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

About the Authors
Arnold Johansson is a versatile information technologist with a true passion for improving people, businesses, and organizations using "good tech".
As an early adapter of the Java language and its growing ecosystem, he is an outspoken proponent of secure Java Enterprise solutions and real Open Source software.
After nearly two decades as an IT consultant in many levels and verticals, Arnold now focuses on leading organizations on an architectural stable and efficient path of excellence.

Mother and Father, thank you for everything!

For More Information:
Anders Welén embraced the object-oriented techniques of the Java language early in life, and later evolved to Java Enterprise specifications. As a true believer and evangelist of Open Source, he naturally discovered the JBoss Application Server, which led to years of providing expert consultation, training, and support for the JBoss and Java EE infrastructures.

As a result, Anders has seen a lot of both good and bad architectures, software solutions, and projects, most of which were a struggle from time to time due to performance problems.

Whenever Anders, through presentations, consultation, training, and (in this case) a book, sees that what he's trying to explain is getting through and the audience is picking up on it and adopting it for their own challenges, it gives him a warm feeling inside.

To my parents, Stig and Birgitta, for teaching me to be curious. To my love, Lotta, for her support while I was writing this book.

And finally, to the French language lessons in school, as I chose, and therefore discovered, Computer Science to avoid them.
WildFly Performance Tuning

Buying a new suit isn't easy. Hopefully, you know why you need one. There might be a certain occasion, such as a wedding or party, or perhaps you just need to look sharper at work. Whatever the reason, there are likely to be some inherent requirements. Cloth, colors, patterns, the cut and the placement as well as the number of buttons are examples of factors that will all depend on environment and time of usage. The thoughtful buyer will take all of these factors, and possibly more, into consideration before making a selection.

Getting something cheap can be acceptable and cost-effective in the short run. For longer commitments, however, it will certainly be more financially viable (and probably more aesthetically preferable) to get the quality of a well-designed, tailored fit, especially as this might even be customizable as your needs (size) change.

Should you need to sit a lot in your precious garment, it would be wise to double up with at least an extra pair of pants. When wear and tear, or some other reason, makes you drop a pair, it is good to quickly have another on standby.

Get the requirements sorted, find the right design, try it out, and tune it to fit new needs when needed. If you find a wrinkle, make sure to iron it out and try it on again. That is what you must do during the lifecycle of a suit and an IT system alike.

Let's go and get dressed for success with WildFly!

What This Book Covers

Chapter 1, The Science of Performance Tuning, talks about what performance tuning is all about and how it can be applied within an organization.

Chapter 2, Tools of the Tuning Trade, introduces some useful Open Source tools to use when performance tuning anything covered in this book.

Chapter 3, Tuning the Java Virtual Machine, covers what the engine of Java is and how to tune it as well as all other Java-based applications.

Chapter 4, Tuning WildFly, explains what can be tuned in the WildFly Application Server.

Chapter 5, EJB Tuning in WildFly, talks about how Enterprise JavaBeans can be tuned.

Chapter 6, Tuning the Persistence Layer, covers how to design an effective database as well as how to tune JPA and queries.

Chapter 7, Tuning the Web Container in WildFly, explores Undertow—the blazingly fast, new web container in WildFly—and discusses how it can be tuned to become even better.

For More Information:

Chapter 8, Tuning Web Applications and Services, covers numerous tuning tricks and tips surrounding the web applications and services based on Java EE.

Chapter 9, JMS and HornetQ, explains how JMS works and can be tuned in HornetQ, the JMS provider of WildFly.

Chapter 10, WildFly Clustering, explores tuning in a clustered WildFly, HornetQ, and Java EE components.

For More Information:
The Science of Performance Tuning

There are many definitions of science. However, the most common definition is the systematic gathering and organization of knowledge. This refers to knowledge that, apart from its original purpose, can be used for further explorations of even more, and possibly further detailed, knowledge. Gaining new knowledge often involves validating theorems or ideas through experiments and tests. When a test has been validated or falsified, some things can be changed, or tuned, and the test can be performed again in order to gather more knowledge about what is going on in certain environments or situations.

As we will see, this approach of searching for knowledge in science is directly transferable to the improvement of performance in IT systems. In our opinion, performance testing and tuning is a science that is possibly spiced with some gut feeling (which is often based on experience, so we would say knowledge possibly sprung from science).

In this chapter, we will kick off by defining some key terminologies and measures of performance. After that, we will turn our focus to the process of performance tuning and its place in the software development process and the software life cycle. This will include its iterative behavior and talking about when, where, and how performance tuning should be done in an enterprise stack.

Performance

In the world of Information Technology (IT), performance is often used as a generic term of measure. This measure can be experienced somewhat differently depending on the role an individual has.

For More Information:

A user of a certain application will think more or less favorably of the application depending on how fast it responds, or flows, from his or her individual perspective and interactions. For a developer, an administrator, or someone else with a more technical insight of the application, performance can mean several things, so it will need to be defined and quantified in more detail. These expert roles will primarily need to distinguish between **response time**, **throughput**, and **resource utilization efficiency**.

### Response time

Response time is normally measured in seconds (and is often defined with some prefix, such as milli or nano) and relates to the sum of time it takes to send a request to an operation, the execution time of the operation in a specific environment, and the time taken to respond to the requester that the operation has completed. The request, the execution of the operation, and the response are collectively called a **roundtrip**: a there-and-back-again trip.

A typical example, (depicted in the following diagram), is a user who has filled out a form on a web page. When the user sends the form by clicking on the **Submit** button and sends the data in the form, the timer starts ticking. As the data is received by a server, the data populates a JavaBean in a Java Servlet. From the servlet, subsequent calls to other components such as other servlets and EJBs will occur. Some data will then be persisted in a database. Other data can be retrieved from the same or other databases, and everything will be transformed into a new set of data in the shape of an HTML page that is sent back to the browser of the end user. As this response of data materializes at the user's end, the timer is stopped and the response time of the roundtrip can be revealed:

![Diagram showing a roundtrip](image)

The response time, starting at t=0 and ending at t=x, and the roundtrip for a use case in a system.

Deciding what, or how much, should be included in a use case for measuring the response time, will vary on the test or problem at hand.

---

**For More Information:**

The total response time for the roundtrip of a system can be defined as the total time it takes to execute a call from an end user through all layers of the network and code to a database or legacy system, and all the way back again. This is an important and common value that is often used in service level agreements (SLA). However, it provides a far from complete picture of the performance and health of a system. It is important to have a set of use cases with the measured response time from various points in the system that covers its common and most vital functionality and components. These will be extremely helpful during tuning and when changes or problems occur.

It is important to remember that the roundtrip in a use case must be constant, in terms of start and stop points, between measurements. Changing the definition will render the measurements useless as they must be comparable!

So, what can affect the response time? In short, any change might affect the response time in a positive or negative way. Changes that you can perform as a technician in the software, hardware, and related infrastructure—code, configuration, hardware, network topology, and so on—of a system, will have their effect and not seldom other than you might expect.

With all these things static, there might still be changes that can make the response time vary. Here, we’re mostly concerned with the load on the system. As the load of the system increases, its response times will eventually rise as the system throughput decreases.

When an application performs work, that work is often triggered by external or internal clients. This will require resources, such as CPU, volatile memory, network, and persistent storage. The level of this utilization is the load of the system. Load can be measured for one or several of these resources.

Take, for example, an increasing number of users and the interactions they do in a system. The increase in system transactions could eventually exhaust available connections of a pool. The excess of transactions would need to be queued for released resources or even timeout. This then turns into a bottleneck, where the system isn't able to handle the increasing number of transactions quickly enough.

**Throughput**

The load a system can manage is coupled to the measure of the system throughput. Throughput is generically measured in transactions per given time unit, where a transaction can be a task or operation, or even a set of operations that act as one.
The time unit is often measured in seconds but can be significantly smaller (milli, nano) or bigger (minutes, hours, and so forth). Commonly, throughput is denoted as transaction per second (TPS).

Here, a transaction or operation can be of any size such as a small computational function or a big business case spanning over several components or systems. The size is not of importance but the amount of operations is.

An alternative measure of throughput that is often used is the amount of data transferred per second, such as bytes per second. Just like an SLA often has one or more stated response times for a set of use cases, throughput normally also has TPS values for the system as a whole and possibly, some for subsystems/components that are important from a business perspective.

From a technical or IT-operations perspective, it is equally important to know what throughput certain systems or subsystems regularly have, and at what levels these might start to have problems and failures. These will be important indicators for when upgrades are in order.

With a focus on Java EE, it is important to remember that the Java EE specification, and therefore, most application servers that implement it, were designed for overall throughput and not guarantees about response times.

**Utilization efficiency**

Low response times and a high measure of throughput is normally what anyone wants from a system, as this will help in keeping customers happy and the business thriving. If money is of no concern, it might not be a big deal from a technical perspective to have a lot of hardware and using more resources than needed, as long as business is booming.

This poor efficiency will, however, invoke unnecessary costs, be it to the environment, the employment force, development time, system management, administration, and so on. Sooner or later, a business that wants to be or stay profitable must have an efficient organization that can rely on cost-efficient IT departments and systems. This includes utilizing available resources in the most efficient way while keeping the customers happy. It’s a balancing act that business and IT departments must do together.

Having a bit less computational force, memory, and IT staff available might (among many things) cause higher response times and worse throughput. Consequently, software must be more efficient on available hardware. To make the software efficient, we need to test and improve its performance.

For More Information:
Scalability
A system needs to be able to handle an increasing load in order for the business to stay attractive to customers. The response times need to be kept down for each individual user and the total throughput needs to increase as the amount of transactions increase. We say that the system needs to scale to the load, and scalability is the capability that the system needs in order to increase total throughput.

When a system needs to be scaled, there are two major disciplines that can be followed: **vertical scaling** and **horizontal scaling**. Vertical scaling (or scaling up), as shown in the following diagram, involves adding more hardware resources, such as processor cores and memory, to an existing computer. This was the prevalent way to scale in the days of the mainframes but is used to some extent even today as virtualization has gained momentum:

![Vertical scaling diagram](image)

Vertical scaling involves adding more resources to an existing computer.

Horizontal scaling (or scaling out) involves adding more computers that are connected through a network. A simple example of horizontal scaling is shown in the following diagram. This has been the concept of most computer topologies for several years and has gained enormous momentum with cloud services and big data:

![Horizontal scaling diagram](image)

Horizontal scaling involves adding more computers to a networked collective of computers, such as a cluster.

For More Information:
In general, adding more resources to a single computer becomes more expensive than adding more computers at some point. The single computer will be of low volume and will often be very specialized, as it needs to have an advanced and expensive architecture to handle a lot of processors and memory, whereas the many cheap computers can be simple, off-the-shelf products. The single computer will be a *better* computer standing next to any of the cheap ones, but at some point, the grand number of cheap computers will collectively be cheaper, faster, and thereby, better than the expensive one.

In the extreme, having just one computer can be hazardous as it will be a single point of failure. Using one computer will, however, be easier from an administrative point of view. There will only be one place to make changes or configurations. It will also be easier from a developer's point of view, as the programming model won't have to deal with many of the more complex scenarios that a distributed model can require.

As with all things, there are pros and cons with the two different types of scaling. There are also several factors that bridge the gap between them. The single computer in the vertical scaling scenario is seldom a single one. It is most common to have at least one backup server. In the horizontal-scaling scenario, the programming model has been simplified thanks to modern enterprise frameworks and the topology of computers can be adapted to let each cheap server in the network work on its own without the need (and complexity) to know about the rest.

### Performance tuning anti-patterns

To tune the performance of a system means to improve one or more of its measures of performance. The question is how, when, and where should it be done? This is an area of great underestimation and major misconception. In the subsections that follow, we'll give you some examples of common mistakes and problems. Try to avoid them, or help out by correcting them, whenever and wherever you can.

### The one-off

For many not-so-developed or understaffed organizations, performance testing with some possible tuning is more or less a one-off, something done just before an application is shipped out to production.

By only performing the testing and tuning at this point, the amount of work, if done properly, is much higher and much more complex than if it was done iteratively during the development of the application.

---

Naturally, an organization must keep within its financial limits, but doing performance testing just before a release is very hazardous. What will happen if an application turns out to not live up to the expected and necessary measures in production? It would clearly be very bad for business.

**The wrong team**

Very often, performance testing and tuning is run by staff that lacks the knowledge of how this testing and tuning should be performed. Getting the right individuals, in terms of competence, on board in the testing and tuning team is crucial. This can vary but normally involves **quality assurance (QA)** staff, experienced performance testers, and technical staff, such as architects and developers that have actually been involved in creating the system under test.

**The lack of mandate**

Even though the quality value of a system's performance has grown to be relatively recognized in most organizations today, there are still places where staff responsible for performance-related tests and tuning lack the voice or mandate to enforce proper quality tall-gates.

**The clever developer**

As passionate developers, we like to be clever. Making our code run smoothly is satisfying, and the performance improvements we make can give us a feel-good boost and self-confidence. That is great, but it's often not very meaningful to just (over)optimize our own functions or components. As developers, we won't know for sure how much or what parts of our code will actually be executed in such an amount that it will need performance tuning. In the long run, it can even hamper the performance as our optimizations might cause problems in other places of the system as a whole.

So, performance tuning is something that an organization as a whole should take seriously. It should be done iteratively and handled by a competent team with complementing skills, experiences, and mandate. As it's such an important factor for today's businesses, it should have a given place in any organizational process map.
Software development and quality assurance

Back in the days, when software development started to be structured and development teams grew, the waterfall methodology ruled. Today, that methodology has mostly been replaced by agile counterparts that include highly iterative approaches to the work. What has changed is the iterative and shortcut behavior among the tasks involved along with their iterative frequency. It is not uncommon to perform several iterations per day in modern development teams.

No matter what the methodology is, we perform some kind of analysis (requirement, architectural) in software development from which design and implementation phases follow. After, and often during the implementation phase, unit tests are run.

The unit test should be on a functional level, verifying the smallest building blocks in code, such as specific functions or methods within a class. These tests are normally run by the individual developer and are also advantageous to automate in order to run during daily/nightly builds.

Higher levels of tests include the following:

- **System tests**: These tests are used to verify a system as whole
- **Integration tests**: These tests are used to verify integration points between two or more components or systems
- **Acceptance tests**: These tests are used for the final verification by the product owner before or during the deployment in a production environment

Most of these tests should be automated and run with live data as soon as possible. Note that both combinations and other variants of tests can exist in different organizations depending on various needs, organizational, or other inherited reasons.

All these types of tests are commonly part of a well-run business' QA process, but they are also heavily entwined in the software development process as the knowledge and cooperation from both IT and QA staff are required. This is all good, but what about performance tests and performance tuning?

Naturally, performance testing should be included as a compulsory step in the software development process and the results thereof should simultaneously be an integral part of the QA process. The ownership might be arguable, but the important thing is that it gets done, and gets done well. The exact location of when to do performance testing will, however, need a bit more discussion.
Software development with performance focus

Let's first revisit the major steps of the software development process in more detail with a healthy focus on performance and some quality. Remember that these steps are run iteratively, with possible shortcuts, and sometimes, with very short iterations!

Some organizations may also define the process a bit differently, with some steps included in other processes such as the requirement and QA processes. The following diagram shows us a common version of the software development process from which we will discuss its different phases. We will, however, not talk about the acceptance testing and deployment phases in the process, as they normally won't have any direct impact on performance tuning.

A generic version of the software development process with performance tuning.

Analysis

Creating high-quality software should always begin with some thorough requirement analysis. This is often very focused on the business functions and their values, but it is also very important to pay attention to the architecture of the software itself and its required performance.

It is important to identify a set of situations that will occur in the system and turn them into structured scenarios or use cases. These use cases need to be measurable and their values need to be assessed from both business and technical perspectives. Not all use cases need to have their performance assessed, but for those that need to, deciding what types of benchmarks to use are important.

Some common performance-related questions that should be answered during the analysis phase are:

- How many concurrent users should the system as a whole be able to serve and what minimum response times are required in different situations?
- What levels of different software, hardware, and network resources must the various parts of the system have at their disposal in order to run smoothly?
- Which information and level of audit is needed in different scenarios to uphold legislative, business, or operation requirements?

For More Information:
From the preceding questions, it should be clear that the software requirements span not only the business-related functionality but also nonfunctional requirements, such as security and logging, as well as estimates of hardware and network resources. All of these can, and will, affect performance.

**Design**

Everything related to the software structure, and the software itself, is structured here and defined in more detail. The overall architecture should be set for all major components. Tiers of both hardware and software are detailed to fit and adhere to the architecture and various requirements. Databases and data structures at different levels are to be designed wherever possible. The efficiency of candidate algorithms and libraries should be evaluated.

In short, the architectural decisions and design details must constantly be weighted in performance.

**Implementation**

If the previous steps have been performed properly, implementing the software source code with standard configurations of the system can be quite straightforward. There should be information and decisions about what use cases and functions should be paid special attention in terms of performance. Utilizing known best practices and experiences (such as the ones mentioned in this book) should also be in a developer's toolbox.

Try to not do to any overzealous tuning here though, as it might be useless and possibly even counterproductive for the system in large and not very cost-efficient.

After, and during, the implementation phase, there should be some testing performed. Normally, the amount of testing increases as the software gets closer to production. Unit testing should be performed pretty much all the time and is actually tightly merged with coding in the implementation phase. However, system and integration testing will not be that useful until the software reaches some minimal level of testing maturity.
Performance testing and tuning

The new kids on the block in the software development cycle are performance testing, and its crafty cousin, tuning. Performance testing and tuning can be performed in pretty much every iteration during the software development of the system. There must be some reasonable need for it though, and it should be performed in a controlled environment with competent staff.

A performance test within a development iteration might focus on individual functions or components of the software being developed in order to verify that design decisions are sound. These isolated tests can however, never replace a complete and more realistic system-wide performance test.

Doing more complete performance tests on the entire system won't normally be useful until the later iterations of development. Naturally, it should be performed before deploying the system into production. However, remember to leave plenty of time to correct any faults or unreached requirements, so test earlier rather than later. Don't wait until the last iteration to do all the performance testing and tuning.

As it is often pretty much impossible to immediately live up to all requirements and foresee all factors that might affect a system, the performance tuning process must explore how different factors (configuration, environment, load, and so on) influence the different use cases of a system. Furthermore, factors are quite likely to evolve over time.

In order to structurally handle all these variables and variances while delivering a system that effectively lives up to requirements, performance tuning is (currently) best turned into a cyclical and iterative process in itself.

The iterative performance-tuning process

The tuning of a system involves testing in order to find bottlenecks in the system and eliminate them by tuning the system and related components.

Test cases and iteration

Before performance tuning actually starts, it must be determined what test-cases, or rather indicators, to focus on. This set of indicators might stay static, but after some work (iterations), it is also common that some that are deemed not to be as fruitful as expected are simply removed. Similarly, some new ones might be added over time, both to follow the evolution of the system as well as to widen or deepen our understanding of it. All this is done to improve its efficiency.
It is important that the bulk of test cases are kept between test iterations and even between product releases. This is done in order to be able to compare results and see how different changes affect the system. All these results will build into a knowledge base that can help when tuning and making predictions to both the system at hand and others in similar cases and environments.

The result of a performance test case is normally a measure of the response time, throughput, or utilization efficiency of one or more components. These components may be of any size or complexity. If the components are subcomponents of a larger application or system, the set of test cases often overlap—some covering the entire system and some covering the various subcomponents.

**Setting the baseline**

The first time a test is to be performed, there might not be much real data to lean on. The requirements should give, or at least indicate, some hard numbers. These values are called a baseline.

**Running tests and collecting data**

With the baseline set, tests are set up and run. As the tests execute, data must be collected. For some, only the end result might matter, but for most, tests getting data during the entire test run and from various points of measure will give a more detailed picture of the health of the tested systems, possible bottlenecks, and tuning points to explore.

**Analyzing the data**

Analyzing the data might involve several people and tools, each with some area, or areas, of specialty. The collective input and analysis from all these people and resources will normally be your best guide to what to make of the test data, as in, what to tune and in what order.

**Tuning and retesting**

After the analysis is done, the system will be tuned and the baseline will possibly be refined, and more retests will follow. Each of these retests will explore a possible tuning alternative.

It is vital that only one individual thing is changed from one test till its retest. Change more than one thing and you won't know for sure what caused any new effects that are seen. Also, consider that several changes might neutralize or hide their individual effects.

For More Information:

Remember that not only direct code or configuration changes to a system require a performance test. Any and all changes to a system or its environment actually make the system an aspirant for performance tuning. Also, note that not all changes require performance tuning to be performed.

As you can imagine, following all tuning possibilities and always doing complete retests could easily spin out of control. It would result in infinite branches of tuning and tests—a situation that would be uncontrollable for any organization. It is, therefore, important to choose carefully between the various possibilities using knowledge, experience, and some healthy common sense. The tuning leads should be followed one by one, normally starting with the one identified to give the most effect (improved performance or reduction of bottleneck).

The performance-tuning process is normally complete when all requirements are satisfied or when enough of an improvement has been reached (normally defined by the product owner or architect in charge). The tuning process is an iterative process that is realized by the major steps shown in the following diagram.

Apart from resolving bottlenecks and living up to requirements, it is equally important to not over-optimize a system. First, it is not cost efficient. If no one has asked for that extra performance—in terms of business or architectural/operational requirements—it should simply not be done. Second, over-optimizing some things (such as very minor bottlenecks) in a system can very easily, turn its balance off, thus creating new problems elsewhere.

For More Information:
The Science of Performance Tuning

**Test data**
Possibly one of the hardest areas of the software development and QA processes is related to having / finding / creating useful test data. Really good test data should have the following properties:

- It should be realistic and have the same properties as real live data
- It should not expose real user data or other sensitive information
- It should have coverage for all test cases
- It should be useful for both positive and negative tests

For tuning during load testing, the test data should also exist in large quantities.

As one can imagine, it requires a lot of work, and it can be very expensive to have a full set of up-to-date test data with all these properties available, especially, as the data and its properties can be more or less dynamic and change over time.

We highly encourage all efforts to use test data with the preceding properties. As always, it will be a balancing act between the available resources of an organization such as financial strength, people, and getting things done.

For load testing, the test data is normally generated more or less from scratch or taken from real production data. It is important that the data is complete enough for the relevant test scenarios. The test data does not, however, need to be as complete as for functional testing. Volume, is more important.

**Documentation**

Throughout the performance-tuning process, it is important to have a stable and complete documentation routine in order. For each iteration, at a minimum, all test cases with traceable system configuration setups and measurement results should be documented and saved. This will add to the knowledge base of the organization, especially if it is made available to various departments of the organization.

It can then be a force to efficiently compare data of old releases with over time or to make good estimates of hardware procurement or other resources. Never forget the mantra of performance tuning:

*Test, tune one thing at time and test again.*

For More Information:

The environment of performance tests

It has been mentioned that performance testing and tuning should be performed in a controlled environment. In a perfect world, this means an environment that is free of disturbance, production-like, and unchanged between tests.

Using the following three rules of thumb for your test environment, you will be as close to achieving the perfect environment as you can for your performance tests:

- **No disturbances**: The tests should not be disturbed by other events, such as the executions of batches, backups, unrelated network traffic, or similar factors, to ensure that measurements relates only to the system under test. In a production environment, there is likely to be external disturbances, but the origins of these are hopefully known, and the systems that generate them should have gone through separate performance tests. Simulations in performance tests of what happens to a system at the same time as an external disturbance runs might be useful for some situations, but it is seldom an exact science and is not recommended in general.

- **Production-like**: The test environment should also be as similar to the production environment as possible in terms of test data, configuration, resources, services, hardware, and network capabilities in order to have results that would actually be worth something as the system is deployed into the real production environment. To have a full-blown copy of the production environment available for performance testing is not always possible due to various reasons. When the test environment isn't quite up to level with its production counterpart, it is important to be aware of the differences and to be able to extrapolate any test results. Just be very careful to trust any estimates you make about the results in a different environment.

- **Unchanged**: The test environment must stay equal between iterations of the same test and preferably for all tests. This intertest equality of the environment is needed in order to make reliable comparisons of the results from repeated tests. The exception to this, naturally, is when some part of the environment itself is required to change as part of tuning. Then, only one thing per test run can change and it must be thoroughly documented.

The software life cycle

After a system has successfully gone through the last phases of software development (including performance testing, tuning, and acceptance testing), it will be deployed in production where its hopefully long and successful life will begin for real.
Upgrades

Over its lifetime, the system will most likely need to be upgraded for one reason or another. Upgrading might involve changes to the hardware, code, and configuration. Before this upgraded system is put into production, it should be as thoroughly tested as it was when it was first released in order to ensure that it will meet old, and any new, requirements. Naturally, this includes performance testing and tuning, when needed.

Metrics

During its life in production, a lot of things about the system will be of interest to the business, QA, and the different IT departments. Some important questions that need to be addressed among the different instances could be:

- **Business**: What use cases are actually utilized and to what grade? For what reasons are important functions not used? Are they avoided due to poor response times, perhaps? Does the system and its components really give the expected **Return of investment (ROI)** or can there be optimizations made?

- **QA and IT**: Are the error rates under control? Is the hardware utilization actually in alignment with what is estimated or is there need for more or less of something? What about the response times and usage of components, caches, and other software resources? What is the health of the system at any given time?

Information to answer these questions and more can quite easily be answered by the system itself. Some information might be available for extraction directly out of the box from the system or from underlying resources, while others might need to be enabled by configuration or by more or less advanced instrumentation in code.

The information is often extracted/collected by logging or monitoring through a protocol such as SNMP (mostly used by hardware and operating system services) or by using an API such as the **Java Management Extension (JMX)** API.

WildFly exposes information about quite a few resources through JMX, and instrumenting your application code to expose values using JMX is very easy and powerful. JMX can also be used externally from a system to give it instructions such as clearing a cache, starting/stopping a service, and so on.

Quantifiable information from and about a system, regardless of how it is retrieved, is called **metric**. The various metrics can be useful for a single situation such as a monitoring alert for something going wrong. However, it is also important to collect metrics over time as a proof of living up to SLA and be able to do various analysis related to the business, quality, or technology.

For More Information:
Performance testing and tuning is one of the areas that can benefit hugely from having metrics available. It is, for example, very valuable during the design, or modification, of test cases and setting realistic baselines.

**Tuning an enterprise stack**

Tuning can be broadly divided into different categories based on the different layers of an enterprise IT environment. This environment is often called an enterprise stack and consists of the layers shown in the following diagram. We will now turn our attention to these layers one by one and discuss what tuning means and consists of in each of them, starting from the bottom:

![Layers of an enterprise stack.](image)

**Network**

Network tuning typically involves the configuration of various network equipment such as firewalls, routers, and network interfaces, but can also include verifying the use of the correct type of cables and connectors. This type of tuning is often initially missed during performance tuning, but in today's communication-heavy solutions, it is absolutely vital to have a network that runs smoothly and at its highest performance. Network tuning is also highly related to, and thus overlaps, hardware and OS tuning.

**Hardware**

Hardware tuning includes selecting the right hardware components—CPU, memory, discs, and so on—for a given system and its requirements. Shortage of memory will increase I/O operations. Slow disks might make databases and entire systems crawl.

Data encryption and other computing-heavy functions will require a relatively large amount of CPU. Often, the solution can be to just to add more or better hardware, but it is equally important that the hardware is well-balanced and plays well together.

**Operating System**

*Operating System (OS)* tuning is closely related to network and hardware tuning as it defines how the OS and hardware/network will cooperate and what restrictions will be enforced. For example, CPU time slicing, I/O behavior, and network access.

Through the OS, a lot of information can also be retrieved regarding the health of not only the OS itself, but also of the hardware and network.

**Java Virtual Machine**

*Java Virtual Machine (JVM)* tuning involves configuring the memory levels and the garbage collector of the JVM. Although modern JVMs are considerably more intelligent, effective, and advanced compared to older versions, they often still need a bit of love and application-specific tuning. Tuning a JVM can drastically improve the performance of the application that is being executed in the JVM. This tuning is, however, quite volatile as things can easily go wrong and create new bottlenecks and even worsen performance unless used in a very controlled way. More about this will be covered in detail in Chapter 3, Tuning the Java Virtual Machine.

**Middleware**

Middleware tuning includes adjusting various configuration parameters of the platform called *middleware*. This is done in order to make the platform and its services more optimized for the applications and its components that run within it. A middleware platform is often realized as an advanced application server; for example, WildFly. Others might be simpler and won't include as many services; for example, a web container like Apache Tomcat.

Some parameters and services of the middleware can be utilized by all applications, while others can be application specific. For WildFly, some configuration and services include thread pools, connection pools for EJBs, JMS (queues/topics) and databases, EJB component lifecycle management, and much more. All these configurations have default values that might be just fine, but they also might be tweaked in order to achieve magnitudes of improved performance. Middleware is arguably where most configuration-related tuning can be made in the stack, but more of this will be discussed in chapters to come.

For More Information:
Application tuning is first and foremost achieved by making a thoughtful design and writing good, efficient code. This also involves selecting the best algorithms and libraries for your specific application. If the original design proves to be insufficient, and other tuning types won’t solve the problem, the design or code might need to be redone completely or at least be improved in some way.

This can, for example, involve changing an entire platform, framework, or programming model, or it can involve just improving a specific function or pattern. It could also involve making better use of APIs or available resources. For example, by using the `StringBuffer` or `StringBuilder` classes instead of `String` or by improving the speed of database calls by using indexes. Application tuning, in terms of initial design and implementation of a system, is often not directly seen as tuning. However, creating a tuned application is, without a doubt, the most important type of tuning you can do. Think about it. If you make poor design decisions or write poor code, it will be really hard, if not impossible, to fix that by just tuning the hardware or JVM. It would also be quite expensive—both in terms of time and money—to make large design and code changes to a system.

As we have seen from the preceding text above and in the following diagram, tuning can be performed pretty much everywhere in the stack, and tuning in one place can and will affect things in all locations. Thus, having a broad and open-minded view about possible ripple effects of singular changes will aid you in making the best decisions.

Iterative tuning in the enterprise stack. Tuning is everywhere and everything depends on many direct and indirect relations. Here, WildFly is depicting the middleware. Also hardware- and network-tuning is included in the OS tuning.

For More Information:
Summary
In this chapter, we made the connection and discussed performance tuning as a science. We defined performance as measures and listed some of the most important ones used in IT—response time, throughput, and utilization efficiency.

We learned that the performance-tuning process is highly iterative, and that it is vital to only tune one thing at a time between tests. Here, we also specified the main place of the tuning process within the software development process and listed some common tuning anti-patterns.

Good quality test data and production-like environments are fundamental cornerstones of testing in general and for performance testing, this is no exception; it's quite the opposite!

After going through metrics and their inherent value made available during the life cycle of a piece of software, we finally talked about the tuning possibilities available in all the layers of a complete enterprise stack. This is a stack that encompasses many software layers (such as an application or system, the middleware, and the operating system) as well as hardware and network equipment.

To put the theories of this chapter into practice, we will need a powerful set of supporting tools. Moving on, this is exactly what we will be looking at in the next chapter.

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.