Daniel Vasconcelos

JUN 8, 2020 • 4 MIN READ • SCALABILITY

Measuring your application's scalability

Technology is rapidly changing the world and we have high expectations from the systems that we use nowadays. But as software engineers, it is really difficult to keep all our systems working as expected when we face a growth scenario like the world is requiring.

We have to be aware of the load that our applications can handle in order to be prepared for the increased demand, and this is related to a concept known as Scalability.

Scalability concept

In Software Engineering, scalability is defined as the ability of a system to remain functional as the load increases. But the concept for itself is not really clear. We need something that we can use as a reference, such as a number that we can follow and see it changing through time. There are a few strategies that we can use to measure scalability.

Measuring scalability

Software applications are all about computational resources. Knowing that, we can ask ourselves some questions that are going to indicate how we are using those resources and maybe help us extract more of them.

For example, let's consider that we are building a web application that responds to HTTP requests, such as a REST API. But let's try to imagine that the application's load is going to increase because we are making really good offers on our website. If this happens:

- Is the performance (response time, for instance) impacted?
- Does the system's error rate, such as timeouts, increase along with the load?
- Are the system's resources such as memory and CPU usage in the expected rate?

These metrics can help us to measure our scalability. We can track these numbers just by monitoring our application, but we don't want to dive into those numbers after we reach a higher number of requests in our API and see that we couldn't handle all the load.

It's just too expensive to expend with marketing and have a crashed application, for example. Therefore, in order to see if we can handle a given number of requests, we can perform load tests.

Load testing

There are different objectives that can be pursued, but with a load (or stress) testing strategy, we are mainly trying to evaluate an application's performance in a given scenario.

For example, we can do load testing in our build pipeline checking if new code impacted the response time, or a load test running in our production environment and making sure that everything is scaling properly.

Let's check a simple example to demonstrate visually how we can analyze that our application is facing issues with increased load.

Example

We are going to look into a REST API in which you can purchase an item and check its status. This application was built using the Micronaut framework and an H2 database (with hikariCP).

Our test scenario was designed to simulate a user that is going to make a purchase and then check its status, meaning that every user is performing two requests.

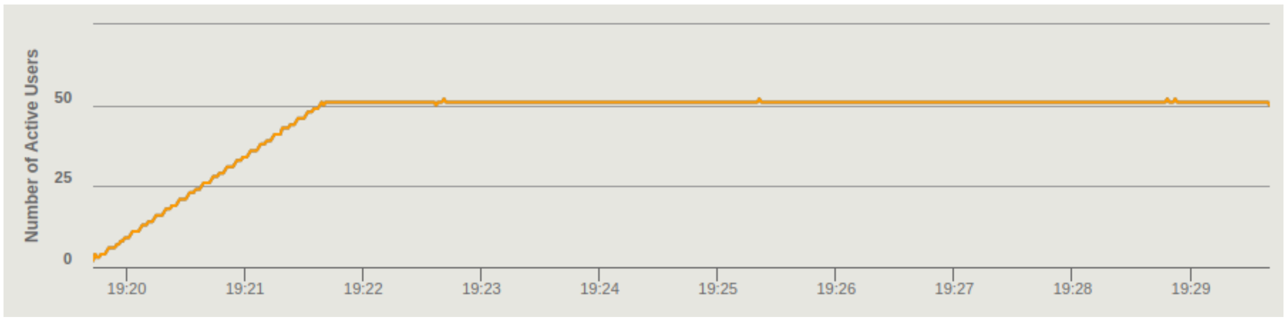
I am not going to detail the tools used to collect the metrics (probably a topic for a later article), but we've used Gatling as the Load Testing tool. With Gatling, we can configure our increasing load in order to test our application properly and simulating a behavior.

For this example, we are expecting our number of users to grow to a given number for a period and then inject this number constantly. The code looks like this:

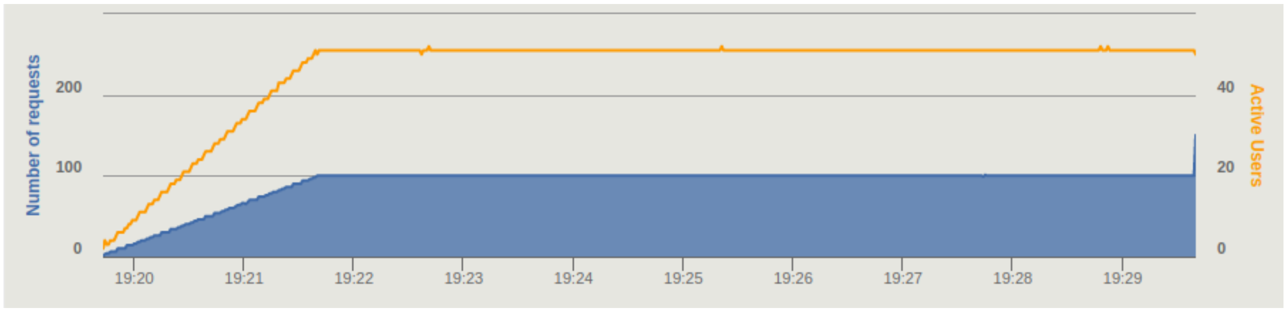
```
inject(
        rampUsersPerSec(0) to 150 during (2 minutes),
   constantUsersPerSec(150) during (8 minutes)
)
```

We have executed the load test multiple times, on the same machine, to stress the application to its limits and then look into the collected metrics. Let's check some graphs generated by Gatling.

In the first execution, the configuration was to ramp from 0 to 50 users and then inject 50 new users constantly. In this first graph, that shows the active users through our test execution, we can see that we indeed got exactly what was configured.

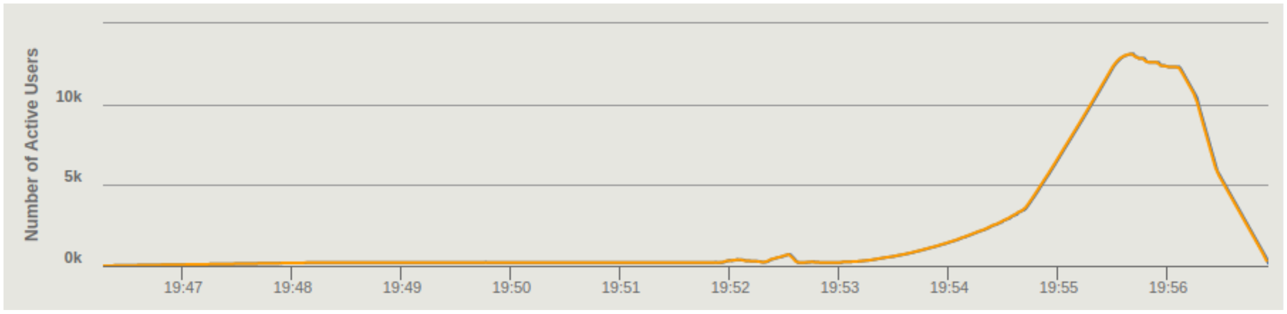


In the second graph, requests per second, we can see that the application responded in a constant rate.

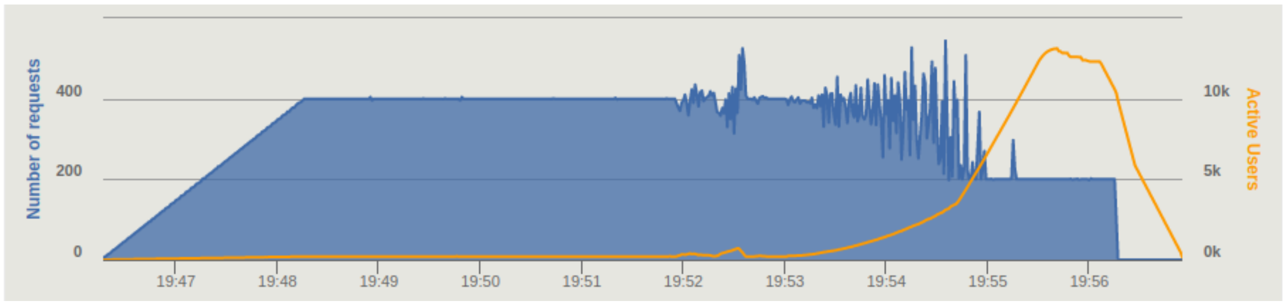


Let's check into the second simulation. Same strategy, ramp and then constant. But now, 200 users.

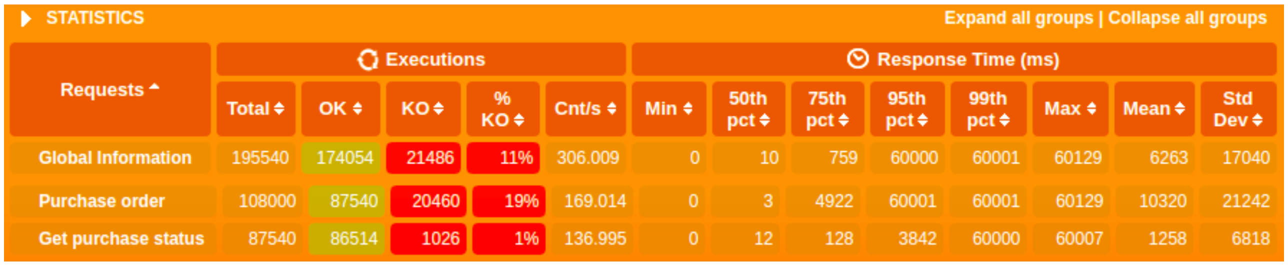
We can see that something different happened. The graph is showing that the number of active users accumulated and ended up exceeding 10k simultaneously.



Analyzing requests per second, we can see that things started good and then something went wrong.



If we keep digging into these simulation we can see that we've reached the limit of database connections and then users would be kept waiting for a response until get a timeout error.


STATISTICS
Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	195540	174054	21486	11%	306.009	0	10	759	60000	60001	60129	6263	17040
Purchase order	108000	87540	20460	19%	169.014	0	3	4922	60001	60001	60129	10320	21242
Get purchase status	87540	86514	1026	1%	136.995	0	12	128	3842	60000	60007	1258	6818

Conclusion

By using the metrics and reading the application's logs, we could see that the database connection pool was the bottleneck for this example. Here, we've used a webserver that answers to HTTP requests as an example, but we can also monitor events processed by a worker or read/write rate into a database.

It really depends on your application purpose. Keep in mind that it is important to choose the right metric so you can measure your scalability properly and keep up with the evolution of your application.

Published by:

Powered by Ghost

→