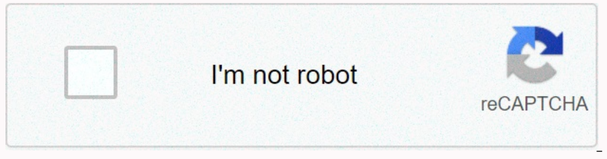

Continue

Locust python report

The Locust performance framework provides hands-on tracking of scripts in GUI and non-GUI mode. But monitoring the out-of-the-box graphical interface has some drawbacks. First of all, its web UI reports are not customizable. Of course, you can try to dive into the locust source code and make your own custom locust version. But in such a case, you break compatibility with the main version and it could bring difficulties as soon as you want to update Locust to a newer version later. Also, it's worth mentioning that this approach is usually quite long unless you're a Python expert. On top of that, the Locust Web interface provides you with a good way to monitor results, but it doesn't store the measurements presented on the web, for comparison with future results. Yes, you can take screenshots. Yes, you can create a pdf from the results page. But all this requires a lot of extra effort and manual interactions, and only gives you static measurements, which you can't play with later. In this article, we'll show you one of the ways to monitor Locust scripts, record important measurements in a time series database (to keep them stored for future comparison) and present all this useful data on the live-time dashboards in circulation. Also, as you might notice in the name of the article, you can do it in just 15 minutes! Sounds intriguing? Let's go then! Before we begin In recent years, we are experiencing a strong shift in the way we deal with the installation of additional tools and systems that help our development processes. Now, almost all of these tools can be installed and configured in minutes using Docker and other virtualization/containerization technologies. If you need to set up a separate database or monitoring dashboard, it doesn't make sense to spend many hours trying to install all the preconditions required for this additional tool, or to study how you can apply additional configurations inside the tool. Today you can easily find the Docker image that already gives you the container with the tool or even with a combination of different tools that you probably need to install for integration. In this case, you don't even need to spend time integrating one tool into another. We're not going to waste any time either and we're just going to use an existing container that allows us to install the entire environment that can collect and display locust performance measurements. Everything you need to do The steps mentioned below is to install the Docker app with the docker dial extension. I'm sure you've already installed it because it saves a lot of time and helps you focus on the main tasks instead of spending time on tool installation and configuration. But if for some reason you still haven't installed Docker on your machine, feel free to use this link and the appropriate section based on your current operating system. As soon as you have Docker installed, you're good to go! Grafana and Graphite Graphite Grafana is one of the best dashboard applications for monitoring and analysis. It provides you with very robust dashboards that have hundreds of components for building a very detailed monitoring. In addition, it gives you the ability to perform different analyses in addition to your data, drawing any combination of measurements and even some it formulas based on your business objectives. It is very often used as a business standard and maybe you don't even need to proceed with installation steps if it is applicable to your business as well. However, don't worry if you don't. We'll show you some tips on how you can run the Grafana dashboard on your laptop in just 5 minutes. However, the main reason we want to use an additional monitoring tool is that we want to collect measurements and keep them as historical data. Measurements collected from performance testing tools are represented by time series data points. Each data point has a time lamp that appears when the measurement is taken out, the metric name (e.g. response time or response length) and the metric value (e.g., 100 ms or 1MB). Obviously, we need a place to store this type of data. Fortunately, there are already many data storage tools to achieve this goal and such a type of storage is called time series database. There are many series databases in the time series available in open source and each has different pros and cons. We are not going to focus on the process of choosing between them because it is worth a separate article. We will stop our selection process on the Graphite tool because it is one of the most famous open-source tools that is usually used in combination with Grafana, and it provides simple database storage to keep data from time series, called carbon. Graphite is used by many companies around the world so you don't need to worry about long-term support and it could become obsolete anytime soon. So we have now developed a combination of two different tools that can be used in integration, and together provide a convenient way to store and draw our performance test measurements. They also have the ability to perform additional analyses in addition to this data. Now we are ready for the easiest part - the installation. As mentioned earlier, there is no need for standard installation steps for these two tools when we can find a preconstructed dockerized solution that should work out of the box. If we spend a few minutes on Google, you must find that there is an excellent Dockerized solution that uses these two tools and has a very simple installation that we can cover in a few steps: This project will allow you to run a combination of these tools in minutes: Grafana (dashboard and measurement analysis monitoring) Graphite (hour series database) StatsD (light statistics collector) Kamon (a set of tools to monitor applications running on the JVM) At this point, we are mainly interested in Grafana and Graphite Graphite this integration has both. If you can't find a need for the rest of the tools, this Docker file can be easily modified to remove them. But we shouldn't care right now because these extra tools don't take up a lot of resources and will operate silently in parallel without having an impact on our main reporting workflow. To implement Grafana monitoring, we need to implement reports that will send measurements from a Locust script to a time series database that will be used as a source for Grafana dashboards. In our case, this time series database will be carbon, which is a backend component of Graphite providing data storage of time series. In this case, our architecture will look like this: As soon as you have all the preconditions mentioned in the previous paragraph and Docker is installed on your local machine, installing these tools will take you only both commands. First, let's check the repository with the dockerfile: git clone After all you need is to go inside the downloaded directory and run: To check that you can open Grafana use this link: Check graphite from this link: Surprisingly, that's it. The installation is complete and now you are running this batch of these tools on your local machine! With the help of Docker and the appropriate Dockerfile, we've just set up all the basic infrastructure needed to start monitoring your Locust performance scripts in Grafana. Locust Metrics Reporting First, to show you the Grafana reports in action, we need to create a basic performance test using the Locust framework. Let's implement a script that triggers a few endpoints of the web application : SimpleLocustScript.py: from the article import httpLocust, TaskSet, task, events, FlightSearchTest (TaskSet) web class: @task def open_login_page(self): self.client.get(/login)) @task def find_flight_between_Paris_and_Buenos_Aires(self): self.client.post(/reserve.php , 'fromPort': 'Paris', 'toPort': 'Buenos Aires') @task def purchase_flight_between_Paris_and_Buenos_Aires (self): self.client.post (.php, 'fromPort': 'Paris', 'toPort': 'Buenos Aires', 'airline': 'Virgin'America, 'flight': 43, 'price': 472.56') class MyLocust (HttpLocust): task_set ' FlightSearchTest host ' ' ' Even if you don't have Python script experience, you can get an idea of the steps. In this test, we simulate the user connection, the search for flights between the cities of Paris and Buenos Aires and the purchase of air tickets. You can run this script using this locust -f SimpleLocustScript.py --no-web -c 100 -r 1 As soon as you see the same output, you are ready to go further and start implementing metric reports. If we go back to our diagram, this part of the workflow is here: the most important data in our performance scripts is response time. Let's see how we can push response time measurements to Graphite. Graphite. we need to find an appropriate graphite API that can be used to accept data for storage. Fortunately, like any other time series database, Graphite provides flexible APIs for incoming data. There are several protocols supported by Grafite. We will use the most convenient and simple complaint protocol. This protocol allows you to send data inside the Graphite (carbon graphite storage) simply by using a simple text in this format: Carbon will automatically manipulate this simple text and convert it into a measure. Now we know the type of data we can carry from locust to carbon writing. But what about the layer of communication? For communication and data transport, you can use a simple socket that allows communication between two different processes on the same or different machines from the same network. By default, Carbon provides the 2003 port that can be used for incoming outlet connections. Now we know how and what we will send. So the easiest part remains. Let's implement reporting measurements from Locust using a socket-to-carbon connection. First, we need to open a plug connection that will be used for metric transport. For better performance, it is highly recommended to open the socket once we start testing and close it at the end, instead of doing the same actions over and over again, every time we send the separate measurement. This is because these operations take on resources that cost money, as well as running time. In order to open the plug at the beginning of the test, you can use the __init__ function this way: Class MyLocust (HttpLocust): task_set - FlightSearchTest host - sock - No def __init__ (auto): super (MyLocust, self).__init__() self.sock - socket.socket() self.sock.connect((localhost, 2003))) Second, we need to make sure that we close the plug after running the test. By default Python takes care of connections that are not closed if the application has been closed, but relying on it is a bad practice. If you don't close the sockets properly, the socket could get stuck thinking that the grip of our end is just slow. That's why it's highly recommended to close the plugs before the end of the app (you can learn more about the Python plugs from this link). To close the socket correctly, even if our application crashed for some reason, we can use the Python output manager that allows us to write a feature that will be performed before the app is terminated: Class MyLocust (HttpLocust): ..... __init__(self): super (MyLocust, self).__init__() self.sock - socket.socket() self.sock.connect((localhost, 2003)) atexit.register(self.exit_handler) def exit_handler (self): self.sock.shutdown (socket. SHUT_RDWR) self.sock.close() After that, we have to decide how much of the performance script we will send collected measurements. Of course, to track our tracking in real time, we need to send measurements as quickly as we can to metrics/ and ideally after each server response. Such behavior can be achieved by locust event hooks. In software development, an event hook means a code that allows you to react to particular events that occur in an application. As we want to send measurements for each request, we will be interested in these two event hooks, provided by Locust out of the box: request_success and request_failure hooks. We can use these event hooks this way: MyLocust class (HttpLocust): ..... def __init__ (self): ..... locust.events.request_success - self.hook_request_success locust.events.request_failure - self.hook_request_fail def hook_request_success (self, request_type, name, response_time, response_length): self.sock.send (%s%d%d % (performance - name.replace ('.',','), response_time, time.time()) def hook_request_fail (auto, request_type, name, response_time, exception): self.request_fail_stats.append ([name, request_type, response_time, exception)Probably, it makes sense to explain the magic that is happening in this chain (the rest seems to be clear): self.sock.send (%s%dd (. - name.replace ('.',','), response_time, time.time()) As we will use a complaint protocol To send measurements, we must send all the measures (each measure must be sent on a separate line) in this format: In this case, if we 'metric path' value , we have: 'metric path' and 'performance'. The last two values seem obvious response_time, but the first one requires an explanation. In order to group our measurements within Graphite, we need to use a single prefix. This prefix can be any channel you want and we have chosen performance. Point symbols are reserved and used in graphite (carbon) to separate metric levels. That's why we added name.replace('.', '-'), because our request name may contain a few points and we don't necessarily want to divide the request names into different levels. In this case, we just use another separator for this. Here's what the result script should look like: from the locust import httpLocust, TaskSet, task, events, web import locust.events import time import import import atexit class FlightSearchTest (TaskSet): @task def open_login_page (self): self.client.get/login) @task def find_flight_between_Paris_and_Buenos_Aires(self): self.client.post (/reserve .php, 'fromPort': 'Paris', 'toPort': 'Buenos Aires') @task def purchase_flight_between_Paris_and_Buenos_Aires (self): self.client.post (/purchase.php, 'fromPort': 'Paris', 'toPort': 'airline': 'Virgin+America', 'flight': 43, 'price': 472.56 }) class MyLocust(HttpLocust): task_set = FlightSearchTest host = « » sock = Aucun def __init__(auto): super(MyLocust, self).__init__() self.sock = socket.socket() self.sock.connect((« localhost », 2003)) locust.events.request_success += self.hook_request_success locust.events.request_failure += auto. </metric></metric></metric></metric></metric></metric> </metric></metric></metric></metric></metric></metric> def hook_request_success (self, request_type, name, response_time, response_length): self.sock.send (%d%d% d (performance. - name.replace('.', '-'), response_time, time.time()) def hook_request_fail (self, request_type, name, response_time, exception): self.request_fail_stats.append ([name, request_type, response_time, exception]) def exit_handler (self): self.sock.shutdown(socket. SHUT_RDWR) self.sock.close() After that, you can run the script and check the data inside Graphite through this link Now we know that our data is up and we can implement some cool dashboards in Grafana! Grafana Dashboard Configuration Grafana provides extremely powerful features for creating dashboards. That's why I recommend you go to examples of this link to get an idea of how to create very detailed dashboards in Grafana using the Graphite query language. But in order to encourage you to do so, it should be mentioned that Grafana has an exceptional ability to export and import existing dashboards, so we have already prepared a small example that you can use immediately. All you need to do is: 1. Connect to Grafana through this link (use admin for username and password) 2. Click the main menu button (top left corner) 3. Enter 'Dashboards' 4. Click the 'Import' 5 button. Download the downloaded json file from our repo examples from this link As a result, you should automatically get the Locust Monitoring Example dashboard that already uses the existing Graphite measurements: This is a basic dashboard that contains information about the response time for our tested settings. Using powerful Grafana capabilities, you can add additional analytical elements to calculate the average, minimum, maximum time and much more. Final Thoughts In this article, we showed how we can implement Grafana monitoring for your Locust performance scripts in just 15 minutes. Docker's powerful container service capabilities can set up the entire Grafana surveillance infrastructure in minutes. The rest is also simple as soon as you have some basic experience with Python. Of course, our solution is not complete and there are many improvements that you can apply to make monitoring even better. For example: Implement measurement reports as a reusable component to use through different tests Adding additional measures in addition to response time Implementing alerts in Grafana to get notifications in case your measurements a threshold Adding additional scans to get other useful measurements like 90% response time, I hope we've given you the main idea that you can use to monitor your Locust scripts in Grafana. Let us know which tools you prefer to monitor your performance scripts because detailed and comprehensive monitoring makes the work of performance test engineers impressive! Click here to subscribe to our newsletter. Run your Locust tests with BlazeMeter! Get scalability, multiple geolocations and Reports. For more information, ask for a demo. Or, put your URL in the box below and your test will start in a few minutes. Minutes.