Plan the app and its test

The first step is to plan the application and the performance testing. Whether you're creating the application from scratch or adopting the code written by someone else, you must understand what the software should do. You must identify where the data comes from, where it must end up, and how the software needs to transform that data along the way.

This plan should be careful to anticipate all contingencies when possible, but it doesn't need to be overly thorough. Everyone should understand the main parts of the software and the way that data flows into, out of, and around the sections, but it's not necessary to read all the code or understand each section perfectly. Any detailed knowledge of a section can be acquired if it's clear the section is not performing as expected.

Pro tip: Think ahead but don't use this as an excuse to delay.

Identify potential delays

Just as every chain has a weakest link, every piece of code has a slowest section. Sometimes the delay is obvious, such as in code that does a great deal of background processing—for example, a photo sharing website that immediately constructs thumbnails of every new photo. Image resizing is computationally expensive and it's bound to place more demand on the server than many of the other tasks.

Sometimes finding a bottleneck is difficult because it changes with time. Code that relies upon outside resources like databases or web services may run smoothly most of the time until the resource fails. Stochastic or seemingly random pauses in performance can be some of the most difficult to locate because they often can't be confirmed. When you look for them, they aren't there because the service is behaving correctly again.

Before adding logging functions or installing profiling code, start by sketching out the flow of data through the system. Chart where it enters, where it goes, and where it ends up. Mark the sections that you can test directly and note the sections which are out of your control. This flowchart will not offer guaranteed answers but it will be a road map for exploring.

Pro tip: Most of your troubles will be caused by the slowest part of your code. Identify it, plan for the delay, and try to minimize it.

Identify shared resources

While every developer dreams that their code might run in a pristine environment, one devoted to the execution of the code alone, this is rarely the case. There are often different tasks competing for same resources: the network, the RAM, the database, or parts of the hardware. Sometimes what has to be shared can be as simple as the background work of the operating system; other times it can be another web application that runs on the same server.

Finding the shared resources for competing tasks can be harder than ever thanks to virtualization. More and more applications look like they're running in their own separate computer, but they're really sharing a larger hardware environment and running as a virtual machine. Recognizing which other applications are running in neighboring virtual machines can be a bit tricky, and identifying when they're affecting your software is even trickier. It's common for virtualization layers to give unused cycles to machines with the most load, and this means your application could run faster or slower depending upon how much work the neighboring machines are doing.

Scott Oaks, author of Java Performance: The Definitive Guide, reminds us, "A systemic, system-wide approach to performance is required in Java environments where the performance of databases and other backend systems is at least as important as the performance of the JVM."

Pro tip: Taking a system-wide approach to performance means understanding and mastering shared resources, including virtual resources.

Watch complexity

Does your boss want the software to solve 242 different problems all at once? Do the users expect that the software will do as many things as a Swiss Army Knife? Be wary. Many performance problems begin when we ask the software and the machine to juggle too many things and produce too many different types of answers.

"Complexity kills," Ray Ozzie, one of the creators of Lotus Notes, likes to warn. "It sucks the life out of developers, it makes products difficult to plan, build, and test, it introduces security challenges, and it causes end-user and administrator frustration."

Brian Kernighan, one of the creators of the programming language C, suggests that "Controlling complexity is the essence of computer programming."
The best practice is usually to split your application into parts so that each can work independently. This makes planning simpler and analysis more effective. Alas, this is often easier said than done. Some applications seem inherently integrated and impossible to separate. But often a bit of analysis can identify parts of the application that can work independently. Separating them will produce smaller, less complex sections. The entire application may be just as complex, but the programmers can concentrate on the smaller sections with their part of the complex mechanism. Some are beginning to use the term "microservices" to describe an architecture made up of a number of small programs that answer questions independently.

Pro tip: Aim for simplicity. When possible, separate the application into smaller apps that are as independent as possible.

Watch database calls

When RebelLabs asked developers about their biggest performance headaches, slow databases were at the top of the list. Over half the respondents (54.8 percent) reported that their software was held back by a database that wasn't responding as quickly as expected.
But that wasn't all. Another subset said that the database was the root of the problem but that the software developers were partly to blame—38.6 percent of respondents cited "too many database queries." Their software was asking for answers more frequently than the database could handle.
These large percentages shouldn't come as a surprise. Modern databases provide complex solutions for keeping data organized. When users update records or interact with the application, they end up adding or updating rows of data and this will force the database to update indices that it uses to quickly locate particular rows. In many cases, the databases do the bulk of the work and execute the bulk of the machine instructions in the stack. Keeping the data safe, collated, and quickly accessible is a big challenge for your stack, which is why planning the interaction with the database is often the biggest job for anyone looking to maximize performance. More on this in the next tip.

Pro tip: Databases can be slow, so don't compound the problem by hard-coding more database queries than you absolutely need.

Watch the interaction with the database

Databases can run slowly for a number of reasons. Sometimes they answer requests from multiple servers and they are pulled in too many directions. In many architectures, the database is the central clearinghouse that synchronizes the work of multiple servers. The load can skyrocket when all the servers start sending requests and transactions at the same time.
In other cases, the queries are more complex than the database can handle quickly. If a query includes multiple join operations or it demands complex sorting, the answer may be slow to arrive because the database needs to juggle too much information. It must often pull data from multiple tables into memory to complete these processes and when the operations are large and the data overflows the cache, the operation becomes limited by the speed of the hard disks.
In many cases, it makes sense for the developers and the database administrators to reexamine the queries and decide whether they need to be as complex as they are. The developers need to plan their requests and do their best to streamline them to prevent overloading. The database administrators must anticipate these needs and structure the database to be able to deliver the requests.
There are many options for simplifying the structure and adapting it to the needs of the users. Sometimes the joins can be postponed or eliminated by presenting a smaller amount of data to the user who will then drill deeper if the extra information is needed. Sometimes the data can just be dropped altogether because the users don't really need it. Paying attention to the complexity of the queries is one of the simplest ways to reduce the load on the database server.
Choosing the right size for the database and the server is also important. The best practices are evolving quickly as more complex and elaborate databases enter the marketplace. Sometimes the correct solution is to replace the database server with a more powerful machine with more RAM. Other times, it's better to create a cluster of N database servers that can share a larger load. Many of the newer clustering databases make it possible to solve problems of heavy load by adding more servers.

Pro tip: The interaction with the database is one of the greatest sources of delay.

Study how you use databases

It's not always the database's fault. Even if your profiling shows that the database calls are the bottleneck for the system, it doesn't mean that the database is to blame. Nor does it mean that everything could be improved by throwing more money at the database layer.

Many Java programmers quickly discover that the real bottleneck isn't storing the data itself, but simply establishing a connection to the database. The Java database connectivity (JDBC) layer, one of the common ways for interacting with a database, must authenticate the user and reserve I/O resources. If your application is jumping through all these hoops for every record it stores in the database, it's going to be much slower.

The best practice is to create a pool of connections that remain open with the application using a library such as Apache Commons, Apache Excalibur, or the ones from Oracle. When a thread needs to store data, it grabs an open Connection object, sends along the data, receives confirmation, then returns the Connection object to the pool where it can be reused. This process reduces the overhead spent on constant authentication and initialization.

Setting up efficient connections with the database is surprisingly complex and often the source of many bottlenecks. It often makes sense to use some of the built-in persistence models. There are several included in the standard versions of Java and J2EE and a number of good open source and commercial packages. These offer highly optimized mechanisms for interacting with databases and you can benefit by relying upon their testing.

Pro tip: Use good libraries for connecting with databases efficiently.

Study the network architecture

Many programmers often overlook the network hardware and topology. They didn't write the code for it, so they take it for granted. But in many cases, a slow network or a bad topology can do more to slow down an application than bad code. If the web application can't communicate effectively with the database, it won't have anything for the users.

"Databases are the fountain from which data streams to all kinds of applications," explains George Reese, the author of Java Database Best Practices. "It is therefore critical—especially for high-volume database servers—to allocate the necessary bandwidth to database servers. It's not uncommon for database servers to be connected to the network through multiple fiber-based gigabit Ethernet ports."

When planning your performance tests, be aware of what other servers share your network. If they're dominating the data flow, they may be slowing your servers in a way that seems imperceptible. All the servers under your control might appear to have low loads, but the entire application could still be slowed.

Reese spells out the best practice for design: "A good network architect structures the topology of the network to minimize packet collisions and bring the database network as close as possible to the other networks that rely on the databases."

Pro tip: Make sure the network can handle all traffic between components.

Be willing to compromise on demands

Everyone wants their data to be stored perfectly but that can be expensive when the data model is elaborate. If the database is going to juggle multiple tables and keep the data in them consistent, it must work much harder to ensure that the results are accurate even after a power failure or other catastrophe.

Sometimes the best practice requires relaxing the expectations for storage in order to speed up response times. In other words, choosing a storage mechanism that's not the best in order to get the best speed.

These decisions are as political as they are technical. Will the users be willing to accept very occasional errors in return for faster performance at a cheaper price? If it's their bank account, the answer will probably be no. But if it's something disposable like a click on a social media website, the answer is often yes.

Pro tip: Look for opportunities to use less perfect transaction models to add speed.

Watch service interactions

Databases aren't the only common bottleneck. According to ZeroTurnaround's RebelLab survey, 12 percent of respondents reported problems with "slow or unreliable third party entities." In other words, just as Sartre proclaimed in his play *No Exit*, "Hell is other people."

The trouble is, we're often creating elaborate software packages by knitting together many smaller ones. Many developers speak of splitting their application into microservices, which are smaller programs tuned to answer one particular question. Each microservice often has databases buried inside it, which means that at its core, it shares many of the same issues with database calls.

Tracking microservices is often similar in spirit to profiling database and tracking response times, but without the same level of access. The database is often wrapped up by the service protocol and the best we can do is track the performance of the service. Is the average response time adequate? Can you identify which types of requests produce a slow response? The best practice is to track their failures and then design the final user interaction to reduce the effects of failure. If we can't fix the third-party service, we can prevent it from blocking the performance when it slows down. That is, we can separate the reliance on the different services so if one fails, it won't stop the others.
If the end product is a website, it's now common to load the different sections separately with AJAX calls. If one microservice is slow, the rest of the page will still load correctly and the user will be less inconvenienced.

Pro tip: Gathering data from other web services can be a source of delay. Watch their performance.

Watch object creation

Dealing with databases or services often requires treating them like a black box. You may not know what's going on inside. When you're dealing with your own Java code, though, you'll be able to look at the individual steps and watch for issues.
One of the most common problems with Java code is too many object creations. While the language is meant to be object-oriented and the best practice is to use this architecture to produce clean code, it's important to recognize that there's significant overhead to creating and disposing of an object. The memory must be set aside and then reclaimed when the object is released.
The RebelLabs study found that 10.6 percent of developers surveyed reported that their applications suffered from "excessive memory churn" and 17.9 percent had problems with pauses from garbage collection. In other words, too many objects are created and then thrown away.
Many Java programs can be sped up dramatically by reducing the number of objects created. If the data can be stored in a primitive type like an int, it's much faster than creating a wrapper class for each new result. Another common practice is to create an object once during initialization and reuse this throughout the run.
Many good profiling tools concentrate on tracking object creation and destruction. They can highlight the places in the code where objects are created and point you in the direction of where the code can be improved.

Pro tip: Object churn is a big source of Java performance problems.

Be stingy with string creation

Creating new strings of characters is slow for the same reason that creating objects is slow. Each string is an object and the system must allocate them again and again. To make matters more complex, each string is an *immutable* object because it increases the security of the system and makes threading simpler. This becomes a performance problem when gluing together many strings, which is a common practice in web servers. The code may be creating many string objects along the way and then throwing them out.
The best practice is to use flexible objects such as StringBuilder and StringBuffer when you're going to be constructing larger strings from smaller parts. In many cases, the compiler can be smart enough to replace simple concatenation ('+') in your code with StringBuilder objects.

Pro tip: Too much string concatenation can produce too many string objects.

Avoid creating and destroying too many threads

Creating and disposing of multiple threads is a common issue in Java performance. While the artful use of thread objects makes it possible to create software that juggles multiple users and many tasks at the same time, the thread objects themselves can take time to create and destroy.
If your application uses many threads to issue quick answers to many people, it can be faster to create a pool of threads so they can be reused without being destroyed. This practice is best when the amount of computation done by each thread is small and the amount of time creating and destroying it can be larger than the amount of computation done inside of it.

Pro tip: Use threads sparingly and only when necessary to avoid the cost of creation and deletion.

Profile early and often

There are a wide range of Java profiling tools that will track how quickly the code completes its project. The simplest measure the speed of code on one CPU while the most sophisticated can track an entire constellation of software packages

and how they interact. They can install themselves with the JVM, analyze all new code when it's loaded, and then track the important parts like database calls or service requests.
No matter which tool you choose—and there are many—it's important to use them often because they can offer deep insight into which parts of the code are taking the most time to execute. Concentrating on these sections is the simplest way to increase performance. Many programmers disagree on whether to use profiling early in the project because it can lead to spending too much effort on optimizing code that doesn't run very often.
If developers keep this waste of time in mind and avoid being distracted by optimizing code that isn't used very often, it's useful to start testing the performance from the beginning. It creates a good habit that will pay off at the end when the pieces start working together.

Pro tip: Profiling early identifies potential problems and design flaws.

Create mock variables

One of the biggest challenges for testers is creating a realistic set of data that will simulate how the software is expected to perform. If they don't do this, they may run the software successfully on their test rig and assume it's fine. As Vidiu Platon, the programming pundit, says, "I don't care if it works on your machine! We are not shipping your machine!"
A common best practice for Java developers is to create mock variables. In the simplest cases, these may just be variables filled with random numbers. But in more serious examples, the mock variables may be filled with historic data that's designed to simulate the most demanding load on the machine. Sometimes creating the software to generate an accurate set of test data is more complex than the software it is to test.

Pro tip: Mock data is essential for testing your code. Write routines to create mock data that mirrors the real values.

Watch the CPU load

One of the simplest profiling metrics to watch is the server's CPU load. Watching the average load over time reveals just how much computation the CPU is actually performing and this can help diagnose the problem when performance is sluggish.
Osama Oransa, author of Java EE 7 Performance Tuning and Optimization says that high CPU load on a server is often an indication that the server is working too hard. "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." He explains that much of the interaction is done with the client machine where the user fills out forms or selects options.
When the CPU load is low but performance is sluggish, the problem often lies with other machines. Slow database or service calls are the most common issue. But if CPU is high, Oransa suggests several suspects:

- A running cron job that is temporarily slowing down the server
- An antivirus or other back-end processing job
- High traffic load due to incorrect capacity planning
- A poorly written algorithm or badly described logical pathway

The solution is to use profiling to identify which of these issues is the actual problem. If it's back-end processing or a cron job, this can be offloaded to a second server that performs the high CPU load in the background. If the server's logic itself is slow, it makes sense to revisit the code to see if it can be improved or rewritten. If it can't, and it's an issue of load, then the solution needs to add more servers and use a load balancer to spread out the workload.

Pro tip: The CPU load is a crude but effective metric for diagnosing problems in your code.

Watch your own actions

Just as Erwin Schrödinger described how the act of measuring the results of a physics experiment can change the answer, we need to be aware that what we do can change the behavior of the code.
Scott Oaks writes about how he checked all the possible problem sources like the external resources, the system load, and the network. He came to the conclusion that he was causing the program to run slower and slower.
"The next most likely issue, therefore, was the test harness," he writes. "And some profiling determined that the load generator—Apache JMeter—was the source of the regression: It was keeping every response in a list, and when a new response came in, it processed the entire list in order to calculate the 90th% response time."
The best practice is to run tools like JMeter on other machines where the analysis can't slow down the code that's being studied. This works well for some web applications because the interaction naturally occurs over the network, but it may

not be possible for other code that interacts directly. When possible, try to keep the code in question as isolated as possible, as it would be in normal circumstances. When that isn't possible, be aware of how your testing can skew the results.

Pro tip: Make sure that your profiling code isn't the source of the delay.

Final thoughts

Performance engineering in a Java context is about much more than writing great code. You need to master the many dependencies and variables in play within an increasingly complex ecosystem. As noted earlier, the 17 best practices described here are all important, so they're not numbered or ranked in any way. I recommend you study the list and find ways to incorporate these practices into your own team and infrastructure.