Software Performance and Load Testing Utilizing JMeter

Anna Sharpe

Asharpe213@gmail.com

Abstract

Software of any type requires users to interact with it and depending on the business requirements, the software needs to perform with a certain number of users. A company which expects to sell its software to 100 customers for example needs to understand how often their platform is going to be used. If each customer is expected to use the software once during each business day, the software needs to be able to perform when 100 people are logged in at any given time.

The software company has the obligation to its customers to guarantee their software can perform at a certain level with 100 people logged in each day. Management at any software company should decide what level of performance is adequate and relay those expectations to the customers and development teams.

Utilizing JMeter by Apache is a cost-effective method for evaluating the current performance of the software and exposing errors to the development team which can be fixed. Routine performance testing with JMeter will also show performance over time, exposing how new functionality has affected the performance of the software.

The goal of this presentation is to demonstrate how to set up a JMeter project for the first time and understand the results. There are other ways of utilizing JMeter, but this is my preferred method and I challenge you to expand off it. I will not be going into detail on how to fix any errors found from performance/load testing as I do not expect QA engineers to be fixing code nor server settings.

Bio

Anna Sharpe is a quality assurance engineer located in Spokane, Wa. She currently is employed with RiskLens as their first quality assurance engineer. Sharpe graduated from Whitworth University in 2014 with a BS in computer science. During college she began her career as a quality assurance engineer working as an intern and was quickly hired as a full-time employee. When she is not glued to a computer, she spends her time swimming with the local Masters swim team. She is currently working on renovating her first house.

1 What is the problem?

Software development is very expensive. Finding an engineer who understands how to performance/load test software is difficult. This leads to a lack of performance and load testing before software goes to the hands of customers. An unproven platform in the hands of customers can have devastating side effects. One of which is when engineers must drop what they are doing to babysit production servers and help customers. If the performance of the application is not improved quickly, customers could feel the need to shop the competitors and find a more reliable platform. The reputation damage and fallout of this is huge.

When a new software is being developed, timelines put pressure on management and the engineering team. Both want the software to be complete and in the hands of customers. This can lead to shortcuts being taken and performance/load testing is an easy step to put on the sidelines. However, procrastinating on performance/load testing means any bugs found could be more difficult to fix.

2 Why is this difficult?

Every company wants quality software to sell and stake its reputation on. To achieve this quality software status, managers and engineers must work together to prioritize expectations, including expected performance metrics. It is a substantial undertaking to performance/load test software because a knowledgeable engineer must be allocated who can write the performance/load tests (in any tool) and the systems team responsible for the environment being tested must be available to monitor and fix bugs found. Deadlines always force limitations on what resources are available which could mean even if a bug is found, no one has time to fix it. An added complexity is performance/load tests are most effective when run against the production environment. Any bugs in lower environments may be due to server configurations, which are important to find, but not as important as issues specific to production. The only way to guarantee performance/load metrics in production for customers is to run the tests in production. Even a duplicate of production is not a 100% guarantee because an exact duplicate of production is almost impossible to create.

3 How I solved it

I prefer to use free software which has a lively community of support. I have noticed paid for software is too rigid to be useful and having to contact their support team is often too slow for a QA Engineer under a deadline. I also look for software which has a plethora of detailed documentation to influence my design decisions so I don't have to learn the software by trial and error. JMeter by Apache has been my favorite tool for performance/load testing to date. I can write performance/load tests very easily through a GUI and if I have a particularly tricky feature to test, I can create custom functions to meet my needs. I went to my lead and informed them of the importance of performance/load testing so they could help me relay the importance to all necessary management. I was able to discuss with the appropriate teams responsible for the servers what I needed from them and what impact they should expect.

4 JMeter by Apache

JMeter is an open source application which mimics users to put stress on an application to reveal the performance. To achieve this, it offers a plethora of functions from sending JSON requests to native

commands or shell scripts. JMeter works at a protocol level to look like a browser, but never executes the JavaScript in HTML pages. The only real limitation is the speed of the internet available. Apache maintains a wonderful website documenting all available features and a large community of users who work together.

5 Getting started

The following information is intended to walk a first-time user through the basics of JMeter and give confidence to explore the capabilities further. JMeter has a surfeit of features which are documented on their website to facilitate performance/load testing any application[7]. In addition, I recommend reading the first few pages of the Apache JMeter website to give you more inspiration[1].

5.1 Gather a flow

Once the team, company and management agree to the development of performance/load tests, I begin by gathering a flow through the software which includes all major functionality. A good flow through the platform should include features such as logging in, creating new items, loading a list page, editing an item, etc. The performance/load tests should consider as much database functionality as possible, especially if a list page has a large amount of data it is returning, as these can expose failures in the database connections. Another area to include is any icons being loaded as overly large icons can be the reason a specific page is slow to load. Once the flow has been established, it is time to set up the JMeter project and prepare the application to be tested.

5.2 Create the flow

I recommend first time users of JMeter use the GUI for development as it can be easier to understand since it is a visual representation of what is happening in the tests. The easiest way to get started is to add a recording node to your JMeter project and record the flow through the application. To record interactions, add a HTTP(S) test script recorder node[9] to your project and configure the proxy[3] on a browser. Then simply perform the flow in the browser and the requests will automatically be created as HTTP request nodes in your JMeter project.

If the software being tested requires a login, multiple users need to be available in the environment where the performance/load tests are going to be run. I prefer to use a CSV file to contain the users' data. I organize it such that each row is a unique user and each column represents a specific value for that user (username, password, etc) as shown in the figure below. In the JMeter project, add a CSV data set node[8] and configure the settings appropriately.

	A	B	C
1	asharpe@gmail.com	P@sswOrd1	Red
2	asharpe0@gmail.com	P@sswOrd1	Blue
3	asharpe1@gmail.com	P@sswOrd1	Yellow

A basic test plan should contain a thread group[10], HTTP cache manager[11], request default node[2], cookie manager[6], and one or more HTTP request nodes[12]. At the end of the test plan, I recommend using the summary report[13], results tree[14], and aggregate graph[15] to decipher the results. Other result processors are available, but these three are easy to read and detailed enough to investigate any subsequent errors. These graphs are also excellent for distributing to the team so they can also be informed of the results.

5.3 Variables

Variables are extremely useful throughout the test plan. It is rare to have HTTP requests exactly the same for each thread due to some values in a response are necessary in future requests. One example would be, if I create a new item, that item will probably have an ID value which is unknown until I send the request. In this instance a variable is needed to scrap the ID from the response to insert it into the request to view that new item. Use a JSON extractor[5] inside the new item request node to gather the ID value from the response to an appropriately named variable. Then on the request to view that new item, simply use the variable created: ${itemID_1}. I prefer to rename any variables extracted to ensure I can easily understand my requests. Another type of variable I use frequently is the user defined variable[17]. I use this type of variable primarily to determine which environment I am testing so I can easily change which environment the tests run against. I develop the tests in my local environment, so when I eventually run them in production, I can easily change one value in the user defined variable node to the production URL.

5.4 Using Fiddler

HTTP request nodes can be developed in several ways, my preferred method is using Fiddler as the guide to understand my application under test rather than the recording tool mentioned above. With Fiddler open and recording interactions, I manually perform the flow in the application to understand the format of the requests and any values which may be dynamic for each user. It may be necessary to include variables such as any random numbers, environment values which might change, values read from a CSV, files to be uploaded, etc. Look for values which are dynamic in the requests, for example item IDs or date values which change for each thread. If a value is dynamic or should be different for each iteration, a variable needs to be defined. In other situations, the value can be read from an external file, for example using a different user for each iteration, the user credentials can be read from a CSV file.

For each request observed in Fiddler, I take note of the expected response for validation purposes. By default, JMeter will show a request failed if the response is not 200, however, all other validation must be done by the engineer. Adding validation nodes is very important for each node as false positives will render the test useless; the response assertion[18] child node should be added to each request to avoid this. The response body might contain an error parameter and I always validate that the error parameter is empty. The assertion can be more specific and search for exact text in the response of the request indicating exactly what failed. For example, if the response has the possibility to return validation error text containing “incorrect password”, I will validate on that text string so when it fails I know exactly why it failed. This can be expanded to sending a request to create new object, if the request is expected to return an object ID and data specific to the new object, I want to validate those values are present and populated. If the request to create an object returns a 200 but an empty response with a “name must be unique” validation error and I do not check for the expected values, I would never know my request failed. To avoid this circumstance, I add a response assertion to the request specifically checking for an object ID value and any other text indicating the object was indeed created. It can be frustrating to have the last request fail when an issue originated from an earlier request failing silently. Each request I send has at least one assertion to lower the risk of false positives.

6 Running JMeter

I prefer to run the tests using the GUI and there are a few buttons I use routinely. The green play button at the top when clicked will start the test run and while it is running the stop sign button just to the right will become enabled which is used to terminate the test. When running the tests multiple times, i.e. during development, the results node can become crowded and too difficult to read. There is a clear all button in the top with two brooms as the icon, this will clear all results and any error and warning messages. The error and warning messages can be viewed by clicking the yellow triangle sign with an exclamation point

located in the upper right corner of the GUI. These error and warning messages are useful in debugging any issues which may arise.

It is important to always run the tests outside normal business hours for when customers are not expected to be using the application. If the production application is expected to be used by customers 24 hours a day and 7 days a week, the company needs to inform customers about a maintenance window when the tests will be run. It is important to minimize customer interaction with the application while the tests are being run because they will not appreciate a slow application. In addition, the test results will be skewed if for example 10 customers are in the application using resources not represented in the tests. The conclusion of any test run should include a manual smoke test of the application being tested to ensure no lasting damage. It is embarrassing to leave an application in an un-usable state for someone else to discover.

6.1 Performance tests

With the same JMeter project, both load tests and performance tests can be run. Performance tests are defined by management and their requirements of how many users they expect to be in the system at once interacting with the application. In a performance test, the number of threads should reflect the number of expected users. I run the performance test for 2-3 hours and monitor the servers for errors which might occur and the task manager performance. The task manager can give useful data on how the server is handling the performance test. The goal of this performance test is to confirm the application can handle the business requirements set by management.

6.2 Load tests

Load tests on the other hand will test the resources of the application and how gracefully the application can recover from being overloaded. I always prefer to start this process by running the test with only one thread and loop through that one thread 10 times. This step gives me confidence the application can handle a minimal amount of load and my tests are written correctly. The next part of this process is to ramp up the number of threads. While watching the CPU usage on the servers, run the test for 10 minutes repeatedly with each run increasing the number of threads until the server(s) show a CPU usage above 95%. I use 95% CPU usage as my benchmark for load testing because the system is working hard and from here it is easy to add a few more threads to then overload the system. Once the number of threads is determined, I let the test run for 1-2 hours and monitor the server(s) under load. Once the test has concluded, I watch as any queues empty and take note of any error messages recorded.

7 Understanding the results

As the test runs, I prefer to watch the results tree node and monitor the requests, mindfully skimming them for any issues. If too many requests are failing, I will terminate the test run to triage what the issue might be. I also will review the summary report node to keep track of how many requests have gone through so far. Usually I am running these tests while a systems engineer is assisting me to monitor the server logs and CPU usage of the application under test, so I like to communicate to them every so often how many requests have been sent.

The most common issue I have run into is my IP address being blocked due to so many requests coming from my one computer. When this happens, all requests in the view results tree node return 401 unauthorized and it is easy to verify by manually opening the application. If I cannot access the application manually, I know I need to ask the network engineer to whitelist my IP address.

The results tree node can be saved to a file for review after the test run is complete. This will yield details on exactly what the request body sent and exactly why it errored out. These error messages are useful in determining if a certain request is unstable compared to the rest of the requests. For example, I have run into issues with too many users logging in at once and 10% of the requests fail; it was obvious in the results file the server returned a 500 error for the 10% failed requests.

Upon the completion of a satisfactory test run, I save any results to a specific location and document the specifics of the run, including the number of threads used and the duration of the test. I also save the graphs generated on the aggregate graph node as this represents the performance for that day. As these tests are run routinely, these graphs can be combined to show a trend of how performance is changing over time. With this data, the QA engineer can relay information to the team if a certain release has increased or decreased the performance of the application.

7 Conclusion

This outline of how I use JMeter is meant to be a steppingstone for other QA Engineers to begin the process of performance/load testing for their company. None of the companies I have worked for believed performance/load testing were important initially. I started from scratch in working with management to get approval and then develop the tests. The important note is the JMeter tests I have created thus far in my career have found bugs in production which the team decided needed to be fixed.

References

1. https://jmeter.apache.org/usermanual/get-started.html
2. https://jmeter.apache.org/usermanual/component_reference.html#HTTP_Request_Defaults
3. https://jmeter.apache.org/usermanual/jmeter_proxy_step_by_step.pdf
4. https://jmeter.apache.org/usermanual/component_reference.html#HTTP_Header_Manager
5. https://jmeter.apache.org/usermanual/component_reference.html#JSON_Extractor
6. https://jmeter.apache.org/usermanual/component_reference.html#HTTP_Cookie_Manager
7. https://jmeter.apache.org/usermanual/test_plan.html
8. https://jmeter.apache.org/usermanual/component_reference.html#CSV_Data_Set_Config
9. https://jmeter.apache.org/usermanual/component_reference.html#HTTP(S)_Test_Script_Recorder
10. https://jmeter.apache.org/usermanual/component_reference.html#Thread_Group
11. https://jmeter.apache.org/usermanual/component_reference.html#HTTP_Cache_Manager
12. https://jmeter.apache.org/usermanual/component_reference.html#HTTP_Request
13. https://jmeter.apache.org/usermanual/component_reference.html#Summary_Report
14. https://jmeter.apache.org/usermanual/component_reference.html#View_Results_Tree
15. https://jmeter.apache.org/usermanual/component_reference.html#Aggregate_Graph
16. https://jmeter.apache.org/usermanual/component_reference.html#JSON_Extractor
17. https://jmeter.apache.org/usermanual/component_reference.html#User_Defined_Variables
18. https://jmeter.apache.org/usermanual/component_reference.html#Response_Assertion