SUBJECT7
QUALITY IS A CULTURE

How to Start Performance Testing

By Matthew Heusser for Subject7

Sometimes when people talk about a discipline, they make it harder than it needs to be.

This does not have to be on purpose. When the aspiring performance tester starts out, they have to read a bunch of documentation, thick books with big words, and, frankly, figure out what works by trial and experiment. As they grow, the performance tester speaks in that language, uses it at conferences -- they simply have to in order to get hired. By the time they have achieved some level of professional success, to admit the process just is not that hard, that the emperor has no clothes, is to devalue their own prior work. So the cycle repeats, and, frankly, enables large day rates for project work.

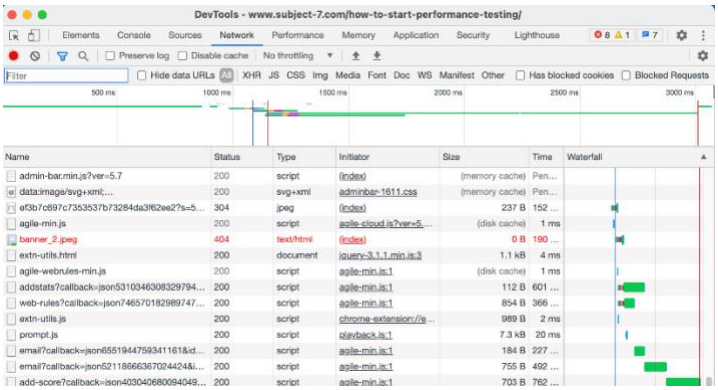
Don't get me wrong, there certainly is a fair bit to performance testing. There is modeling work, load generation, monitoring, and interpreting results, plus some hard-learned lessons along the way. Getting started though? Getting started does not need to be hard, but it does need to address some of the counterintuitive realities of simulating large numbers of users.

Here are my basics of performance and load testing.

Performance vs. Load Testing

First of all, there are no standard definitions for these things. I would even go so far as to argue that there are benefits to competing definitions forcing us to fight for shared understanding. In this article I will use performance testing as an umbrella term, to see if the system is responsive enough under real-use-like conditions. That is different from *simulating* a large number of users, or "load." With these definitions, load testing is a specific kind of performance testing.

To do performance testing with this definition, you can just … test. Explore the software as a single user. Notice the problems. Re-do things that seem slow. Use the browser tools to find a waterfall diagram and figure out what is loading slowly and how quickly.



A Google Chrome waterfall diagram

Often performance problems appear with just a single user, or are so borderline with a single user that those problems will be the place to start to look with many users. Load testing requires simulating multiple users. Doing that well requires a bit of modeling.

Modeling

Like a toy plane or car, models are simplified descriptions of what users do in systems. If you look at what users actually do on websites, they read them. They click around in ways that are difficult to predict. Scott Barber, cofounder of the Workshop on Performance and Reliability, invented an entire modeling language called User Community Modeling Language, designed to graph the user experience and to approximate how often users jump between features. If the system is in production, there is an easier way. Look at log data, collate it by base web page, count the number each appears and calculate the percentage.

With the wrong model, the load will not realistically match the users. The users will go off and do different things, and the load tests will not be as valuable as they could have been. Generally modeling is constrained by the tool you choose to generate load.

Load Generation

This is the tool that takes a script of some kind and runs it over and over again. Generally, these tools are more interested in the time to get service requests, and less interested in the correctness of those results. They might report 404 errors and timeouts, but they might not report a bug on-screen. In the worst case, login might fail. All the results could be "fast" as the software is essentially just re-displaying the failed to login page over and over again.

Most modern tools accept a script with some variation, including variables, to have different users log in, each on their own "thread." The script may have some randomization in it, either for steps or for what to type. The load tools usually "ramp up" the number of users slowly, to see where the system "falls over." Once you find that spot, try hitting it all at once with slightly less load. Another less-common practice is soak testing, where the load running runs for an extended time, say over an entire sprint, to see if there is a memory leak or if a bad file write will overfill a disk.

Monitoring & Reporting

When it comes to data, we care about the time for the entire request on the server (which we can control), the user experience, and the use of the components. That's a fancy way to say we want to know how the CPU, disk, memory, and network are being taxed. Generally, when we hit poor performance at a certain load factor, we look for what component "fell over" due to over-

use. If the system is cloud-based, then use should hit a certain level, the software should "spin up" a new server, and then use should go back down. In that case, it might be helpful to monitor the resources the cluster manager has allocated.

Front-end performance tools look at how long it takes the request to display, usually on a web page. In many cases, it is possible to structure the page so that it loads the incremental elements, instead of loading all the files, and then showing the completed page. These tools can also identify, for example, when an image file is very large. In most cases, a lower resolution image can work just fine.

You can find time-to-live by injecting time into the server logs, from request-in to request-served. The difference between the time on server and the total trip is the *propagation delay*, the delay due to the network, or perhaps, limited by bandwidth. It can be hard to get utilization, time to live, and the component use into a single test run report. Early on, you can just "watch the dials" as the test runs - afterward, the load tool should be able to create a test run report that provides at least the front-end experience data over time as the number of users grows.

Once you have the data of performance under load, you will likely have something like a hockey-stick graph. At a certain number of users, the performance falls apart. That is likely because some component becomes over-used, the system runs out of disk space, or the network becomes saturated and suddenly everything takes much longer.

Which brings up the question, once you have the data … what does it mean?

Evaluation

In an ideal world, you would have a service level agreement (SLA) with the business sponsor on how fast is fast enough, sort of like a specification. Sadly, for most clients, that remains as out of reach as a perpetual motion machine.

My point is that if you are learning about performance testing, you probably don't have SLAs or any sort of mature process. Which, as I alluded earlier, provides an excellent opportunity to experiment and learn. It is possible the engineering team can make up SLA's, collaborating with an empowered product owner. More realistically, the actual performance results will be all over the place, and even if you had SLAs, the decision makers might be willing to compromise on this or that feature to get the software out to users. In that world, you don't work to the SLA as much as express a *service level capability* (SLC) and let the decision makers choose if that is acceptable or not.

There are a couple different schools of thought on how to present the information. You could present the decision makers (read: executives) with the rest results. Generally raw data will make no sense to them and need to be interpreted in context. The way to do that is to create a *story* that uses the measurements as supplemental detail. This is probably best done in person, perhaps backed up by a one-page document with a few tables and graphs. Once the context is

understood, to show change over time, a dashboard may suffice. In that case, the *change in performance over time* becomes more important than today's results out of context.

One way to get there is to use the software under load yourself and see how it feels. That won't work as part of a continuous integration run, but it can create a story and a visceral experience. In the worst of cases, when you present the data, and are told that it's fine, you could point to a laptop and encourage the decision maker to put fifteen minutes into trying to use the software themselves. This creates an opportunity for learning. And, should the executive say "nope, it is fine, then when complaints roll in, the problem-resolution discussion is likely to go very differently.

Right Now

Like most skills, performance testing is best learned by doing. Today, my goal was to take away the mysticism and magic, to make it approachable. Once you understand the concepts, you can dig into a specific application to model, a tool for generating load, an approach to modeling/reporting. By now, you have at least a conversational understanding of performance testing. You'll be able to separate the different pieces of the work, to recognize when a vendor representative is over-simplifying the task, or a technical person might be, perhaps accidentally, over-complicating it.