
76ms
-49ms
-152ms
405ms
74ms
230ms
46ms
-625ms
175ms
102ms
10
-152ms
WEBSITE PERFORMANCE TESTING
HTTP response
HTTP request
SERVER

An Introduction to Performance Testing

The phrase "performance testing" can mean a great many things to different people in different scenarios, so I thought covering a few of the different types of tests may be helpful.

Performance Testing is generally an umbrella term covering a number of different, more complex test environments. I've also used the term to describe a very simple set of scenarios meant to provide a baseline for performance regressions.

Load Testing generally uses a number of concurrent users to see how the system performs and find bottlenecks.

Stress Testing throws a huge number of concurrent users against your system in order to find "tipping points" – the point where your system rolls over and crashes due to a huge amount of traffic.

Endurance/Soak Testing checks your system's behavior over long periods to look for things like degradation, memory leaks, etc.

Wikipedia's Software Performance Testing page has some very readable information on the categories.

You can also look at performance testing as a slice of your system's performance. You can use a specific scenario to dive down in to specific areas of your system, environment, or hardware.

Load, stress, and endurance testing are all that, but turned up to 11. (A reference to Spinal Tap for those who've not seen the movie.)

With that in mind, I generally think of performance testing in two categories: testing to ensure the system meets specified performance requirements, and testing to ensure performance regressions haven't crept into your system. Those two may sound the same, but they're not.

Performance testing to meet requirements means you'll need lots of detail around expected hardware configurations, baseline datasets, network configurations, and user load. You'll also need to ensure you're getting the hardware and environment to support those requirements. There's absolutely no getting around the need for infrastructure if your customers/stakeholders are serious about specific performance metrics!

Performance testing to guard against regressions can be a bit more relaxed. I've had great successes running a set of baseline tests in a rather skimpy environment, then simply re-running those tests on a regular basis in the exact same environment. You're not concerned with specific metric data points in this situation – you're concerned about trends. If your test suite shows a sudden degradation in memory usage or IO contention then you know something's changed in your codebase. This works fine as long as you keep the environment exactly the same from run to run—which is a perfect segue into my next point.

Regardless of whether you're validating performance requirements, guarding against regressions, or flooding your system in a load test designed to make your database server weep, you absolutely must approach your testing with a logical, empirical mindset. You'll need to spend some time considering your environment, hardware, baseline datasets, and how to configure your system itself.

Performance Testing: Laying the Groundwork

Performance testing isn't something you can slap together and figure out as you go. While you certainly can (and likely will!) adjust your approach as you move through your project, you do indeed need to sit down and get some specifics laid out around your testing effort before you begin working.

First and foremost: set expectations and goals

Ensure everyone's clear on why you're undertaking the performance testing project. If you are looking to meet specific metrics for delivering your system then you'll need to be extremely detailed and methodical in your initial coordination. Does your system have specific metrics you're looking to meet? If so, are those metrics clearly understood – and more importantly reasonable?

Define your environment

If those same metrics are critical to your delivery, then they will also need to be defined based on a number of specific environment criteria such as exact hardware setups, network topologies, etc. These environments should be the same exact environment you recommend to your customers. If you're telling your system's users

they need a database server with four eight-core CPUs, 32 GB of RAM, and a specific RAID configuration for the storage, then you should look to get that same hardware in place for your testing.

A tangential topic: it's happened more than once that a server and environment acquired for performance testing somehow gets borrowed or time-shared out to other uses. Timesharing your performance environment **can** be a highly effective use of expensive resources, but you'll need to ensure nothing, **absolutely nothing**, is being utilized on that server once your performance runs start – you have to have dedicated access to the server to ensure your metrics aren't being skewed by other processes.)

Agree on baseline data

Something that's commonly overlooked is the impact of your system's baseline dataset on your performance tests. You likely won't get anything near an accurate assessment of a reporting or data analysis system if you've only got ten or thirty rows of data in your database.

Creating baseline data can be an extremely complex task if your system is sensitive to the "shape" of the data. For example, a reporting system will need its baseline data laid out across different users, different content types, different date patterns.

Often the easiest route to handle this is to find a live dataset somewhere and use that. I've had great success coordinating with users of systems to get their datasets for our testing. You may need to scrub the dataset to clear out any potential sensitive information such as e-mail addresses, usernames, passwords, etc.

If using a live dataset isn't an option, you'll need to figure out tooling to generate that dataset for you.

Determine your usage scenarios

Talk through the scenarios you want to measure. Make sure you're looking to measure the most critical scenarios. Your scenarios might be UI driven, or they could be API driven. Steve Smith has a terrific walkthrough of a real world scenario that gives a great example of this.

Set up your tooling

Once you've got a handle on the things I've discussed above, look to get your tooling in place. Performance testing utterly relies on an exact, repeatable process. You'll need to do a large amount of work getting everything set up and configured each time you do a perf run. Avoid doing this work manually; instead, look to tooling to do this for you. You shouldn't rely on doing the setup manually for two reasons. One: automating setup ensures you'll cut out any chance of human error. Two: it's really boring.

Build servers like Hudson, Team City, or TFS can interface with your source control and get your environment properly configured each time you need to run a perf pass. Scripting tools like PowerShell, Ruby, or even good old command files can handle tasks like setting up databases and websites for you.

You'll also need to ensure you're setting up your tooling to handle reporting of your perf test runs. Make sure you're keeping all the output data from your runs stored so you can keep track of your trends and history.

Change only one variable at a time. Compare apples to apples!

It's critical you take extraordinary care with the execution of your performance testing scenarios! You need to ensure you're only changing one variable at a time during your test passes, or you won't understand the impact of your changes.

For example, don't change your database server's disk configuration at the same time you push a new build to your test environment. You won't know if performance changes were due to the disk change or code changes in the build itself.

In a similar vein, ensure no other folks are interacting with the server during your performance run. I alluded to shared servers earlier; it's great to share expensive servers for multiple uses, but you can't afford for someone to be running processes of any shape or form while you're doing your performance passes.

Profiling: Taking the simple route for great information

All the work above can seem extraordinarily intimidating. There's a lot to consider and take in to account when moving through some of the more heavyweight scenarios I already laid out.

That said, you can look to simpler performance profiling as a means to get great insight in to how your application is behaving. Profiling enables you to use one scenario, or a very small set, and see in a slice how your application's behaving. Depending on the tooling you can see results of performance back to the browser, dive in to performance metrics on the server (think CPU or disk usage, for example). You may even be able to dig down in to the application's codebase to see detailed metrics around specific components of the system.

Profiling is a great way to start building a history of your application's performance. You can run regular profiling tests and compare the historical performance to ensure you're not ending up with performance regressions.

Start small, start smart

As you've read, performance testing can be particularly complex when you're looking to ensure high performance, reliability, and scalability. You need to approach the effort with good planning, and you need to ensure you're not changing variables as you move through the testing.

Make sure your performance efforts get you the information you need. Start with small environments and scenarios, ensure you've clearly laid out your goals and expectations, and keep a careful eye out as you're running your tests.

What to Monitor (Plus Learning Resources)

What to Monitor?

Figuring out which metrics, measurements, and counters to monitor can be extremely daunting—there are hundreds of individual counters in Performance Monitor alone! In most cases you don't need anywhere near the entire set of metrics. A few counters will give us all the information you generally need for starting your performance testing work.

Most performance testing gurus will tell you just a few items will get you started in good shape:

- Processor utilization percentage
- ASP.NET requests per second
- SQL Server batch requests per second
- Memory usage (total usage on the server, caching usage)
- Disk IO usage
- Network card IO

If you're doing load testing you'll likely be interested in errors per second and queued requests. Often times soak or endurance testing will look to counters associated with memory leaks and garbage collection too—these help you understand how your application holds up over a long period of stress. However, those are different scenarios. The few counters mentioned above will get you started in good shape.

Jim Holmes has around 25 years IT experience. He is co-author of "Windows Developer Power Tools" and Chief Cat Herder of the CodeMash Conference. He's a blogger and evangelist for Telerik's Test Studio, an awesome set of tools to help teams deliver better software. Find him as @aJimHolmes on Twitter.

Where to Learn More?

Microsoft's "Performance Testing Guide for Web Applications" is somewhat older, but remains a tremendous resource for learning about performance testing. It's an extensive, exhaustive discussion of everything around planning, setting up for, executing, and analyzing results from your performance testing. The guide is freely available on Codeplex.

Steve Smith of NimblePros (now known as the Telerik Enterprise Services group) in Kent, Ohio, has been extremely influential in my learning about performance testing. Steve's been appointed by Microsoft as a Regional Director because of his technical expertise in many areas. He blogs extensively on many software topics and has great practical examples for performance testing. He also has an online commercial course offered through Pluralsight that's well worth checking into.

The website Performance Testing has a great number of references to performance testing information across the Web. The site lists blogs, articles, training material, and other highly helpful information.

Go! Get Started!

Spend some time planning out your performance testing effort. Make sure you work HARD to only change one variable at a time. Don't get flooded with information; more often less information can be more helpful at the start.

Performance testing is a tremendous asset to your projects, and it can also be an extremely fun, interesting, and rewarding domain to work in.

Go! Get started!