Apache Tomcat Performance Best Practices

Improving Your Tomcat Performance: Why It Matters

From a web app that is just slow to load, to a microservice that is effectively DDOS'd due to resource exhaustion, poorly performing Tomcat servers can equate to hundreds of hours of lost productivity across an organization and, ultimately, impact a company's bottom line.

Mission critical web applications and microservices are ubiquitous across the enterprise these days, so taking the time to properly tune your application servers can translate into real world dollar and time savings.

5 Best Practices for Apache Tomcat Performance

When a poorly-performing Tomcat deployment can negatively impact a business, it's important to improve that performance quickly. In the following sections, we look at five of our favorite best practices for improving Apache Tomcat performance.

1. Stylin' and Profilin': The Importance of Profiling Your Application

The first place to start with any performance tuning, on any platform is with the application itself. If you don't profile your application, you can't answer questions like:

- Do you know how much time your web application spends in garbage collection?
- Do you know how many threads your app uses?
- Do you know what its longest running query is?
- How many disk operations does your app perform at any given time?
- Do you know the answers to these questions when you have 10 concurrent users? How about 100? 1000?

Profiling your web application is a must! It also happens to be one of the most commonly overlooked aspects of application development we run into.

When we start a Tomcat professional services engagement and ask if they know what their average garbage collection times are under load, the answer is all too often "no". In way too many cases, garbage collection logs aren't even being collected.

Building an application profile doesn't have to only occur when doing load/performance testing (though it should; more on that later). We can build a real-world application profile over time by making sure we are collecting the right data and metrics.

By making sure we are collecting garbage collection logs, capturing regular thread dumps, performing access logging, gathering database call times, and etc., we can

make sure we have the data to review over time to get a better understanding of how applications are running and what they look like in a real-world scenario.

Of course, it should go without saying but there is not much value in collecting this data if no one is looking at it.

A review of an applications performance profile should be something that is baked into every release cycle.

2. Take a Load Off: Automate Your Load and Stress Testing

As mentioned in the previous section, you don't have to do load testing to build an accurate application profile, but you should.

In a perfect world, we would have a 1 to 1 replica of our production environment to perform load and stress testing, but rarely do we find this to be the case. Granted it is a lot easier to spin up a production like environment these days compared to 20 years ago when were doing this all on bare metal, but you don't have to have a perfect production replica to perform worthwhile load and performance stress testing.

Taking a scaled-down model of your production environment and pushing it to its breaking point with a utility like JMeter will still provide your team with invaluable information and creates a baseline that the rest of your performance tuning can be based upon.

Wondering what your JVM memory sizing should be? Wondering if your application would benefit from enabling compressibleMimeType? Load and performance testing will tell you.

We all know how tempting it is (and how often it happens) to skip load and performance testing, but just like the previous tip this should be baked into every release cycle. With modern CI/CD tools like Jenkins and GitLab, automating your load and stress testing to have them run in your CI/CD pipeline is easier than ever.

3. The Matching Game: Configure Based on the Needs of Your Application

Most Tomcat tuning is "needs-based" tuning. In other words, we are tailoring the Tomcat configuration to the given application it is running. Now that we know what our application and performance profiles look like we can start building out environments to match these profiles.

Does the application churn CPU? Does it need a ton of memory for a large JVM? Does it have a lot of disk I/O? Now that we know these things, we know how to build out the compute environment in a way that matches the needs of the application and the projected users that are going to be using the system.

We can also start making fact-based decisions on specific Tomcat settings, like number of threads, max keepalive times and etc..

Does your application have a lot of long running queries? If so, then we may need to set longer keep alive timeouts than we normally would.

Or, on the other end of the spectrum, does the application perform a lot of short duration queries but it fires off a lot of them at one time? In that case we would probably want to consider short keep alive timeouts, but a larger number of connections in the connection pool.

4. Easy Does It: Don't Over-Tune Your Tomcat

We've discussed a lot so far, about people not doing things, but there is the other end of the spectrum as well: people who do way too much.

Be careful to not over-tune your environment. While maybe not as common, we do run into environments that have way too many unnecessary Java arguments, or they are set incorrectly and the default values were providing better performance to begin with.

For the naturally curious, it's tempting to tinker with your environments — and that's good, but just not in production. We want to be sure we are using fact-based evidence when deciding which parameters or Java arguments we are introducing into our productions environments and not just playing the guessing game.

5. To AJP or Not to AJP: Evaluate Your Need for Apache JServ Protocol

Another common issue/question we see regularly here at Open Logic is: "Should we still be using Apache JServ Protocol (AJP) to connect to our Tomcat servers?" The answer to this question is a very unsatisfying "it depends."

Almost invariably, the most common Tomcat configuration that we see is Tomcat running behind some sort of web-based proxy like Apache HTTP Server or NGINX. However, for decades the only option to do this was to use AJP, so a lot of current AJP usage is based on more industry inertia than anything else.

There are definitely some security (no TLS/SSL support) and support (no HTTP 1.2 support) considerations that need to be made here, but from a performance perspective the fact that AJP is a binary protocol means it can provide some performance benefits.

That said, in most environment it's not nearly as important as it used to be. Back when 10/100 ethernet was the norm and bandwidth came at a premium, using a binary protocol over a text-based protocol like HTTP(s) was a no brainer.

Today, where gigabyte ethernet is the standard for any network backplane, the binary aspect of AJP has become a lot less important. Bandwidth constrained environments still exist, so it's not uncommon to see small Tomcat environments running on remote IoT installations. In such cases choosing a binary based protocol like AJP in spite of its security and support issues might make sense.

One also might consider whether or not your environments even need a proxy service at all. The Coyote web server that ships Tomcat is a more than capable HTTP server and can handle serving up web content just fine.

Moving to the dedicated hardware-based load balancers directly connecting to the Coyote HTTP connector resolved this issue altogether. Granted, we still utilized proxy services in our DMZ for security reasons but offloading the load balancing of our Tomcat cluster from AJP and onto real load balancers dramatically increased the number of concurrent connections we could handle.

Regardless of the method, we want to make sure we are tuning Tomcat to handle the number of connections that will be thrown its way — whether that's from AJP or HTTP(s).

If your web proxy is configured to handle 1000 connections but your Tomcat server is only configured to handle 500, you're probably in for a bad time.

Other Apache Tomcat Performance Tuning Tips to Consider

One of the most overlooked and basic Tomcat performance best practices I see from organizations is using an up-to-date Tomcat version.

Making sure you are keeping your Tomcat environment up to date by upgrading Tomcat and Java means gaining access to performance improvements in the JVM and in Tomcat.

As an example, if you are using CMS because the version of Java you are using doesn't support G1GC garbage collection, you are already at a disadvantage right out of the gate.

Keeping your Java and Tomcat environments up to date not only keeps your deployments secure, but there are always continuous performance improvements occurring in these communities that you'll be missing out on by running older versions.

Final Thoughts

Tomcat is fast, and improving Tomcat performance at a basic level isn't difficult. But, for complex or clustered deployments, fine tuning and maintaining your performance can be more complex. Ultimately, how you match your Tomcat deployment strategy to the needs of your application is more important than piecemeal tuning — especially for complex applications.

The five best practices we discussed above are just the start, but there are a few mentioned above that will apply to just about any application:

1. Make sure your application is profiled, and use that data to analyze and tune your application accordingly.
2. Make sure your application is using up-to-date Java and Tomcat versions.

If you follow those two principles, you'll be in a better spot than most enterprises using Tomcat today.