7 Lab: Non-functional testing (performance)

v2025-03-26

7.0 Introduction

Learning objectives

- Plan performance tests to verify Service Level Objectives (SLO)
- Use k6 tool to develop HTTP-based load tests.
- Assess essential quality attributes in a web application (with *lighthouse* tool).

Support references

- K6
  - k6 built-in metrics
  - k6 learn resources
- Lighthouse, Web Vitals
  - https://web.dev/articles/vitals#core-web-vitals
  - Lighthouse performance scoring
  - Lighthouse scoring calculator
  - Lighthouse node CLI

7.1 Set up k6 tool

Non-functional testing includes several types of tests, but one of the most relevant is performance tests, in which we study the efficiency of the system. A specific case of performance testing is load testing, which generates synthetic system loads to replicate users activity and assess the system behavior.

We'll use a sample QuickPizza website (deployed in a container) providing a recommendation feature along with a basic API. We'll perform some load testing scenarios focused on assessing how a target website behaves (focused mostly on API performance metrics).

a) Clone the repository k6-oss-workshop
b) Run the target app using Docker; besides the QuickPizza website, you'll also have access to a *grafana* and a *prometheus* containers.

```
$ cd k6-oss-workshop && docker compose up -d
```

c) Install k6 command line tool

We'll test the pizza recommendation endpoint (e.g., http://127.0.0.1:3333/api/pizza) by doing a HTTP **POST** request.

d) Run a simple test scenario, without load, adapting the included test script. Assess the metrics on the output, including the “http_req_duration, “http_req_failed”, http_reqs”, “iterations”, “vus” (see k6 docs for more info on available metrics)

1) Edit test.js to complete the http request with authorization properties:

```
"Authorization": "token abcdef0123456789"
```

2) Execute the test:

```
k6 run test.js
```

Take note of the following results:

- How long did the API call take?
- How many requests were made?
- How many requests failed? (i.e., whose HTTP status code was not 200)

7.2 Load test and results monitoring

a) Create a load test scenario (e.g., test_stages.js) assuming the evolution of traffic/load according with the following stages:

```
export const options = {
  stages: [
    // ramp up from 0 to 20 VUs over the next 5 seconds
    { duration: '5s', target: 20 },
    // run 20 VUs over the next 10 seconds
    { duration: '10s', target: 20 },
    // ramp down from 20 to 0 VUs over the next 5 seconds
    { duration: '5s', target: 0 },
  ],
};
```

Execute the test and take note of:

- How long did the API calls take on average, minimum and maximum?
- How many requests were made?
- How many requests failed? (i.e., whose HTTP status code was not 200)

b) Create another load test scenario configuration (i.e., test_stages2.js), considering a more high-demanding usage scenario, with 20 VUs and a ramp-up and ramp-down time of 30s as follows.

```
export const options = {
 stages: [
   // ramp up from 0 to 20 VUs over the next 30s seconds
   { duration: '30s', target: 20 },
   // run 20 VUs over the next 30 seconds
   { duration: '30s', target: 20 },
   // ramp down from 20 to 0 VUs over the next 30 seconds
   { duration: '30s', target: 0 },
 ],

 thresholds: {
    ...
 },
};
```

c) Use thresholds criteria to track whether certain Service Level Objectives (SLOs) are being met.

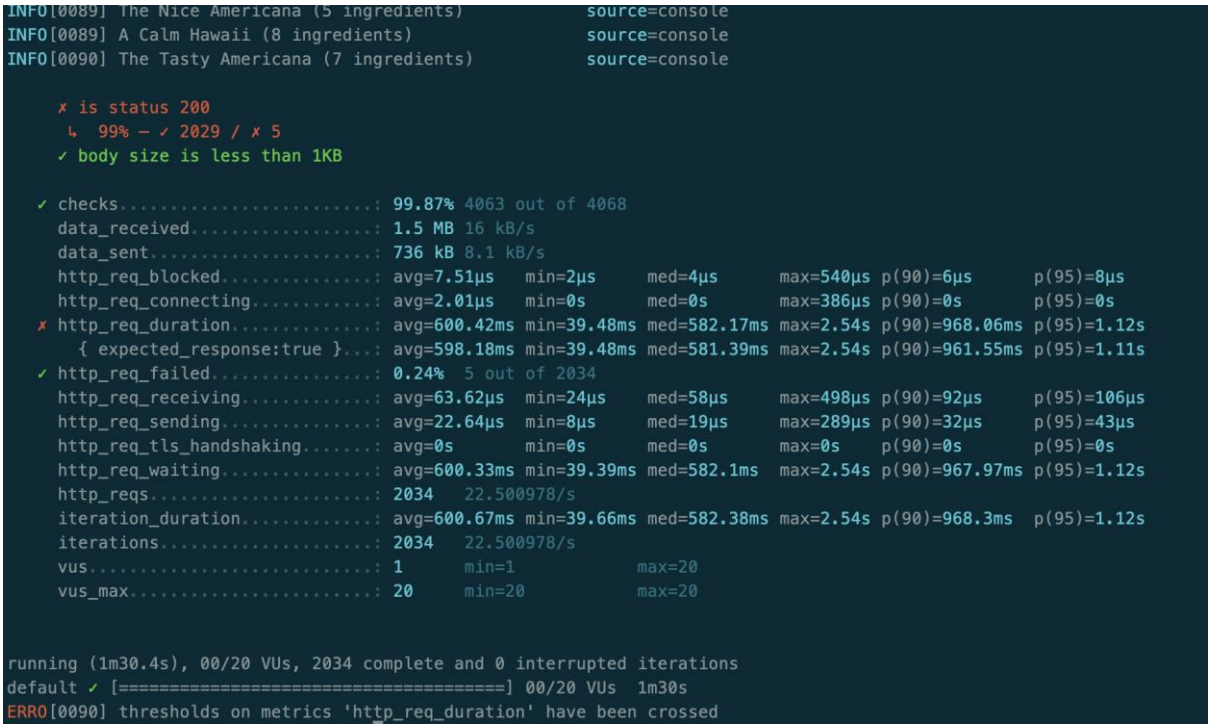
Assume that your team has agreed on a set of SLOs to:

- Ensure that 95% of the requests take less than 1.1 seconds
- Define a threshold so that the 95% percentile (i.e. p(95)) of the http requests duration is lower than 1.1 seconds.
- Ensure that the percentage of HTTP errors is less than 2%.

You may not find a metric already available to codify all kinds of thresholds. Instead, you can define a criterion (with *check*) and apply the intended limits:

- Use checks to track that the system behaves accordingly with the expected content, such as successful HTTP status code (200).
  We can also track whether certain text is present on the response. These checks won't fail our test unless they become part of a threshold.
  In sum, add checks to track that:
  - the API response was successful (and implicitly how many API requests fail)
  - the body size is lower than 1K

Run the test.

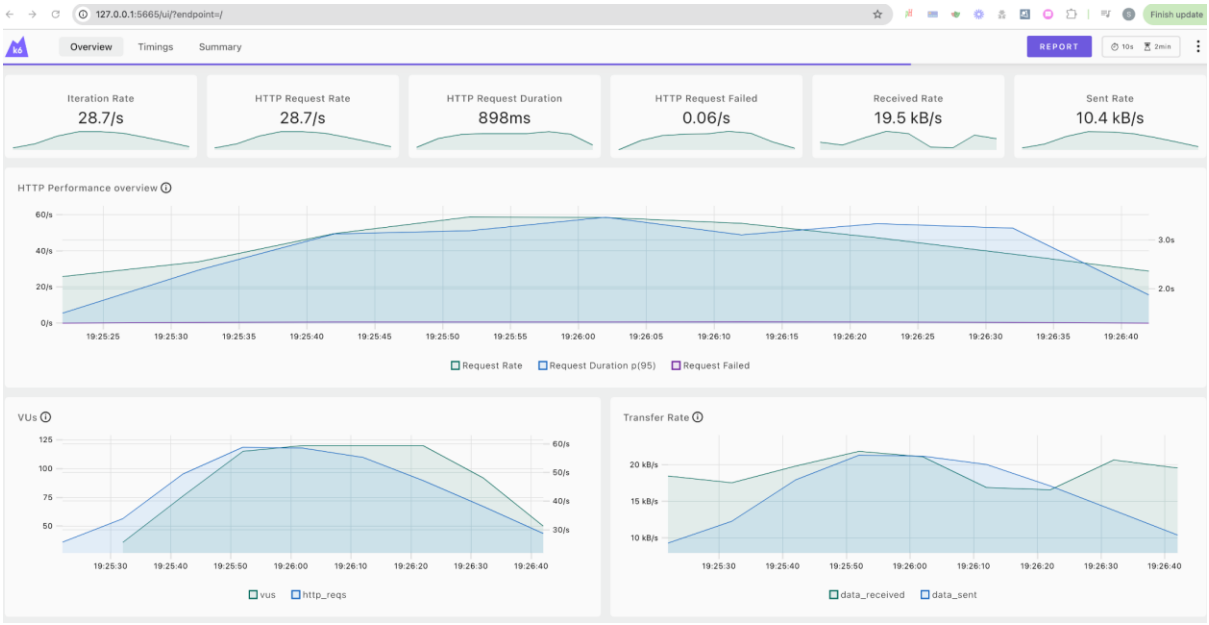


d) The test result summary is useful; however aggregated results hide a lot of information as they don't show how the metrics change during the test execution. It can be more useful to visualize time-series graphs to understand what happened at different stages of the test.
You may monitor the usage of resources using a web dashboard available **while** k6 is running (http://127.0.0.1:5665/). Activate the dashboard using the environment variable K6_WEB_DASHBOARD, e.g.:

```
$ K6_WEB_DASHBOARD=true k6 run test_stages_thresholds_checks.js
```

e) Update the previous configuration to use 120+ VUs instead of 20 VUs and re-run the load test scenario. Check the results. What happened?

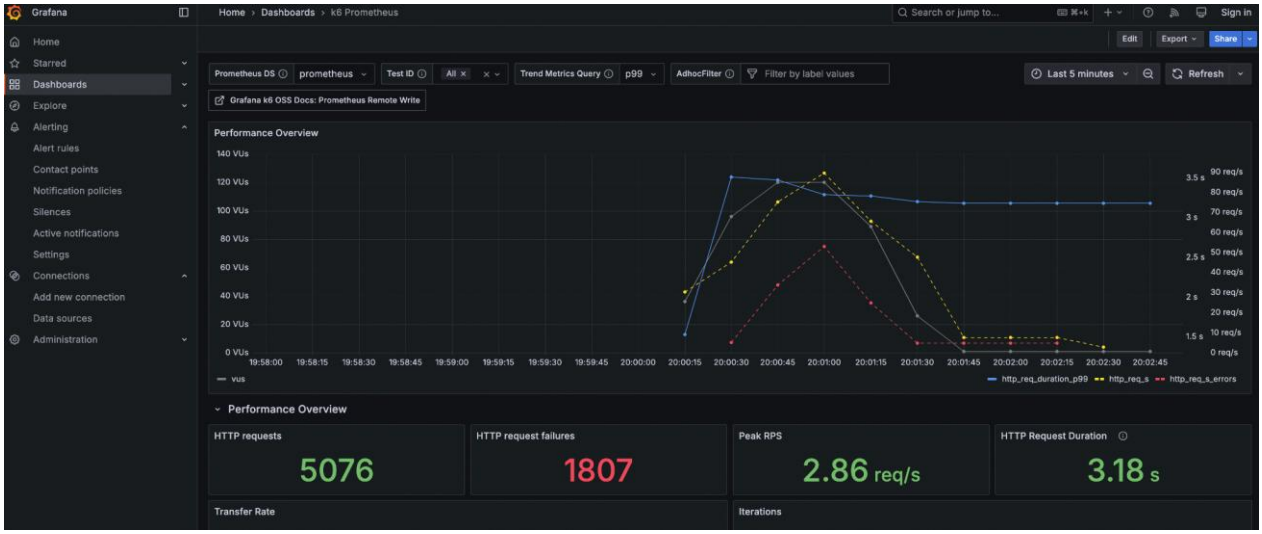
```
INFO[0090] The Tame Napoletana (7 ingredients)           source=console
INFO[0090] The Soft Quattro Stagioni (5 ingredients)     source=console

     ✗ is status 200
      ↳  53% — ✓ 2107 / ✗ 1852
     ✓ body size is less than 1KB

   ✗ checks.........................: 76.61% 6066 out of 7918
     data_received..................: 1.7 MB 19 kB/s
     data_sent......................: 1.4 MB 16 kB/s
     http_req_blocked...............: avg=12.37µs min=1µs     med=4µs   max=1.17ms p(90)=7µs   p(95)=14µs
     http_req_connecting............: avg=5.73µs  min=0s      med=0s    max=709µs  p(90)=0s    p(95)=0s
   ✗ http_req_duration..............: avg=1.85s   min=74.78ms med=1.84s max=5.58s  p(90)=2.85s p(95)=3.17s
       { expected_response:true }...: avg=1.54s   min=74.78ms med=1.5s  max=5.58s  p(90)=2.59s p(95)=2.93s
   ✗ http_req_failed................: 46.77% 1852 out of 3959
     http_req_receiving.............: avg=55.36µs min=15µs    med=50µs  max=727µs  p(90)=83µs  p(95)=98µs
     http_req_sending...............: avg=22.74µs min=8µs     med=20µs  max=558µs  p(90)=33µs  p(95)=42µs
     http_req_tls_handshaking.......: avg=0s      min=0s      med=0s    max=0s     p(90)=0s    p(95)=0s
     http_req_waiting...............: avg=1.85s   min=74.71ms med=1.84s max=5.58s  p(90)=2.85s p(95)=3.17s
     http_reqs......................: 3959   43.861648/s
     iteration_duration.............: avg=1.85s   min=75.41ms med=1.84s max=5.58s  p(90)=2.85s p(95)=3.17s
     iterations.....................: 3959   43.861648/s
     vus............................: 2      min=2           max=120
     vus_max........................: 120    min=120         max=120


running (1m30.3s), 000/120 VUs, 3959 complete and 0 interrupted iterations
default ✓ [======================================] 000/120 VUs  1m30s
ERRO[0090] thresholds on metrics 'checks, http_req_duration, http_req_failed' have been crossed
```



f) The web dashboard integrated in k6 is transient and won't persist the values. An alternative, is to use the dashboard available in the Prometheus instance (http://127.0.0.1:3000) to analyze the evolution of the checks.

Note that you should change the run command to export to Prometheus.

What check failed on the last run you made and up to what value?



7.3 Frontend performance, accessibility & best practices

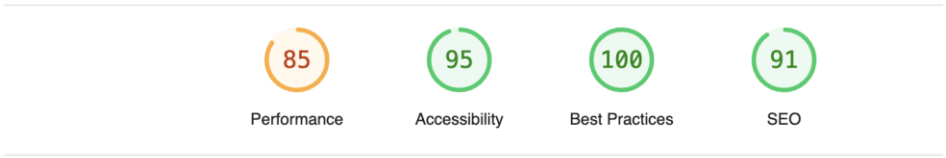
Consider the previous QuickPizza website running locally (e.g., http://127.0.0.1:3333). We wany to audit the frontend, not only for performance, but also considering other quality aspects, such as accessibility.

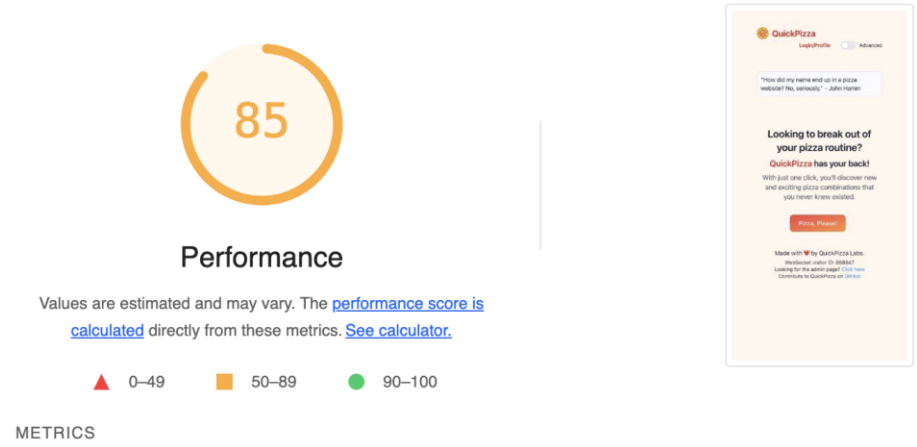
a) Install "lighthouse" CLI tool

```
npm install -g lighthouse
```

b) use the lighthouse to audit our target site:

```
lighthouse --view http://127.0.0.1:3333
```

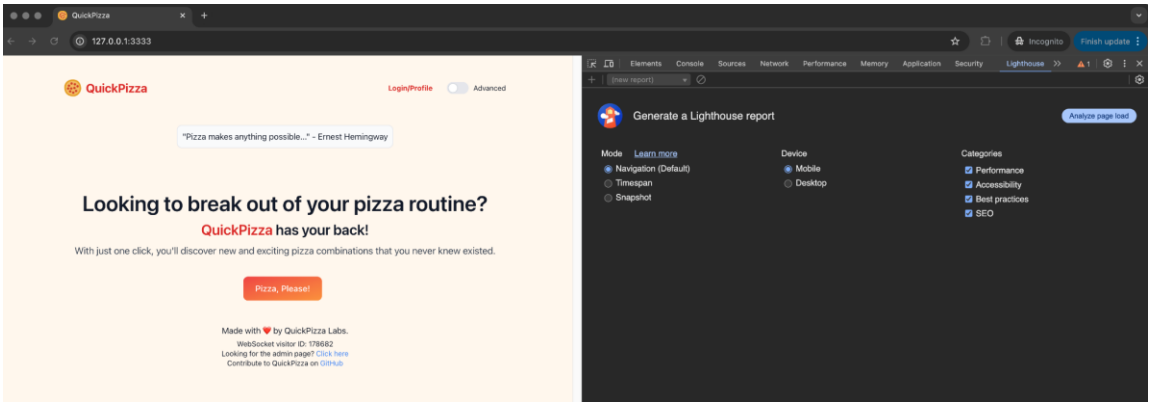




c) Analyze the results report and try to answer the following questions:

- What metrics are contributing the most to the frontend perceived performance? What do they mean?
- How would you make the site more accessible?

d) Use the Chrome browser and open the Developer Tools, namely the "lighthouse" tab. Perform the analysis for the same site and compare with the previous results. Use an incognito tab (why?).



e) Using the embed Lighthouse tool in Chrome, make the analysis considering a desktop device. Are the results the same as the previous ones or are they different? Why?

f) The activities in this lab are also testing, but no functional test was made.
Why are the different aspects (performance, accessibility, best practices, and SEO) worth testing?

g) Consider some well-known website (e.g., https://www.ua.pt or maybe one application developed by yourself) and analyze it using lighthouse.

What are the main problems, and which are their impacts? How easy is it to fix issues in a site that has been *live* for a long time? Could this be avoided and how?