponding to the functionality to retrieve all the developers from the database), the following step was to develop Hibernate Eh-Cache in hope of improving the long-running response time (between 2 and 3 hours for a number of 100 concurrent threads). Therefore, **Second-Level Hibernate Eh-Cache** was implemented and properly configured for the *Developer* entity in order to cache database information corresponding to each developer and retrieve it from the second-level cache on successive accesses of the same instance.

Moreover, in order to see considerable performance improvements offered by the Hibernate Eh-Caching implemented, the data being cached (all *Developer* instances), and further on accessed, has to be of large amount. Therefore, all performance tests were executed in the context of 1000 developers existent in the database. However, as this list is retrieved twice for each thread in the selected use case scenario (**Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout**), the simulated number of users (threads) from JMeter cannot be very large (like hundreds or thousands) because this consumes all of the CPU's activity and takes a large amount of time to complete. As a result, the tested use case was executed in the context of 100 concurrent users.

The main steps involved in developing this Hibernate Eh-Cache solution are:

- Add Hibernate Eh-Cache required dependencies in *pom.xml* file:

```
<properties>
        <!-- Hibernate Eh-cache  -->
        <ehcache-core.version>2.6.11</ehcache-core.version>
</properties>

<dependencies>

        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>${hibernate.version}</version>
         </dependency>

         <dependency>
                <groupId>net.sf.ehcache</groupId>
                <artifactId>ehcache-core</artifactId>
                <version>${ehcache-core.version}</version>
        </dependency>

        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-ehcache</artifactId>
                <version>${hibernate.version}</version>
        </dependency>
</dependencies>
```

- Add **Hibernate Eh-Cache configurations** in *persistence.xml* file:

```
<property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="net.sf.ehcache.configurationResourceName"
value="/hibernate-ehcache.xml"/>
```

  where:
  hibernate.cache.region.factory_class - is used to define the Factory class for Second-Level caching,
  hibernate.cache.use_second_level_cache - is used to enable the Second-Level cache and
  net.sf.ehcache.configurationResourceName - is used to define the Eh-Cache configuration file location. By default, Eh-Cache will create separate cache regions for each entity configured for caching. The defaults corresponding to these regions might be overridden inside the customized *ehcache.xml*, specified as **net.sf.ehcache.configurationResourceName** property in hibernate configuration file.

- Define the **Eh-Cache** configuration in */hibernate-ehcache.xml* file in order to override the default one:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"
        monitoring="autodetect" dynamicConfig="true">

        <cache name="developer"
                maxEntriesLocalHeap="10000"  maxElementsInMemory="10000"
eternal="false" timeToIdleSeconds="3000" timeToLiveSeconds="6000">
                <persistence strategy="localTempSwap"/>
        </cache>
</ehcache>
```

Through the ***cache*** element, regions are defined and configured. Therefore, a cache region with “***developer***” name was created and its attributes customized.

- Add *@Cache* annotation on the entity model class to be cached (*Developer.java*):

```
@Entity
@Table(name="developer")
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY, region="developer")
public class Developer {
    ...
}
```

The **@***Cache* annotation is used for specifying the caching configuration and allows to define the *caching strategy* and *region* to be used for the entity model. As the use case scenario under test only needs to read, but not modify, instances of the *Developer* class, the *read-only* cache strategy was used. Moreover, the mapping to the cache region defined in *hibernate-ehcache.xml* file was accomplished through the *region* attribute of the *@Cache* annotation.

A final remark concerning Hibernate Eh-Cache implementation regards the specifics of the selected approach. More precisely, as the Hibernate Eh-Cache configuration was defined in ***persistence.xml*** (and not in a ***hibernate.cfg.xml*** file, as usually done), there is no need of developing custom configuration loading (through Java code), because in *Spring* the context, and therefore ***persistence.xml*** configurations, are automatically loaded at server start-up.

In addition, information regarding the proper configuration of Hibernate Eh-Cache for the *Developer* entity is logged at server start-up:

```
18:07:21,979  INFO localhost-startStop-1 util.LogHelper:46 - HHH000204: Processing PersistenceUnitInfo [name: JpaPersistenceUnit...]
18:07:22,109  INFO localhost-startStop-1 hibernate.Version:54 - HHH000412: Hibernate Core {4.3.11.Final}
...
18:07:23,780 WARN localhost-startStop-1 strategy.EhcacheAccessStrategyFactoryImpl:57 - HHH020007: read-only cache configured for mutable entity [developer]
...
Jul 15, 2016 6:07:30 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 12708 ms
```

Further on, the next step is to assess the performance improvements obtained for the **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** use case execution time. Therefore, the JMeter testcase was run in ***GUI*** mode with only two output listeners (*Summary Report* and *View Results Tree*) attached to the testplan and the execution time took about **2 hours and a half**. Afterwards, the testcase was run in JMeter's ***Non-GUI*** mode with no listener added to the testplan and the total execution time took **more than 3 hours**. Therefore, no performance enhancements in the testcase execution time were obtained by the implemented Hibernate Eh-Cache.

As a result of the disappointing resolution regarding the performance improvements offered by the developed Hibernate Eh-Cache, the subsequent phase consisted of conducting several investigations so that to encounter the possible cause of this issue. This also involved revising the developed solution in order to establish its accuracy and identify possible bottlenecks. Therefore, an improved version of the cache region defined in *ehcache.xml* was considered and implemented:

```
<cache name="developer"
        maxEntriesLocalHeap="10000"        maxElementsInMemory="100000"
maxElementsOnDisk="1"
        eternal="false"
        timeToIdleSeconds="3600"                    timeToLiveSeconds="3600"
statistics="true">
        <persistence strategy="none"/>
</cache>
```

After performing the previously listed code changes, another suite of performance measurements were conducted for both 100 and 50 configured threads. Unfortunately, in case of 100 concurrent users, both JMeter's "*OutOfMemory: Java Heap space"* error was reported and a large execution time was obtained (about 3 hours). Even if in case of 50 threads no error was identified, the testcase execution took about 2 hours.

In this context, the next idea was to investigate whether the implemented Hibernate Eh-Caching was actually used by means of cache statistics. In order to monitor this information from VisualVM tool, the display of cache statistics had to be enabled through the following configurations:

- The *<property name="hibernate.generate_statistics" value="true" />* line was included in *persistence.xml* file.
- The statistics=*"true"* attribute was specified for the cache region defined in *ehcache.xml* file.
- *CacheManager* bean was registered through Java code:

```
CacheManager manager = CacheManager.newInstance();
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
ManagementService.registerMBeans(manager, mBeanServer, true, true, true, true);
```

After enabling the display of cache statistics, JMeter performance tests were re-executed and statistical information offered by VisualVM was analyzed. As a result, it might be concluded that the developed Hibernate Eh-Cache was properly identified (together with the configured attributes). However, ***no*** entry was taken from the cache when calling the developers retrieval functionality, which denotes the Second-Level cache ***not*** being used.

Consequently, a clear reason for the lack of performance enhancements brought by developing Hibernate Eh-Cache was finally encountered. More precisely, the weak performance obtained despite of the implemented Hibernate Eh-Cache solution was caused by the fact that the configured cache was not actually used. Further on, the subsequent phase consisted of identifying the reason for which the developed Hibernate Eh-Cache was not taken into account when accessing the cached functionality (retrieve the list of developers). After a suite of research activities were conducted in this direction, the root cause of the not-applied Hibernate Eh-Cache was encountered as the SQL queries run against the database not being cached. More precisely, in the initial configuration for the developed Hibernate Eh-Cache solution, only the Second-Level Cache was enabled, while the Query Cache was not.

Taking into account this resolution, the target was to design and implement a solution to determine the Hibernate Eh-Caching usage for the developers retrieval functionality. The approach to accomplish this aim consisted of the following steps:

- Update *persistence.xml* file with the property to activate Query Cache, without it HQL queries results not being cached:
  ```
  <property name="hibernate.cache.use_query_cache" value="true"/>
  ```

- Set the enable of caching property for the particular SQL query to be cached at project's persistence level, in the corresponding *Data Access Object* file (*JpaDeveloperDao.java*):
  ```
  Query query = entityManager.createQuery("select d from Developer d")
                                        .setHint("org.hibernate.cacheable", true);
  ```

- Moreover, a fine-grained configuration was specified for the SQL query to be cached by naming the particular cache region to be applied (the one configured in *ehcache,xml* file):
  ```
  Query query = entityManager.createQuery("select d from Developer d")
                               .setHint("org.hibernate.cacheable", true)
                               .setHint("org.hibernate.cacheRegion", "developer");
  ```

- Even if the particular cache region to be used for the SQL query had already been specified, an exception was thrown when the application context started in case a default cache region was not defined in *ehcache.xml* configuration file. Therefore, in order to fix this error, a default cache region was added (also in *ehcache.xml* file) :

  ```
  <defaultCache
   maxElementsInMemory="10000" maxElementsOnDisk="1" eternal="false"
  ```

```
timeToIdleSeconds="3600" timeToLiveSeconds="3600"   statistics="true">
        <persistence strategy="none" />
</defaultCache>
```

As a result of the configured query cache for the ***developer*** region, the following statements are logged at Tomcat server’ start-up:

```
localhost-startStop-1 strategy.EhcacheAccessStrategyFactoryImpl:57 - HHH020007: read-only cache configured for mutable entity [developer]
```

```
tomcat-http--14 internal.StandardQueryCache:90 - HHH000248: Starting query cache at region: developer
```

Further on, the properly configured and used Hibernate Eh-Cache for the developers retrieval functionality was assessed through VisualVM tool by monitoring the ***net.sf.ehcache*** results listed under the ***MBeans*** tab. More precisely, the number of cache hits/misses and associated percentage values for different metrics related to cache usage statistics were displayed for each configured cache region. In the actual context, three caches have been identified in the application: ***developer***, *org.hibernate.cache.internal.**StandardQueryCache*** and *org.hibernate.cache.spi.**UpdateTimestampsCache***, as illustrated in Figure 4.45 below:

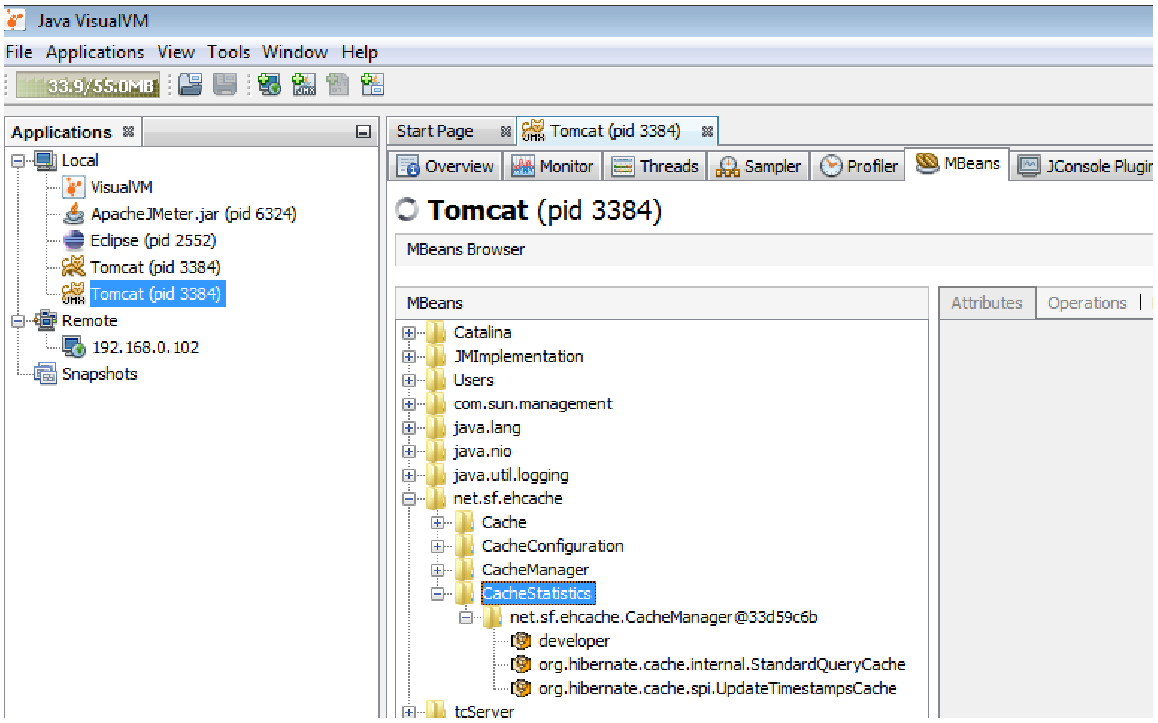
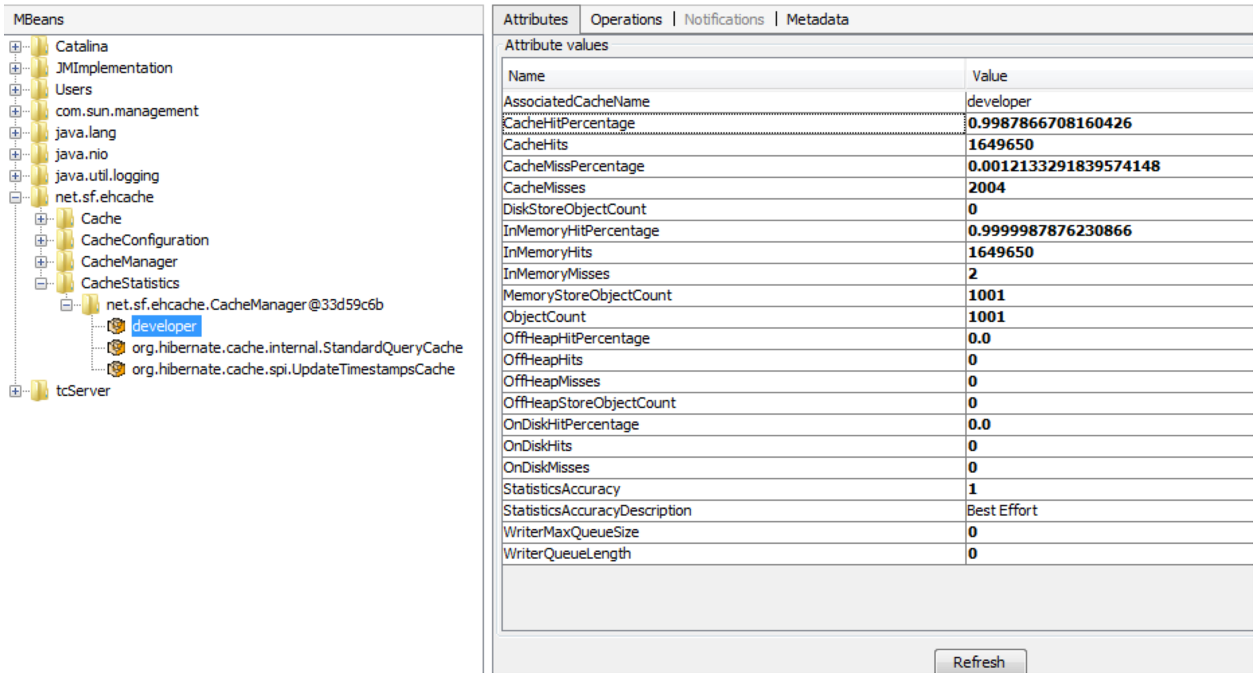


Figure 4.45: VisualVM’s Cache Statistics layout

Afterwards, the JMeter test was executed again in GUI mode (with only *View Results Tree* listener attached to the testplan) and the following results were obtained for a number of 100 concurrent threads:

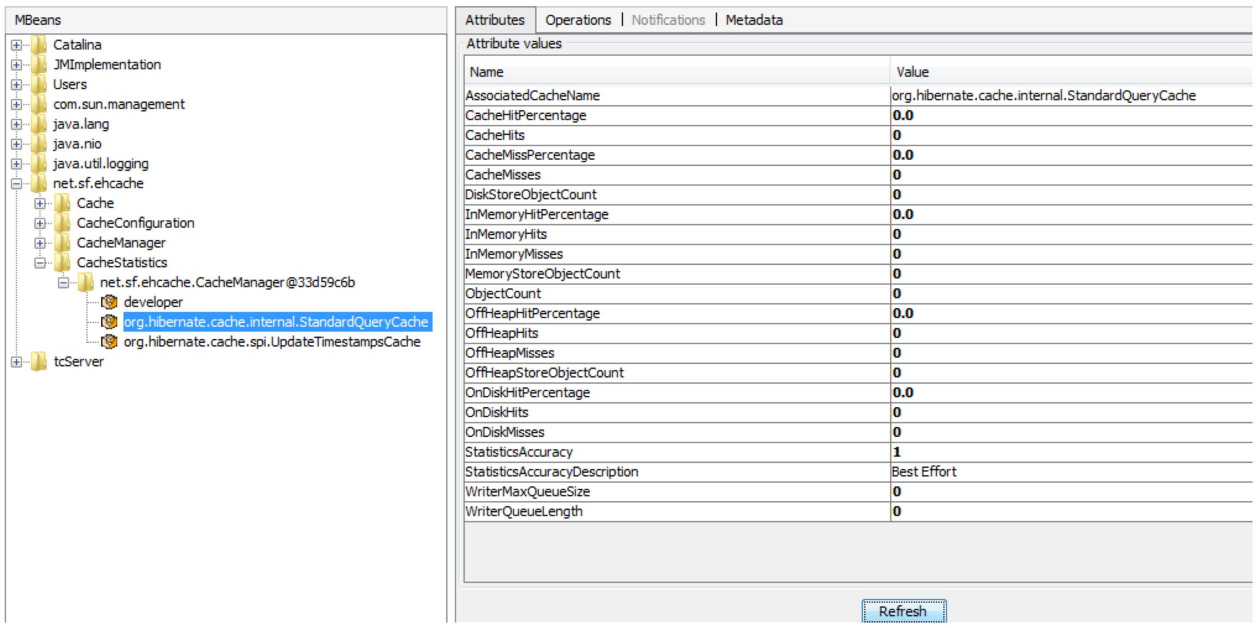
- The execution time took **2 hours and 20 minutes**.
- Also, several "*OutOfMemory: Java Heap space"* errors were reported in JMeter's console.

Meanwhile running the JMeter testplan, the application was monitored using VisualVM tool in order to collect statistical information related to the configured Second-Level Hibernate Eh-Cache. At the end of JMeter testplan execution (executed in GUI mode with 100 concurrent threads), the information associated to cache statistics was collected and analyzed. Consequently, for the three caches listed in VisualVM's ***MBeans*** tab, the following statistics were obtained (Figures 4.46 – 4.48):



Name	Value
AssociatedCacheName	developer
CacheHitPercentage	0.9987866708160426
CacheHits	1649650
CacheMissPercentage	0.0012133291839574148
CacheMisses	2004
DiskStoreObjectCount	0
InMemoryHitPercentage	0.9999987876230866
InMemoryHits	1649650
InMemoryMisses	2
MemoryStoreObjectCount	1001
ObjectCount	1001
OffHeapHitPercentage	0.0
OffHeapHits	0
OffHeapMisses	0
OffHeapStoreObjectCount	0
OnDiskHitPercentage	0.0
OnDiskHits	0
OnDiskMisses	0
StatisticsAccuracy	1
StatisticsAccuracyDescription	Best Effort
WriterMaxQueueSize	0
WriterQueueLength	0

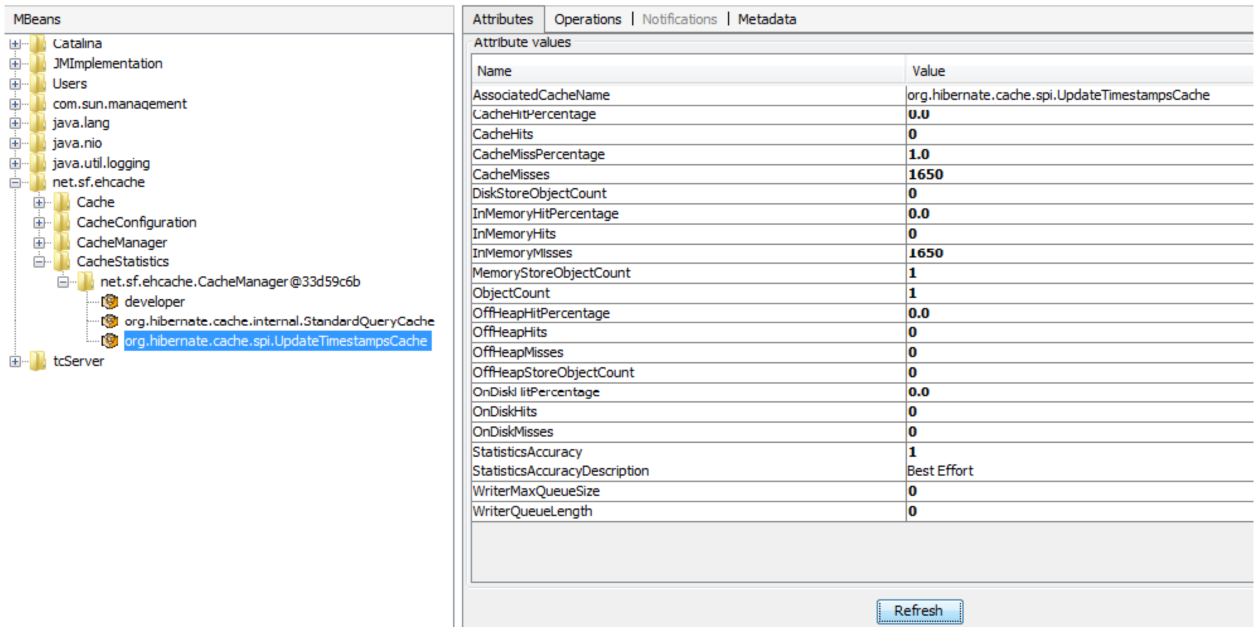
Figure 4.46: VisualVM statistics for ***developer*** cache



```
Catalina
JMImplementation
Users
com.sun.management
java.lang
java.nio
java.util.logging
net.sf.ehcache
    Cache
    CacheConfiguration
    CacheManager
    CacheStatistics
        net.sf.ehcache.CacheManager@33d59c6b
            developer
            org.hibernate.cache.internal.StandardQueryCache
            org.hibernate.cache.spi.UpdateTimestampsCache
tcServer
```

Name	Value
AssociatedCacheName	org.hibernate.cache.internal.StandardQueryCache
CacheHitPercentage	0.0
CacheHits	0
CacheMissPercentage	0.0
CacheMisses	0
DiskStoreObjectCount	0
InMemoryHitPercentage	0.0
InMemoryHits	0
InMemoryMisses	0
MemoryStoreObjectCount	0
ObjectCount	0
OffHeapHitPercentage	0.0
OffHeapHits	0
OffHeapMisses	0
OffHeapStoreObjectCount	0
OnDiskHitPercentage	0.0
OnDiskHits	0
OnDiskMisses	0
StatisticsAccuracy	1
StatisticsAccuracyDescription	Best Effort
WriterMaxQueueSize	0
WriterQueueLength	0

Figure 4.47: VisualVM statistics for *org.hibernate.cache.internal.**StandardQueryCache***



```
Catalina
JMImplementation
Users
com.sun.management
java.lang
java.nio
java.util.logging
net.sf.ehcache
    Cache
    CacheConfiguration
    CacheManager
    CacheStatistics
        net.sf.ehcache.CacheManager@33d59c6b
            developer
            org.hibernate.cache.internal.StandardQueryCache
            org.hibernate.cache.spi.UpdateTimestampsCache
tcServer
```

Name	Value
AssociatedCacheName	org.hibernate.cache.spi.UpdateTimestampsCache
CacheHitPercentage	0.0
CacheHits	0
CacheMissPercentage	1.0
CacheMisses	1650
DiskStoreObjectCount	0
InMemoryHitPercentage	0.0
InMemoryHits	0
InMemoryMisses	1650
MemoryStoreObjectCount	1
ObjectCount	1
OffHeapHitPercentage	0.0
OffHeapHits	0
OffHeapMisses	0
OffHeapStoreObjectCount	0
OnDiskHitPercentage	0.0
OnDiskHits	0
OnDiskMisses	0
StatisticsAccuracy	1
StatisticsAccuracyDescription	Best Effort
WriterMaxQueueSize	0
WriterQueueLength	0

Figure 4.48: VisualVM statistics for *org.hibernate.cache.spi.**UpdateTimestampsCache***

By analyzing this statistical cache information, it might be easily figured out that **only** the ***developer*** cache is used by the application. This is caused by the fact that both configured Hibernate Eh-Caches (the Second-Level one, implemented for the *Developer* entity and the Query-Level one, used when executing the SQL query to find all developers existent in the database) use the ***developer*** cache region defined in *ehcache.xml* configuration file. Therefore, all accesses to cached data from the ***developer*** cache region (for either Second-Level or Query-Level caches) are saved in the statistics corresponding to the ***developer*** cache. Furthermore, the high scores recorded for *CacheHits* and *InMemoryHits* and low ones for *CacheMisses* and *InMemoryMisses* (as well as their associated percentages) indicate a high usage of the cached data stored in the ***developer*** region.

As a result, the actual goal of the developed Hibernate Eh-Cache was reached: when accessing data related to developers, cached information is retrieved instead of invoking the database for each request. However, despite of the encouraging statistical results concerning Hibernate Eh-Cache usage, *no performance enhancement* in JMeter testplan's execution time was obtained.

Additionally, several investigations were carried out in order to determine the root cause of the "*OutOfMemory: Java Heap space*" error reported when executing the JMeter testplan with 100 threads. Finally, this error was removed by adjusting memory-specific attributes in JMeter's executable file (*jmeter.bat*):

- Line *set HEAP=-Xms2048m –Xmx****8192****m* was changed to *set HEAP=-Xms2048m -Xmx****4096****m*
- Line *set NEW=-XX:NewSize=****128****m -XX:MaxNewSize=****128****m* was changed to to *set NEW=-XX:NewSize=****256****m -XX:MaxNewSize=****512****m*
- Line *set PERM=-XX:PermSize=****64****m -XX:MaxPermSize=****128****m* was changed to *set PERM=-XX:PermSize=****256****m -XX:MaxPermSize=****512****m*

With the above configurations in place, when running the JMeter testplan in GUI mode with 100 threads no "*OutOfMemory: Java Heap space*" was reported but the total execution time took **3 hours and 5 minutes**. This large response time indicates that *no performance enhancement* for **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** testplan was obtained.

Finally, the research conducted towards monitoring the performance improvements obtained by implementing Second-Level Hibernate Eh-Cache completes with unsatisfactory results as the target of reducing the use case's execution time was not accomplished.

The last sub-chapter focuses on implementing a Load Balancing mechanism and assessing the performance enhancements resulted.

4.9. Developing a Load Balancing mechanism and monitoring the performance enhancements obtained

The dissertation thesis research completed with the development of a load balancing solution intended to maximize the system's performance capabilities. Finally, dedicated performance tests offered a precise resolution concerning the enhancements brought by the configured load balancer.

The basic idea behind clustering is to ensure better availability and operability of a system by using a group of computers, or cluster, instead of a single computer. Furthermore, clustering is usually part of a larger load balancing process in which requests are distributed among the nodes (or computers) to spread the workload evenly.

Implementing the load balancing mechanism was evaluated on the same scenario used throughout the previous sub-chapters: **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout**. This use case is performed by an administrator user (say Scrum Master) who logins into the application, visualizes all developers, then visualizes the taskboard, adds a new task (together with the required information for it), visits again the list of developers and finally logs out from the application.

By following the same approach as in the last sub-chapters, performance enhancements evaluation was performed using JMeter to simulate the execution of this scenario for many concurrent users retrieving the developers. In essence, the target of implementing a load balancing mechanism was to maximize performance in the context of many concurrent threads, by distributing the large number of requests to several Tomcat server instances. This way, overloading a single server would be avoided and considerable reduction in the testcase execution time obtained.

Therefore, the implemented cluster consisted of the Apache web-server acting as a load balancer to evenly distribute the requests among two Tomcat application-servers. Moreover, due to performance considerations, the configured load balancer was included inside a virtual host. Basically, setting up the load balancing scheme involved performing the following configuration in Apache's *httpd-vhosts.conf* file:

```
<VirtualHost _default_:80>
DocumentRoot "${SRVROOT}/htdocs"
ServerName scrumtaskboard.localhost.com

Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e;
path=/" env=BALANCER_ROUTE_CHANGED

<Proxy balancer://testcluster>
BalancerMember http://127.0.0.1:8080/ScrumTaskboard route=node1
BalancerMember http://127.0.0.1:8081/ScrumTaskboard route=node2
```

```
ProxySet stickysession=ROUTEID
ProxySet lbmethod=byrequests
</Proxy>

ProxyPass /ScrumTaskboard/ balancer://testcluster/ timeout=6000
ProxyPassReverse /ScrumTaskboard/ balancer://testcluster/ScrumTaskboard

ProxyRequests off

<Location /balancer-manager>
SetHandler balancer-manager
AuthType Basic
AuthName "Balancer Manager"
AuthUserFile "C:/Apache24/conf/.htpasswd"
Require valid-user
</Location>

</VirtualHost>
```

In the above cluster configuration, the *balancer manager* component provides a nice web interface of the load balanced cluster, used in remote administration or monitoring activities.

Afterwards, in order to assess the speedup brought by the configured load balancer, the **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** was considered as sample use case. For 100 threads, executing JMeter test in Non-GUI mode took **2 hours and 6 minutes**, as might be determined from the logs:

```
2016/08/07 03:33:10 INFO  - jmeter.engine.StandardJMeterEngine: Running the test!
2016/08/07 03:33:10 INFO  - jmeter.samplers.SampleEvent: List of sample_variables: []
2016/08/07 03:33:10 INFO  - jmeter.gui.util.JMeterMenuBar: setRunning(true,*local*)
2016/08/07 03:33:10 INFO  - jmeter.engine.StandardJMeterEngine: Starting ThreadGroup: 1 : Thread Group
2016/08/07 03:33:10 INFO  - jmeter.engine.StandardJMeterEngine: Starting 100 threads for group Thread Group.
2016/08/07 03:33:10 INFO  - jmeter.threads.ThreadGroup: Starting thread group number 1 threads 100 ramp-up 1 perThread 10.0 delayedStart=false
2016/08/07 03:33:10 INFO  - jmeter.threads.JMeterThread: jmeterthread.startearlier=true (see jmeter.properties)
2016/08/07 03:33:10 INFO  - jmeter.threads.JMeterThread: Thread started: Thread Group 1-2
```

```
2016/08/07 03:33:10 INFO  - jmeter.threads.JMeterThread: Thread started: Thread Group 1-1
...................................................................................................................................
2016/08/07 05:39:41 INFO  - jmeter.threads.JMeterThread: Thread finished: Thread Group 1-16
2016/08/07 05:39:41 INFO  - jmeter.engine.StandardJMeterEngine: Notifying test listeners of end of test
2016/08/07 05:39:41 INFO  - jmeter.gui.util.JMeterMenuBar: setRunning(false,*local*)
```

As no significant response time improvements were obtained by this cluster configuration, a more suitable approach was to distribute the requests load on two Tomcat servers located on different physical machines. Consequently, the new cluster configuration implemented in Apache’s *httpd-vhosts.conf* file is the following:

```
<VirtualHost _default_:80>
DocumentRoot "${SRVROOT}/htdocs"
ServerName scrumtaskboard.dissertation.com

Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/" env=BALANCER_ROUTE_CHANGED

<Proxy balancer://testcluster>
BalancerMember http://127.0.0.1:8080/ScrumTaskboard route=node1
BalancerMember http://192.168.0.102:8080/ScrumTaskboard route=node3

ProxySet stickysession=ROUTEID
ProxySet lbmethod=byrequests
</Proxy>

ProxyPass /ScrumTaskboard/ balancer://testcluster/ timeout=6000
ProxyPassReverse /ScrumTaskboard/ balancer://testcluster/ScrumTaskboard
ProxyRequests off

<Location /balancer-manager>
SetHandler balancer-manager
AuthType Basic
AuthName "Balancer Manager"
AuthUserFile "C:/Apache24/conf/.htpasswd"
Require valid-user
</Location>
</VirtualHost>
```

The above configuration shows an Apache load balancer which distributes the requests among two Tomcat server nodes: one on the localhost (represented by **127.0.0.1** IP address **-** the same computer where the JMeter testplan was run) and another one belonging to a remote host (having the IP address **192.168.0.102**). Both Tomcat server instances run on behalf of the same code base and used the same database stored inside the localhost machine.

The JMeter testplan for the **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** scenario was rerun (in Non-GUI mode) with 100 threads and the execution time took **3 hours and 8 minutes**, as indicated in the following logs:

```
2016/08/14 05:59:57 INFO  - jmeter.gui.util.JMeterMenuBar: setRunning(true,*local*)
2016/08/14 05:59:57 INFO  - jmeter.engine.StandardJMeterEngine: Starting ThreadGroup: 1 : Thread Group
2016/08/14 05:59:57 INFO  - jmeter.engine.StandardJMeterEngine: Starting 100 threads for group Thread Group.
2016/08/14 05:59:57 INFO  - jmeter.threads.ThreadGroup: Starting thread group number 1 threads 100 ramp-up 1 perThread 10.0 delayedStart=false
2016/08/14 05:59:57 INFO  - jmeter.threads.JMeterThread: Thread started: Thread Group 1-1
2016/08/14 05:59:57 INFO  - jmeter.threads.JMeterThread: Thread started: Thread Group 1-2
2016/08/14 05:59:57 INFO  - jmeter.threads.JMeterThread: Thread started: Thread Group 1-3
………………………………………………………………………………………………
2016/08/14 09:07:44 INFO  - jmeter.threads.JMeterThread: Thread is done: Thread Group 1-33
2016/08/14 09:07:44 INFO  - jmeter.threads.JMeterThread: Thread finished: Thread Group 1-33
2016/08/14 09:07:44 INFO  - jmeter.engine.StandardJMeterEngine: Notifying test listeners of end of test
2016/08/14 09:07:44 INFO  - jmeter.gui.util.JMeterMenuBar: setRunning(false,*local*)
```

Again, no significant improvements in the testcase execution time were brought by this cluster configuration. As a result, the following step was to add a third Tomcat server to the actual configuration:

```
<VirtualHost _default_:80>
DocumentRoot "${SRVROOT}/htdocs"
ServerName scrumtaskboard.dissertation.com
```

```
Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/" env=BALANCER_ROUTE_CHANGED

<Proxy balancer://testcluster>
BalancerMember http://127.0.0.1:8080/ScrumTaskboard route=node1
BalancerMember http://127.0.0.1:8081/ScrumTaskboard route=node2
BalancerMember http://192.168.0.102:8080/ScrumTaskboard route=node3

ProxySet stickysession=ROUTEID
ProxySet lbmethod=byrequests
</Proxy>
...
</VirtualHost>
```

Therefore, the new cluster setup consists of an Apache load balancer which distributes the requests among **three** Tomcat server nodes: two on the localhost (on ports **8080 and 8081** of the **127.0.0.1** IP address **-** same computer where the JMeter testplan was run) and another one belonging to a remote host (with IP address **192.168.0.102**). Similar to the previous case, all Tomcat servers run on behalf of the same code base and used the same database stored inside the localhost machine.

Further on, the JMeter testplan corresponding to **Login_Visualize developers_Visualize taskboard_Add task_Visualize developers_Logout** scenario was run (also in Non-GUI mode) with 100 threads and the execution time took **1 hour** and **37 minutes**, result determined from the logs:

```
2016/08/15 06:52:56 INFO  - jmeter.gui.util.JMeterMenuBar: setRunning(true,*local*)
2016/08/15 06:52:56 INFO  - jmeter.engine.StandardJMeterEngine: Starting ThreadGroup: 1 : Thread Group
2016/08/15 06:52:56 INFO  - jmeter.engine.StandardJMeterEngine: Starting 100 threads for group Thread Group.
2016/08/15 06:52:56 INFO  - jmeter.threads.ThreadGroup: Starting thread group number 1 threads 100 ramp-up 1 perThread 10.0 delayedStart=false
2016/08/15 06:52:56 INFO  - jmeter.threads.JMeterThread: Thread started: Thread Group 1-1
2016/08/15 06:52:56 INFO  - jmeter.threads.JMeterThread: Thread started: Thread Group 1-2
..........................................................................................................................
2016/08/15 08:30:01 INFO  - jmeter.threads.JMeterThread: Thread is done: Thread Group 1-54
2016/08/15 08:30:01 INFO  - jmeter.threads.JMeterThread: Thread finished: Thread Group 1-54
```

```
2016/08/15 08:30:01 INFO  - jmeter.engine.StandardJMeterEngine: Notifying test listeners of end of test
2016/08/15 08:30:01 INFO  - jmeter.gui.util.JMeterMenuBar: setRunning(false,*local*)
```

The obtained result represents a **significant improvement** as the total response time was reduced with about **30 minutes** as compared to the best case scenario before setting up the load balancer (2 hours and 6 minutes). As a consequence, this last version of load balancing mechanism (with Apache web-server acting as load balancer and three Tomcat server nodes operating as workers) represents the most suitable clustering solution for the application under test because it reduces the total response time, thus maximizing the system's performance.

This sub-chapter completes the suite of methodologies explored throughout the dissertation thesis research. Finally, Chapter 5 summarizes the conclusions gathered on behalf of the investigations conducted.

Chapter 5. Conclusions

As a result of the research detailed in Chapter 4, the dissertation thesis consists of a suite of phases oriented towards improving the application's performance capabilities. Each phase might be perceived as a methodology dedicated to analyzing and implementing the most suitable solution in the actual context. Moreover, at the end of each phase specialized performance measurements were conducted in order to evaluate the enhancements obtained, as well as their magnitude.

By analyzing the investigations presented in the previous chapter, a strong interconnection among the thesis research steps might be denoted. Even if each phase in the dissertation thesis development follows an approach specific to a certain technique, intended to enhance the overall application's performance, they cannot be accomplished in isolation. Therefore, the entire investigation might be defined as a continuous process of monitoring and improving a system's performance features, in which each such phase, or methodology, plays an essential role.

More precisely, starting from the performance shortcomings concluded on behalf of the evaluations conducted in the initial phase, the second one aims to remove, or at least reduce, the threat affecting the application's operation when an increased number of concurrent users try to access the system. In order to accomplish this, several possible approaches were considered and the most suitable one was established by taking into account the actual stage of the dissertation thesis development process. Specifically, this step focused on reviewing the initial source code so that to identify possible performance vulnerabilities and completed with a cleaner, error-free code base due to implemented code improvements. Following comes the phase intended to evaluate the system's performance capabilities after the underlying code base passed through a complete revision followed by the implementation of clean-up, refactoring strategies and improvement solutions. In order to evaluate and monitor the expected performance improvements, JMeter tool was used and similar measurements to the ones performed in the initial phase were conducted. However, not only did the expected, targeted enhancement in the system's performance never occur, but also a larger error percentage as compared to the initial version was reported, indicating a severe performance downgrade behavior. As the reason behind this issue was unknown, specialized research was conducted throughout the next phase of the dissertation thesis development. Apart from study and research activities, the literature's recommendation to execute JMeter tests using the command-line (console) and reserve the GUI usage for test recording, development and debugging activities was also practiced. After executing, JMeter tests for a sample use case scenario in both GUI and console modes, the performance comparison concluded with negative results. More exactly, for different numbers of concurrent users, not only did the expected enhancement in the system's performance for the console mode execution never occur, but also larger error percentages as compared to the ones obtained for JMeter GUI mode were reported. Despite of these discouraging results, the process was not abandoned and it continued with a further investigation intended to encounter the root cause of the system's performance vulnerability.

In this respect, VisualVM tool was used to conduct advanced application monitoring and obtain accurate information regarding the system's bottleneck. The association between JMeter and VisualVM tools provided accurate information regarding the precise source of failure for the web application under test, indicating that further improvements should be implemented at the database level. In this context, the following sub-chapter focused on exploring and developing several database indexes approaches followed by dedicated performance experiments intended to determine their suitability for each particular use case scenario. This step completed with a summary of the conclusions gathered as well as some general recommendations. When reaching this step in the dissertation thesis research, a new performance boosting approach was considered as adding cache for the web application under test. This way, two caching methodologies were considered for the scope of the thesis research, Spring Eh-Cache and Hibernate Eh-Cache, which were briefly detailed throughout the next two phases in the investigation process. Consequently, the subsequent methodology focused on briefly presenting the applicability of Spring Eh-Cache on the system and ended with experimental results obtained from performance tests execution. However, it completed with negative results as no valuable solutions to reduce the sample use case's response time were encountered and therefore, the effort to prove the performance improvements brought by the implemented Spring Eh-Cache was resumed. The same unsatisfactory result was obtained for the next phase when implementing Hibernate Eh-Cache for a Java entity model class, as the target of reducing the use case's execution time was not accomplished. However, by implementing the caching strategy detailed throughout this step, the initially reported errors when executing the JMeter testcase were completely removed.

Finally, the dissertation thesis research completed with the development of a load balancing solution intended to maximize the system's performance capabilities. By using dedicated performance measurements, it was proved that the configured load balancer offered a significant improvement as the use case's response time was reduced with about 30 minutes as compared to the best case scenario before setting up the load balancer. Therefore, this successful load balancing strategy was considered as being the most suitable clustering solution for the application under test because it reduced the total response time, thus maximizing the system's performance. As a consequence, this last phase completed the suite of methodologies explored throughout the dissertation thesis research.

To conclude, the development of the dissertation thesis consisted of a suite of phases, perceived as definite methodologies, throughout specific activities were carried out. Namely, these activities were research, analysis, design, implementation, testing and monitoring, each of them playing an essential role in completing each phase. Despite of the result obtained for each step, all of them were considered successful as they offered the opportunity to practically explore various software engineering methodologies and gather insights into Application Performance Monitoring area.

Bibliography

[1] APM Digest, *The Anatomy of APM*, 4 April 2012.

[2] Dubie, D., *Performance management from the client's point of view*, NetworkWorld, Retrieved 22 March 2013.

[3] APM Digest, *APM and MoM -Symbiotic Solution Sets*, 11 May 2012.

[4] Realtime NEXUS , *What You Should Know About APM* – Part 1, 2013.

[5] Gartner Research, *Keep the Five Functional Dimensions of APM Distinct*, 16 September 2010.

[6] APM Digest, *Analytics vs. APM*, 28 January 2013.

[7] Gartner, *Magic Quadrant for Application Performance Monitoring*, Retrieved 18 December 2013.

[8] APM Digest, *APM Convergence: Monitoring vs. Management*, 6 March 2013.

[9] TRAC Research, *Application Performance Management Spectrum*, 11 March 2013.

[10] *Differences between approaches to APM - a chat with Jesse Rothstein of Extrahop*, ZDNet.com, 9 December 2011.

[11] Realtime NEXUS, *The Five Essential Elements of Application Performance Monitoring*, 2010.

[12] APM Digest, *Prioritizing Gartner's APM Model: The APM Conceptual Framework*, 15 March 2012.

[13] SearchNetworking, *Application performance monitoring tools: Three vendor strategies*, 25 March 2013.

[14] APM Digest, *Insight from the User Experience Management Panel in Boston*, 23 March 2012.

[15] Forbes, *Big Data and Advanced Analytics: Success Stories From the Front Lines*, 3 December 2012.

[16] Sydor, M.,J., *APM Best Practices: Realizing Application Performance Management*, Apress, 2010.

[17] Waller, J., *Performance Benchmarking of Application Monitoring Frameworks*, Books on Demand, 2014.

[18] Croll, A., Power, S., *Complete Web Monitoring: Watching your visitors, performance, communities, and competitors*, O'Reilly Media Inc., 2009.

[19] Huang, B., Kadali, R., *Dynamic Modeling, Predictive Control and Performance Monitoring: A Data-driven Subspace Approach*, Springer Science & Business Media, 2008.

[20] Shields, G., *The Definitive Guide to Application Performance Management*, Realtime Publishers, 2009.

[21] Erinle, B., *JMeter Cookbook*, PACKT Publishing, 2014.

[22] Halili, E., *Apache JMeter*, PACKT Publishing, 2008.

[23] Hunt, C.; Binu, J., *Java Performance*, Prentice Hall, 2012.

[24] Standtke, R., *Pretty Good Anonymity: Achieving High Performance Anonymity Services with a Single Node Architecture*, Logos Verlag Berlin GmbH, 2013.

[25] Johnson, R. et al., *Spring Framework Reference Documentation – 3.2.4.RELEASE*, 2004-2013.

[26] Miles, R. et al., *Programming Spring*, O'Reilly Media, Inc., 2013.

[27] Partner, J., Bogoevici, M. and Fuld, I. , *Spring Integration in Action*, Manning Publications Company, 2012.

[28] *Spring Framework 3.1 Tutorial*, tutorialspoint.com.

[29] Sharma, J., Sarin, A., *Getting Started with Spring Framework*, CreateSpace Independent Publishing Platform, 2012.

[30] Wheeler, W., Wheeler, J., *Spring in Practice*, Manning Publications Company, 2013.

[31] Brannen, S., et al., *Spring in a Nutshell*, Michigan, O'Reilly Media, Inc., 2012.

[32] Bourke, T., *Server Load Balancing*, O'Reilly Media, Inc., 2001.

[33] Membrey, P., et al., *Practical Load Balancing: Ride the Performance Tiger*, Apress, 2012.

[34] Laurie, B., Laurie, P., *Apache: The Definitive Guide*, O'Reilly Media, Inc., 2003.

[35] The Apache Software Foundation, *Apache HTTP Server 2.2 Official Documentation - Volume IV. Modules (I-V)*, Fultus Corporation, 2010.

[36] Coar, K., et al., *Apache Cookbook: Solutions and Examples for Apache Administration*, O'Reilly Media, Inc., 2008.

[37] Garner, S. R., *Data Warehouse Implementation Strategies: A Mixed Method Analysis of Critical Success Factors*, ProQuest, 2007.

[38] Singh, H., *Data Warehousing: Concepts, Technologies, Implementations, and Management*, Prentice Hall PTR, 1998.

[39] Higgins, K. R., *An Evaluation of the Performance and Database Access Strategies of Java Object-Relational Mapping Frameworks*, ProQuest, 2007.

[40] Ciugudean, M., Gorgan, D., *Methodology for Identification and Evaluation of Web Application Performance Oriented Usability Issues*, in Romanian Journal of Human Computer Interaction 9(2) 2016, 155-176.

[41] Ciugudean, M., Gorgan, D., *Methodology for Identification and Evaluation of Web Application Performance Oriented Usability Issues*, in Proceedings to ROCHI Conference, 8-9 Sept., Iasi, 2016.

Appendix 1

As author of two papers, oriented towards the same topic and having the same application basis as the dissertation thesis, I have included them in this section. The first one presents an extras of Romanian Journal of Human Computer Interaction (RRIOC) [40], while the second paper represents the version sent to ROCHI Conference to be included in the Proceedings [41].