Chapter No. 1
"Getting Started with Performance Tuning"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.1 "Getting Started with Performance Tuning"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Osama Oransa is an IT solution architect with more than 12 years of solid technical experience in Java EE. He is a certified Java enterprise architect and an SME in web services technology. He has worked for most of the key players in the IT industry, such as IBM, Oracle, and Hewlett Packard. He previously worked as a performance consultant at DevFactory, and he is currently working with the Vodafone Group as a solution architect. He has also participated in establishing Pulse Corp as a medical software services company in Egypt.

He has a diploma in IT from the Information Technology Institute (ITI) and a diploma in CS from the Arab Academy for Science, Technology and Maritime Transport (AASTM). He is currently working towards a Master's degree in CS. Being from Cairo, he is a frequent speaker at the Java Developer Conference (JDC) in Cairo.

In 2010, one of his projects in Pulse Corp, "Health Intact", won Oracle Duke's Choice Award. He is the founder of more than 12 open source projects hosted on SourceForge. He has also been selected by Oracle for the future of the Java campaign for his valuable contribution to the industry.

He is a volunteer Java technology evangelist who gives technical sessions at different companies, conferences, and on blogs. His technical blog can be found at http://osama-oransa.blogspot.com/.

For More Information:
Acknowledgments

After more than 17 years since my first IT book on the Assembly language, I got the chance to write my second book. It was really a nice experience to deal with the wonderful people at Packt Publishing. After thanking my God, I would like to thank my parents and my wife for their full support while writing this book. Special thanks to Reshma Raman, the Author Acquisition Executive who contacted me to write this book.

I would like to thank Priyanka Goel, the Project Coordinator who helped me to deliver the book following the agreed timeline and who gave me her full support.

I would like to thank Nikhil Karkal, the Acquisition Editor, for his support in the early, critical phases of the book and his continuous efforts in providing me with the proper guidelines.

I would like to thank Dayan Hyames, the Lead Technical Editor, for his efforts in reviewing the content of this book and guiding me to provide the best outcome.

I would like to thank the technical reviewers E.P. Ramakrishnan, Chirag Sangani, Deepak Vohra, and Jeff Zhang, who caught my errors and provided me with very constructive and valuable comments to enhance the content of this book.

I would like to thank the technical editors Novina Kewalramani and Humera Shaikh for their tremendous efforts in the book.

Finally, thanks goes to all the people who participated in this book who I don't know by name—without you, this book wouldn't see the light. Thank you!

For More Information:
Performance tuning is one of the biggest challenges in dealing with the Java Enterprise applications, where each delay in response to the customer might cause the enterprise to lose valuable customers. In particular, during marketing campaigns, such as launching a new product, the load over the application jumps to a peak. It is common to see some enterprise applications that do not respond to customers! If such a thing happens and the application cannot withstand such load, it will not only lead to a loss of the expected sales, but also destroy the reputation of the enterprise.

During years of working in many large companies with different enterprise applications, which are usually composed of many integrated applications that are provided by different vendors, many challenges have contributed to the performance tuning of these applications, such as how to design a high performing application, how to minimize the occurrences of performance issues, how to build a performance team, how to test the application performance efficiently, how to create an investigation plan, and many other questions that need to be answered correctly. We tried to address these questions and others in this book, aiming to help establish a clear, organized, and smooth way of dealing with performance tuning of enterprise applications.

We have focused, in particular, on simplicity and clarity to remove any mystery around the art of performance tuning. This book is designed in a progressive pattern where each chapter represents a step forward towards completing the complete process of handling the performance issues from end-to-end in any enterprise application, starting from the requirement phase to the production support phase.

Performance tuning activities happen usually in the most stressful times in the enterprise application's life! When I started my work in performance tuning, it was the task that I needed to accomplish as soon as possible. This led to an unnecessary increase in stress and tension. Later, I learned the best way to deal with performance tuning; dealing with it as an art and enjoying the work in performance tasks (just like painting a picture or completing a puzzle). It is an art that we need to respect and enjoy. In this book, we will try to cover all the required details to understand performance tuning and the best way to deal with this art.

For More Information:
Dealing with performance tuning as an art and organizing our thoughts are the most important concepts here, and are also the most helpful factors (other than the person's luck!). Spending months to write this book and organizing its different sections was a very challenging procedure, in particular, to establish a well-defined, organized way of dealing with performance tuning—starting from the knowledge, which is considered as our solid foundation here, up to the strategic thinking. While dealing with these issues, establishing and finding this thinking strategy is much more important than having a look-up book that describes each problem and the possible resolution. This thinking strategy along with the required skills and mastery of different tools will provide the best way to guarantee the resolution of any new performance issues whenever this strategy is strictly followed.

In addition to that, Java Enterprise Edition 7 (Java EE 7) platform introduced a lot of new features and improvements that enterprise applications can benefit from performance perspective. This is a part of our essential knowledge to work in performance tuning of Java EE 7 applications.

Because of all of this, I encourage you to go through this book chapter by chapter, be patient and read the book to the end, download the required tools, and practice all the examples to gain the maximum value from this book.

**What This Book Covers**

*Chapter 1, Getting Started with Performance Tuning*, takes you through the art of performance tuning with its different components and shows you how to think when we face any performance issue. It focuses on preparing you to deal with the world of performance tuning and defining the handling tactics.

*Chapter 2, Understanding Java Fundamentals*, lays the foundation of required knowledge of the new features in Java Enterprise Edition 7 and different important Java concepts, including the JVM memory structure and Java concurrency. It also focuses on the different Java Enterprise Edition concurrency capabilities.

*Chapter 3, Getting Familiar with Performance Testing*, discusses performance testing with its different components, defines useful terminologies that you need to be aware of, and then gives hands-on information about using Apache JMeter to create your performance test plans for different components and get the results.

*Chapter 4, Monitoring Java Applications*, dissects the different monitoring tools that will be used in performance tuning, starting from the operating system tools, different IDE tools, JDK tools, and standalone tools. It covers JProfiler as an advanced profiling tool with its offline profiling capabilities.

For More Information:  
Chapter 5, Recognizing Common Performance Issues, discusses the most common performance issues, classifies them, describes the symptoms, and analyzes the possible root causes.

Chapter 6, CPU Time Profiling, focuses on the details of getting the CPU and time profiling results, ways to interpret the results, and ways to handle such issues. It discusses the application logic performance and ways to evaluate different application logics. It provides the initial performance fixing strategy.

Chapter 7, Thread Profiling, discusses thread profiling with details on how to read and interpret thread profiling results and how to handle threading issues. It also highlights the ways to get, use, and read the thread dumps.

Chapter 8, Memory Profiling, discusses how to perform memory profiling, how to read and interpret the results, and how to identify and handle possible issues. It also shows how to read and query memory heap dumps and analyze the different out of memory root causes. The chapter finishes your draft performance fixing strategy.

Chapter 9, Tuning an Application’s Environment, focuses on tuning the application environment, starting from the JVM and passing through other elements such as the application servers, web servers, and OS. We will focus on selected examples for each layer and discuss the best practices for tuning them.

Chapter 10, Designing High-performance Enterprise Applications, discusses design and architecture decisions and the performance impact. This includes SOA, REST, cloud, and data caching. It also discusses the performance anti-patterns.

Chapter 11, Performance Tuning Tips, highlights the performance considerations when using the Agile or Test-driven Development methodologies. This chapter also discusses some performance tuning tips that are essential during the designing and development stages of the Java EE applications, including database interaction, logging, exception handling, dealing with Java collections, and others. The chapter also discusses the `javap` tool that will help you to understand the compiled code in a better way.

Chapter 12, Tuning a Sample Application, includes hands-on, step-by-step tuning of a sample application that has some performance issues. We will measure the application performance and tune the application issues, and re-evaluate the application performance.

For More Information:
Getting Started with Performance Tuning

Before we start digging in our book to discuss performance tuning in Java enterprise applications, we need to first understand the art of performance tuning: what is this art? Can we learn it? If yes, how?

In this chapter, we will try to answer these questions by introducing you to this art and guiding you through what you need to learn to be able to master this art and handle performance issues efficiently.

We will try to focus more on how to prepare yourself to deal with performance tuning, so we will discuss how to build your way of thinking before and after facing performance issues, how to organize your thoughts, and how to lead your team successfully to build an investigation plan.

In this chapter, we will cover the following topics:

- Understanding the art of performance tuning
- Understanding performance issues and possible root causes from a software engineering perspective
- Tactics to follow when dealing with performance issues
- The difference between handling standalone and web applications from a performance perspective
- How to troubleshoot web application performance issues

For More Information:

Understanding the art of performance tuning

Performance tuning is an art. Yes, a real art, and fortunately we can learn this art because it is based on science, knowledge, and experience. Like any artist who masters the art of drawing a good picture using his coloring pencils, we need to master our tools to be able to tune the performance of our applications as well.

As we are going to cover performance tuning in Java Enterprise Edition 7, the key to master this art starts from understanding the basic concepts of Java—what are the different capabilities of Java EE until the release of v7, how to use the different performance diagnostic tools available, and finally how we can deal with different performance issues.

The final question is related to how we can program our minds to deal with performance issues, and how we will build our own tactics to address these performance issues. But our solid land here is our knowledge and the more we stand on solid land (that is, knowledge), the more we will be able to handle these performance issues efficiently and master this performance tuning art.

Of course with our continuous dealing with different performance issues, our experience will grow and it will be much easier to draw our picture even with a few set of colors (that is, limited tools). The following diagram shows the basic components of Java EE performance tuning art:

For More Information:
As shown in the previous diagram, we have six basic components to master the performance tuning art in Java EE; four of them are related to our knowledge from bottom to top: Understand **environment** (like OS), Understand **Java/JVM**, Understand **Java EE** (we should also have some level of knowledge of the framework used in developing the application), and finally Mastering **tools**.

The challenge we face here is in the **Way of thinking** element where we usually need to get trained under an expert on this domain; a possible alternative for this is to read books or tutorials on how we can think when we face performance issues and put it into practice bit by bit.

In this chapter, our focus will be on how we should be thinking and defining our tactics when we deal with performance issues and in the next few chapters, we will apply this thinking strategy so we can master these tactics.

There are other factors that definitely contribute to how we can use our skills and affect our outcome; this includes, for example, the working environment, that is, different constraints and policies.

If we are not able to access the performance test environment to set up the required tools, we would have a high rate of failure, so we need to minimize the impact of such a risk factor by having complete control over such environments.

As an early advice, we as performance experts, should lead the performance-related decisions and remove all existing constraints that can potentially affect our job. We should know that no one will really point to any condition if we failed; they will just blame us for not taking corrective actions for these bad constraints and it will end up destroying our credibility.

"Don't ever blame conditions; instead do your best to change them!"

- **Osama Oransa**

One important thing that should be noted here is that if for any reason we failed to improve the performance of an application or discover the root cause of some performance issues, we will definitely learn something that should contribute to our accumulated knowledge and experience, which is "don't do it that way again!". Remember the following famous quote:

"I have not failed. I've just found 10,000 ways that won't work."

- **Thomas Edison**

---

For More Information:
When people get overconfident, they are easily susceptible to failure especially when they don't stick to their own troubleshooting process and follow some bad practices; one famous bad practice is jumping to a conclusion early without any real evidence, so the golden advice that we need to stress on here is to always try to stick to our defined process (that is, the way of thinking) even when the issue is really obvious to us, otherwise it will end up being a big failure!

**Understanding performance issues**

We can define performance tuning issues in general as any issue that causes the application to perform outside the target service-level agreement.

Performance issues can take many forms, for example, increased response time is one of the common forms. Let's list a few forms of application performance issues, as follows:

- Slow transactional response with or without application workload
- Failure to meet the processing rate, for example, 1,000 submitted orders per second
- Failure of the application to serve the required number of concurrent users
- Non-responding application under workload
- Transactional errors during application workload, which could be reported by application users or seen in the application logfiles
- Mismatch between application workload and resource utilization, for example, CPU utilization is 90 percent with a few users or memory utilization is 70 percent even during no user activity
- Abnormal application behavior under certain conditions, for example, the application's response slows down daily at midnight
- All other aspects of application failure to meet functional or nonfunctional requirements under workload

We must differentiate between the application's consistent slow response, and sudden, gradual, or intermittent changes of an application's response to be more sluggish or slower.

Having a design issue is the most common reason behind consistent slow behaviour, which is usually associated with missing or bad quality performance tests that didn't discover such issues early on. Dealing with these issues in the production environment is very difficult, especially if they affect a lot of users' transactions.
The other types of sudden or gradual deterioration of the application response time in some transactions can also be design issues but in most cases, it requires a small fix (for example, configuration, database script, or code fix), and usually we can deploy the issue resolution in the production environment once the fix is tested in the test environment.

User transaction here refers to the set of actions/interactions in a single scenario; it could include a wizard or navigation scenario in our application or it could also be a single interaction or a sequence of interactions.

For example, all these are considered to be user transactions: login, add to basket, checkout, update user data, and so on.

Unfortunately, a majority of performance tuning work is executed in a production environment, where the situation becomes more critical and the environment becomes more sensitive to major changes. When we deal with performance tuning of such applications, we should push the transformation to the correct software engineering model so we can have the performance testing stage in place to catch most of the performance issues early on in the application development cycle.

**Classifying performance issues by the discovery phase**

If we classify performance issues according to their discovery time in the typical waterfall development process, we can see the following main categories:

- Requirement issues, mainly related to a missing or unrealistic service-level agreement
- Design issues, where the design is the root cause of these issues
- Development issues, such as not following best coding practices, or bad coding quality
- Testing issues, such as missing or bad quality performance testing
- Operational issues, which are mainly related to production environment-specific issues, such as the database size or newly introduced system, and so on

For More Information:

**Requirement phase and design-time issues**

The best location to discover any potential performance issue is at the design stage where the designer can reduce the cost by discovering and fixing such issues in later steps of the design stage.

The identification of performance issues here means highlighting and taking into consideration some critical **Service-Level Agreements (SLAs)** and also finding possible alternatives for any technology/vendor restrictions.

An SLA is part of a service contract where a service is formally defined; it describes the agreement between the customer and the service provider(s). SLAs are commonly used for nonfunctional requirements like performance measurement, disaster recovery, bug fixing, backup, availability, and so on.

Let's consider the following example.

Let's assume that we have the following SLA (nonfunctional requirements) in our requirement document:

*Under the workload of 1,000 concurrent users, the maximum response time allowed should be less than 0.1 second per web service call.*

The preceding SLA seems hard to achieve under workload so the designer should be doing the following things:

- Seeking some clarifications on what is meant by "workload" here
- Trying to work around the SLA by adding, for example, a cache to such web services
- Trying to develop a quick **Proof Of Concept (POC)** to get early figures in such a situation

A POC is simply a realization of a certain idea to assess its feasibility, usually small and not completed. In our situation, we need to assess the performance of using such a technology and predict if an SLA can be achieved or not.
We cannot say these are actual performance issues, but they are potential performance issues that will violate the SLAs; the conclusion here is that the designer must pay attention to such requirements and find the best design approach for similar requirements, which should be reflected in all the application layers. Such requirements, if not taken into consideration earlier, should still be caught later in the performance testing phase, but it will be too late for big code changes or architecture/design decisions.

We can consider this as a proactive measure rather than a real reactive measure. It is clearly important in an agile development methodology, where the designer is already familiar with the current system behavior and restrictions, so spotting such issues early on would be easy.

We will discuss the different design decisions and potential performance impact in more details in Chapter 10, Designing High-performance Enterprise Applications.

**Development-time issues**

This is where lucky teams discover performance issues! This is almost the last stage where such issues could be fixed by some sort of major design changes, but unfortunately it is not common to really discover any performance-related issues during the development stage, mainly due to the following reasons:

- The nature of the development environment with its limited capabilities, low resources profile (for example, small memory size), logging enablement, and using few concurrent users, where most of the performance issues usually appear under workload.
- Development of the database is usually a small subset of the application's production database, so no valid comparison to the actual performance in the production database.
- Most of the external dependencies are handled through stubbing, which prevents the real system performance examination.

**Stubbing** means simulating the behavior of the components. It can be done in the following two ways:

- Using a simulator for receiving the request and sending a response back
- Reading the response from an I/O resource with optionally configured wait time to simulate the system's average response time

For More Information:  
• Slow response time nature of the development environment, so the developers neglect any noticeable slow response in the application.
• Continuous changes in the development environment, so that application developers usually adapt to dealing with unstable environments. Hence, they wouldn't actually report any performance issues.
• The stressful development stage where no one would open the door for additional stress.

Testing-time issues
In a typical software engineering process, there should be performance testing in the testing stage of the application development cycle to ensure that the application complies with the nonfunctional requirements and specified SLAs like stability, scalability, response time, and others.

Unfortunately, some projects do not give any importance to this critical stage for different reasons such as budget issues or neglecting small deviations from the SLA, but the cost of such wrong decisions would definitely be very high if a single performance issue is discovered in the production environment especially if the system is dealing with sensitive user data that restricts access to some environment boxes.

Production-time issues
From the previous performance issue types, we now understand that this type is the nightmare type; it is the most critical and costly one, and unfortunately it is the most common type that we will deal with!

We can summarize the main reasons behind discovering performance issues in the production environment as follows:
• Missing or non-efficient performance testing (process and quality issue)
• Under estimation of the expected number of application users with no proper capacity planning (quality issue)
• Non-scalable architecture/design (quality issue)
• No database cleanup, so it keeps growing, especially in large enterprise applications (operational issue)
• No environment optimization for the application from the operating system, application server, database, or Java Virtual Machine (operational issue)
• Sudden changes to the production environment without proper testing of the impact on the performance (operational and process issue)
• Other reasons like using stubs in the testing environment instead of actual integrated test systems, unpredictable issues, and so on
All the issues discussed previously can be summarized in the following diagram:

![Diagram showing impact and fixing cost of discovered performance issues during different project phases]

It is important to note that, in the production environment, we should only handle performance issues; no optimization or tuning is to be implemented in the production environment without an actual reported and confirmed issue. Optimization without a real issue can be conducted in the performance testing environment or during development only; otherwise, we are putting the whole application into high-functionality risk, which is much more crucial than reducing the response time a little. So, the production environment is only for fixing critical performance issues that would impact the business and not a place for tuning or improving the performance.

"Things which matter most must never be at the mercy of things which matter least."

-Johann Wolfgang von Goethe

Classifying performance issues by the root phase

In our previous classification, we focused on issue identification time, but if we classified these issues according to the possible root cause from the software engineering perspective, we can have the following types of performance issues:

- Requirement phase issues
- Design/architecture phase issues
- Development phase issues
- Testing phase issues
- Operational and environmental-specific issues

For More Information:
Getting Started with Performance Tuning

Requirement phase issues
Here, no clear (or sometimes unrealistic) SLAs are present, which do not match the actual production expectations.

Design/architecture phase issues
Here, the design does not fulfill the provided SLA, or is built on certain assumptions retrieved from the vendor specifications without any proof of concept to confirm these assumptions. Also, sometimes the design takes some architecture decisions that do not fulfill the actual customer performance requirements.

The design and architecture phases are very critical as the impact here is not easily fixable later without major changes and high costs that always make such decisions very difficult and risky as well.

We will discuss performance issues related to design in Chapter 10, Designing High-performance Enterprise Applications.

Development phase issues
Bad coding quality, not following performance-oriented coding practices, and missing essential code review (either automated or manual) are the main reasons for this phase.

Following best coding practices should always be forced by the project leaders to avoid any potential issues related to applications that do not perform well; they are not difficult to follow, especially if automated code review tools are used early on during the development phase.

We will discuss some of the development performance issues in Chapter 11, Performance Tuning Tips.

Testing phase issues
This phase's issues occur mainly due to missing or bad quality performance testing including test scripts, test scenarios, number of test users, environment selection, and so on.

For More Information:
We should know that testing responsibilities are the biggest here as developers usually claim they did their job well, but testing should either confirm or nullify this claim.

We will discuss performance testing in detail in Chapter 3, Getting Familiar with Performance Testing.

Operational and environmental-specific issues

A lot of operational issues could impact the application performance, for example, missing frequent housekeeping activities, failure to monitor the application, not taking early correction steps, and implementing improperly-tested changes to the environment (or any of the integrated systems).

Sometimes, specific environment issues like the size of application database, unexpected customer flow, and so on can lead to bad performance in the production environment that we can't catch earlier in the performance test environment.

We will discuss different application monitoring tools in Chapter 4, Monitoring Java Applications.

Performance-handling tactics

Dealing with performance issues is a risk management procedure that should be handled with preventive and curative measures, so we need to stick to the following techniques for successful and peaceful management of performance issues:

- Proactive measures (preventive)
- Reactive measures (curative)

Proactive measures (preventive)

Proactive measures aim to reduce and minimize the occurrence of performance issues by following the software engineering processes properly and having efficient performance requirement, early capacity planning, high quality application development, and proper application testing with special focus on performance testing.

For More Information:
Having the required monitoring tools in place and ensuring that the operation team has the required knowledge is an important aspect. We also have to request the output samples of these tools periodically to ensure that the tools are available to help us when we need them.

The proactive tactics only decrease the possibility of performance issues but do not nullify them, so we should still be expecting some performance issues but we will be in a good position to deal with them as everything we need should be ready.

One of the proactive measures is that we should give a "no go" decision for the application. In case the application fails to pass the agreed SLAs in the performance test environment, it is much easier to troubleshoot and fix issues in the performance environment as compared to the sensitive and stressful production environment.

We can summarize the main proactive tactics as follows:

- Having a working process in place for performance tuning, which should typically include reporting of issues, fixing cycles, and testing processes.
- Having a clear performance SLA and good capacity planning.
- Performance-oriented application design (design documents should be performance reviewed).
- Following best coding practices along with automated and manual code reviews; most of the automated code review tools catch a lot of fine tuning issues. Also, strictly following best application logging practices that can help analyze the issues and prevent performance issues related to logging.
- Having a dedicated performance environment that is more or less similar to the production environment specifications.
- Designing and executing good quality performance testing.
- Training and dedicating a team to handle performance issues.
- Having the tools required for performance ready.
- Continuous monitoring of different application layers from trained operational teams.

In Chapter 3, Getting Familiar with Performance Testing, we will discuss performance testing and its related processes in detail that will cover a lot of these points.
Reactive measures (curative)
These are the tactics that we need to follow when we face or discover any performance issues. If the proactive tactics are already followed, then the reactive tactics would be straightforward and smooth.

Understanding the different layers of an enterprise application
Before we discuss the reactive tactics, we need to have a look at the simple Java enterprise application layers; they are illustrated in the following diagram:

As we can see in the preceding diagram, the application layers represent the code on the top of the pyramid along with some database and configuration scripts.

For More Information:
Getting Started with Performance Tuning

When we plan to deal with performance issues, we should consider each of these pyramid layers in our investigation. We don't know at which layer we will have the bottleneck, so as an initial conclusion, we need to monitor each of these layers with the suitable monitoring tools: Operating System (OS), Java Virtual Machine (JVM), Application Server (AS), Database Server (DB), Virtual Machine (VM)—if it exists, and hardware and networking.

Somehow, the application is usually tightly coupled with the development framework and used libraries, so we can treat them as one layer from the tooling perspective if splitting them is not possible.

One of the common mistakes is to focus on a single layer like the code layer and neglect other layers; this should be avoided. If we have the required monitoring tools for all of these layers, our decision will definitely be much clearer and well guided.

In Chapter 4, Monitoring Java Applications, we will discuss the monitoring tools in detail.

Now, let's have a look at the three important pillars required to enable our performance tuning: process, tools, and team!

**The three pillars required for performance tuning**

The following three aspects in the vertices of the triangle need to be fulfilled before we start any performance tuning work; they aim to enable us to work efficiently in application performance tuning:

[Image of a triangle with three circles: Define the process, Get all required tools, Prepare yourself (be ready)]
Define the performance process

This is the first and most important task. We need to ensure this process is already in place before we start any work. We should understand the existing performance tuning process, and if the process does not already exist, then we need to create and define one to use.

The process should include many major elements like performance environment, the reporting of performance issues, fixing cycles, acceptable/target performance goals, monitoring tools, team structure (including well-defined roles and responsibilities), and sometimes a performance keyword glossary to clear any possible misunderstanding.

The reporting of performance issues is the important part here to avoid falsely reported issues and wasting unnecessary time on fake issues. The process should handle the confirmation of reported issues and should cover all necessary steps for issue replication and issue evidence, such as log extract, screenshots, recording, and so on.

It is worth adding here that both lesson-learned sessions and performance knowledge-base content should be part of our performance process execution to reduce the occurrence of repeated performance issues in the future.

Getting ready with the required performance tools

Tools are our coloring pencils, as we described them before, and without them we will not be able to draw the picture. As a part of proactive measures, suitable and sufficient monitoring tools should already be installed in both testing and production environments. We should also obtain periodic reports from these tools to ensure that they are working and helpful at the same time; these tools also give us the required application performance baseline, so we can compare any deviations with this baseline.

If the diagnostic tools are not already installed, they should at least be ready for installation. This means that we have at least selected them, checked the compatibility requirements, and secured the essential licenses, if any.

Since most of the monitoring tools are focused on monitoring certain layers of our application, we need to secure at least one tool per layer. The good news is that each layer usually comes with useful monitoring tools that we can use, and we will discuss these tools in more detail in Chapter 4, Monitoring Java Applications.
Being ready to deal with performance issues at any time

Now, as a team, it's our turn to be ready even if we haven't faced any performance issues. If we are already facing some performance issues, then we need to be ready to handle our investigation plan.

Leading the performance team and giving them sufficient guidance and recommendations is our job, and it is our call to give decisions and bear the responsibility of any consequences.

"It is the set of the sails, not the direction of the wind that determines which way we will go."

-Jim Rohn

As mentioned before, the first and most essential thing that we need to consider is to confirm that we really are facing a performance issue; this can be done in many ways including replicating the issue, checking a recorded scenario, extracting information from logfiles with the response time recorded, and so on.

Once the issue is confirmed, it's our turn to build the investigation plan. We should focus on the root cause identification rather than fixing the issue. Of course, our goal is to fix the issue and this is what we will get paid for, but we need to fix it with a proper permanent solution and this won't happen unless we discover the correct root cause.

The cycle of learning

The cycle of learning summarizes the process that we need to follow once we have performance issues reported till we fix it. If we take a look at the following diagram that illustrates the cycle of learning, we can see that we must have the following milestones to progress with our learning cycle:

- Knowing where the issues are being reported
- Analysis and investigation by different tools
- Thinking of a way to fix it according to the existing inputs that we have from different tools
- Providing a proper fix for the issue

The cycle is repeated from the first step to test and validate the fix. If all the existing issues get resolved, then the cycle is broken; otherwise, we will keep reporting any issues and go through the cycle again.

For More Information:
We need to follow this model and typically try to start the cycle from the reporting step in our model. The following diagram illustrates this model as a whole:

Learning cycle is developed by Peter Honey and Alan Mumford, based on David A. Kolb's ideas about learning styles. The following are the stages of the learning cycle:

- Doing something, having an experience
- Reflecting on the experience
- Concluding from the experience, developing a theory
- Planning the next steps, to apply or test the theory

Honey and Mumford gave names to the people who prefer to enter the cycle at different stages: activist, reflector, theorist, and pragmatist. While different people prefer to enter at different stages, a cycle must be completed to give a learning that will change behavior.

Let's assume we have an online shopping company that has claimed that their own website's response time deteriorated and a lot of users/customers did not continue their own journeys, and the application logs show frequent timeout and stuck threads (we will explain all these issues later in the book).
The company called a performance tuning expert to lead the investigation in this critical situation, who put in some effort without any progress. The operation team noticed that when they restart the cluster servers one by one, the issues disappeared from the site and they asked if this could be recommended as a solution!

Now, if the performance expert followed this recommendation, the issues will only be masked; the company will be deceived and the issue will explode again at any moment. So, don't think of the solution or the fix but focus on how to identify the reason or the root cause behind this issue. Once discovered, the correct solution will follow.

**Tuning yourself before tuning the application**

We need to remember the following points each time we are leading the investigation to resolve any performance issues. They are all related to our behavior and attitude when we are working on performance tuning.

**Be a true leader**

Working on an enterprise application's performance tuning as a performance specialist, we would usually have a team to work with and we should lead and guide this team efficiently.

Here are some of a leader's traits that we need to show the team: support, help, guide, inspire, motivate, advice, listen, and have patience while dealing with their mistakes.

Having a good attitude and behavior towards the team will relieve the pressure from the team and motivate them to work.

**Use your power**

A successful leader effectively uses some of his/her own powers to influence the team. A lot of different individual powers are available but we should be much more oriented towards using either knowledge/expertise or charismatic powers. These power types have a stronger impact on the team.

**Be responsible**

A leader shouldn't be self-defending and blame the team for failure, instead the leader should be responsible for the team. Throwing the issues under team responsibility will impact the team's progression to resolve the issue; instead we need to protect our team and give them full support and guidance and bear the consequences of our own decisions.

For More Information:  
Trust your team
The team will be much more efficient when we show them our complete trust and support; the more we guide them in a clearly-organized process, the more successful a team we will have.

Keep it simple
If we can't explain the plan in a simple and clear way to our team, then we don't really understand what we are planning to do and we should consider redesigning our investigation plan.

Keep it simple (KISS) rule whenever you are leading a team in your investigation.

Respect roles and responsibilities
Everyone should do what is required from them according to their own roles as agreed in the performance process. This will give us the best outcome when everyone is focusing on their own job.

We shouldn't volunteer to do what is beyond our scope, or we will be wasting our time in unnecessary tasks that make us lose our focus. The only exception here is that if there is no one in our team who can do this task and it is really important and relevant to our work, then we can take it up.

Understand the application domain and context
As we are targeting Java enterprise applications performance tuning, a variety of enterprise application technologies exist and applications are built using different frameworks. Before we deal with such applications, we need to understand the framework capabilities and framework-related monitoring tools very well.

A good example here is Oracle ATG e-commerce; this framework supports configuration of the application in layers so we can turn on/off different properties in each application layer or package. Without understanding this simple concept, we won't be able to progress in our troubleshooting to achieve even simple tasks such as enabling the application logging in a certain component. Also, the framework has its own performance monitoring tools that are disabled by default in ATG live configurations. Without knowing this basic information, we won't progress well.

For More Information:
Art Technology Group (ATG) was an independent Internet technology company specializing in e-commerce software and on-demand optimization applications until its acquisition by Oracle on January 5, 2011.

Protect your reputation
No one can harm our reputation more than us; this is a fact, unfortunately. So, for instance, we need to avoid the following things that could destroy our reputation:

- **Don't ever try to shoot in the dark**: If we do not have a solid input from different performance monitoring and analysis tools, then we shouldn't ever try to guess where the issue is. This means our main objective is to have the required tools in place to provide us with the essential inputs.

- **Don't use trial and error**: Trial and error is a good approach for juniors and developers and for learning purposes, but not for performance experts. Also, it is okay to have some trials but don't expand using this approach as it will give a bad impression of insufficient knowledge. It should be mainly used to confirm our thoughts, not to resolve the issue.

- **Quantify your expectations**: Always have a doubt in what is being reported, so don't accept vague words like "the server is okay" or "memory utilization is good". Instead, we should check the results ourselves and ask for solid figures and numbers.

- **Don't jump to conclusions early**: Most of the early conclusions made are not true, so try to be more conservative. Jumping to a conclusion early will convert the current investigation into trials to prove that conclusion! One famous example here is the "same values and different interpretations" issue where the single value doesn't mean the same in all domains. So, let's assume we have an application with low CPU utilization; this doesn't necessary mean the application is fine! Instead, it could point to inefficient CPU utilization and is potentially caused by threading or concurrency issues.

- **If it is dark, step back to see some light**: If the current investigation does not reveal any indicators about the issue's root cause and we keep looping without any progress, then try to step back and look from a wider angle to see the missing parts of the picture. Involving other people from outside the current team could help give us some insight.

For More Information:
• Don't talk too much: In other words, we need to talk a little and think a lot. Don't give what you are thinking of to others, even if you have some early indicators. Keep them for now until we have the required evidence, or even better to keep these thoughts till the issues get resolved. The only exception here is talking to the team to educate them and guide them into the correct direction, or talking during brainstorming sessions.

**Standalone applications versus web applications**

We are going to discuss the different application tier models here so that we understand the behavior of each application type and the expected tuning effort for each type. While we are working with Java application tuning, we will mainly face the following three different types of application:

• **One-tier application**: In this application everything is installed on one machine only; a standalone application without any remote/external connections.

• **Multi-tier application**: This application is installed on different tiers; two different client types according to the client role, either a thick (fat) client or thin client.

• **Smart/rich client application**: These are the applications where the client can work offline and interact with a remote application online through some interfaces like web services. From a performance tuning perspective, we will deal with this type, which is similar to dealing with a thick client.

**The standalone application**

This application has the following main characteristics:

• Runs on a single machine (personal computer, tablet, phone, and so on)
• Connects to a local database, if any
• It is designed mostly for a single concurrent user per installed application
• Performs any required processing locally

Performance issues can be easily monitored and diagnosed and are usually related to the data that is being processed. So, sometimes it might be required to get a copy of the data that causes the performance issue so we can replicate the performance issue in our environment.
Getting Started with Performance Tuning

Thick client application – client-server model
This application has the following main characteristics:

• Thick client is an application that is running on a user machine (personal computer, tablet, phone, and so on), and is connected to a remote machine/server
• It is responsible for GUI and some local processing; it is connected to remote servers mostly for data synchronization (retrieval and persistence)
• It could be an application, applet, Web Start application, or even a widget application
• The server side could be a web application
• Examples of this type of applications are e-mail client, chat application, and so on
• It is usually designed for one user at a time per single device

Performance issues are distributed and investigation could involve both the client and server, and the more functionality the client has, the more value we gain from client application profiling.

Thin client application – web-based model
These applications has the following main characteristics:

• The client does not consume much of the local device hardware and is not installed on the user's machine; users mostly access these applications using browsers on different devices (PC, tablet, phone, and so on)
• The application itself is running on remote servers and these servers are responsible for most of the application functionality
• Processing is done on the servers and only some part of processing can be done on the client side for the presentation layer (like JavaScript code)
• Examples of this type are any browser-based applications, such as e-mail, website, search engine, online tools, and so on
• It is designed typically for multiple concurrent users

Performance issues mostly exist on the server side and are less common on the client side, for example, JavaScript code.

For More Information:
The following diagram illustrates the difference between one-tier and simple multi-tier application models:

Some web applications are deployed locally and used as standalone applications. This concept differs somehow from the general concept that we have discussed here, where web applications are typically hosted on remote servers and clients access those servers using different browsers.

Dealing with web applications' performance tuning

As we are targeting the performance tuning of Java Enterprise Edition 7, the kind of applications that can be developed by Java EE 7 can fit into either web applications or the server side of the client-server model; both will be handled in nearly the same way from the performance tuning perspective.
If the client is our browser, then some additional tools to analyze the traffic and JavaScript code are required. If it is a standalone application, then almost the same tools that we will use on the server side can be used on the client side as well.

**The two dimensions of web applications' performance tuning**

When we deal with such applications, we need to think in two dimensions: vertical and horizontal. So, we start with the horizontal dimension to spot the issue's location, then we go vertically through all the layers in this location to point out the root cause.

**Horizontal dimension (node-to-node)**

From the client to the backend servers, we need to examine each node/component in the flow to spot the root cause of the issue.

Having each node's performance reports or access logs can help us in isolating the bottleneck node in our application.

**Vertical dimension (intranode)**

In every machine/node in the application, we should check the node from the top to the bottom, passing through all the possible layers.

We definitely do not need to go through this systematic approach in all cases, but we need to understand the complete approach in handling performance issues. After gaining more experience, we will bypass certain components according to the nature of the issue that we are working on.

"Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is."

- Rob Pike

In the following diagram, we have explained the horizontal nodes and some of the possible vertical dimensions in each node:

For More Information:

Exploring vertical dimension nodes in horizontal dimension nodes

Now, let's go through the vertical dimensions in some of the application nodes in brief to explain what we need to look into and the possible tools for that.

**Client side**

On the client side, we have to mainly focus on JavaScript code and the loading time of different resources in our client; rarely do we need to consider Cascading Style Sheets (CSS).

The good news is that all modern browsers have integrated tools to use for this troubleshooting and they are usually named developer tools. Also, they have additional useful plugins that can be used for the same purpose.

We can also use external tools that have plugins for different browsers like DynaTrace or Fiddler.

**Network components**

Checking the performance of network components is an essential part of any performance investigation. Monitoring and checking the traffic through these nodes and their security configurations are important as they could be potentially the root cause of slow application response. The most important network elements include router, firewall, load balancer, and proxy.
HTTP servers (web servers)
Most enterprise deployments tend to have dedicated HTTP servers (like Apache HTTP server) to serve the static enterprise content and assets (help pages, different images, CSS, and JavaScript files). Being a part of the enterprise application architecture, we need to consider checking the server status, server logs, and machine's overall performance during our performance troubleshooting.

It is not common to see issues in HTTP servers, so it might be considered a routine checkup before excluding them from our troubleshooting plan. All HTTP servers have instructions to tune them for the best performance optimization. The operation/deployment team needs to apply these recommendations for the best performance outcome. Most of the performance tuning aspects in these servers are simply configuration parameters that need to be adjusted according to our application type and performance needs.

One example for non-configuration tuning elements is the memory size, which is very critical to HTTP server performance. We need to ensure that sufficient memory is allocated because memory swapping increases the latency of a server's response to user requests.

Application servers
As we clarified earlier in the enterprise application layers diagram, an application has many layers; starting from code up to the operating system. Most common issues are in the application code layer, but we need to ensure that all the other layers are performing as expected; all these layers have supported guidelines and best practices to tune and optimize them, for example, JVM.

We need to have monitoring tools in place including operating system monitoring tools, application server monitoring tools, JVM tools, sometimes framework tools, and virtual machine tools if the deployment has to be done over a virtual machine.

Database servers
Monitoring database servers and getting different database reports or logs such as the Oracle AWR report are essential in the identification of the performance issues' root cause. Let's assume we have a query that retrieves data from a big database table where there is no index used in that table. Checking the database report will show this query listed at the top of slow executing queries in that report.

We can then get an execution plan for that query to identify the root cause of its slow execution and prepare a potential fix.
Chapter 1

Checking the status of database servers (operating system), data files, and the underlying hardware is an essential step in our investigations.

**Automatic Workload Repository (AWR)** is a built-in repository (in the SYSAUX tablespace) that exists in the Oracle database.

At regular intervals, the Oracle database takes a snapshot of all of its vital statistics and workload information and stores them in the AWR; it is first introduced in Oracle 10g.

**Middleware integration servers**

All big enterprise applications are just part of bigger architectures in which different applications are plugged into the integration component, that is, a middleware application or service bus to facilitate the exchange of different messages or data between these integrated systems.

Continuously monitoring the performance of this critical layer is a core performance tuning activity. Of course, we always have a scope to work in, but the integration layer should be neutral during our work; this means all integrated communication shouldn't impact our application's performance.

Also, we should be able to get performance results for different integrated components.

Some applications do not have the integration layer in the testing environment and they use **stubs** instead to simulate the response. The stubs latency should be updated periodically with the actual live systems results, otherwise the testing environment won't simulate the production's actual response time.

If the middleware layer is not optimized for a good performance, all the integrated systems will suffer from bad performance, and if not well monitored, most of the effort of tuning the integrated applications will be incorrectly directed.

One example of a poorly performing middleware application is overutilizing the hardware by deploying too many JVMs for middleware applications; this is usually unnecessary scaling as middleware applications are already designed to connect to too many applications efficiently.

Another point to consider here is that due to the critical nature of this system component, it needs to have some sort of redundancy and fail over features to avoid taking the performance of the whole enterprise application down.

For More Information:

Operating system and hardware

Hardware could be a root cause of our performance issues especially when the capacity planning is not well considered. Pointing to hardware issues is usually done after excluding all other factors.

We also need to take the utilization pattern into consideration as it could point to possible cron job activity.

Cron job is a time-based job scheduler that gets executed according to the configured schedule table, for example, cron table in Linux or schtasks in Windows. It can be used to archive, back up, load data, scan viruses, and so on.

Let's take some hardware readings and analyze them.

CPU utilization

Web applications usually consume low CPU power per transaction since during each transaction, the application-user interaction includes thinking for a response, selecting different options, filling application forms, and so on.

If the transactional CPU utilization went high, we can suspect a running cron job, for example, an antivirus that is running (pattern is important here), high traffic load (due to incorrect capacity planning), or a common algorithmic logic issue that needs to be fixed.

With low CPU utilization, we can consider using more asynchronous components to increase the efficiency of utilizing the processing power of the machine.

Network traffic

Network bandwidth utilization is very critical in a production environment and it would be funny to forget that automatic application updates are switched on because it consumes the network traffic in an undetectable manner.

It could also point to architecture issues, missing local caching, backup job, and so on.

For More Information:
Memory usage
After excluding memory issues like application memory leakage, we need to check the JVM memory configuration. Missing memory tuning for our JVM is not expected in a production environment but it is worth considering it as a part of our investigation. Also, check the different components of memory consumption and the total free memory left.

Taking the decision to upgrade machine memory is not the only solution; we can also consider moving some components into different boxes, for example, moving certain services, caching components, or even the database server to another machine.

With low memory usage, we need to consider caching more data to speed up the application by utilizing the available memory.

Storage I/O performance
Storage read/write speed is critical in a production environment as I/O operations are usually the most time-consuming operations in relation to application performance. We need to consider using high-speed storage with a good percentage of free space for the running applications.

The storage performance issue becomes more severe when it affects the database servers.

In Chapter 9, Tuning an Application's Environment, we will discuss in detail the different tuning and optimization options for some of these nodes.

Summary
In this chapter, we discussed the art of performance tuning and its different aspects. We defined six basic components of this art in relation to the Java enterprise edition. We discussed the performance issues, and classified them into different types according to their discovery time and the responsible software engineering phase.

We explained at a high level the tactics that we need to follow while dealing with performance tuning including both proactive measures like defining processes and reactive measures like using the diagnostic and monitoring tools in performance troubleshooting.
Getting Started with Performance Tuning

We also focused on how we need to think when we have to deal with performance issues, from our personal behavior, process-wise, and knowledge-wise.

In the last section of this chapter, we dissected our strategy when dealing with different types of Java applications, and took a detailed approach when dealing with enterprise application performance tuning by using both a horizontal-oriented and vertical-oriented analysis.

In the subsequent chapter, Chapter 2, Understanding Java Fundamentals, we will pave our way for Java EE performance tuning by establishing a solid understanding of the fundamental concepts in Java EE 7 including recent changes in the Java Enterprise Edition 7, memory structure, garbage collection policies, and different Java concurrency concepts, all being an important part in our performance tuning routine.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.