Exercise: k6

This document defines the exercise assignments for the
"Back-End Test Automation" Course @ SoftUni

0. Prerequisites

- **k6 doesn't have a user interface**. In order to write tests, you need a code editor
- It is highly recommended to use **Visual Studio code**

1. Installing k6

- Download and run the [latest official k6 installer]
- After the installation, if you search for k6 in your programs, nothing will appear
- Since **k6 doesn't have a user interface**, you'll have to **check** if it is **properly installed via terminal**
- Open your preferred terminal (we are using PowerShell) and type **k6**

```
PS C:\Users\mddim> k6

          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

Usage:
  k6.exe [command]

Available Commands:
  archive     Create an archive
  cloud       Run a test on the cloud
  completion  Generate the autocompletion script for the specified shell
  help        Help about any command
  inspect     Inspect a script or archive
  login       Authenticate with a service
  new         Create and initialize a new k6 script
  pause       Pause a running test
  resume      Resume a paused test
  run         Start a test
  scale       Scale a running test
  stats       Show test metrics
  status      Show test status
  version     Show application version
```

2. Caution!

First and foremost, **never use k6 against applications you don't own or manage**. It is like going to your neighbor's house to test their door strength without asking. **Absolutely not cool!** For our examples, we will use **https://test.k6.io.** This website is kindly offered by k6 **for small tests**. We won't be sending a large load against this application. Think of it as a friendly knock on the door, rather than trying to knock it down.
k6 is a super powerful tool for performance testing, but with great power comes great responsibility.

3. First Test with k6

- **Create a folder** on your computer and name it appropriately (Ours is called "k6_Testing")
- This folder will **keep your k6 scripts**
- **Open the folder** that you created with **VSCode**
- Inside the folder **create new file** and call it **first-script.js**
- In order to make a HTTP request we have to **import HTTP**
- Next, we're going to **create a function** and we're going to use the **syntax export default**
- And inside this function, we're going to **make the HTTP request against the server**

```
JS first-script.js > ...
1   import http from 'k6/http';
2
3   export default function () {
4       http.get('https://test.k6.io');
5   }
```

- Now, we simply **use our terminal** (We recommend using the VSC integrated terminal, since it's more convenient, but you can use PowerShell, CMD or whichever terminal you like)
- We are using **k6**, the command will be **`run`** and the file would be **`.\first_script.js`**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Program Files\k6\K6_Testing> k6 run .\first-script.js

          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

     execution: local
        script: .\first-script.js
        output: -

     scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
              * default: 1 iterations for each of 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


     data_received..................: 17 kB 39 kB/s
     data_sent......................: 442 B 1.0 kB/s
     http_req_blocked...............: avg=320.26ms min=320.26ms med=320.26ms max=320.26ms p(90)=320.26ms p(95)=320.26ms
     http_req_connecting............: avg=111.57ms min=111.57ms med=111.57ms max=111.57ms p(90)=111.57ms p(95)=111.57ms
     http_req_duration..............: avg=114.41ms min=114.41ms med=114.41ms max=114.41ms p(90)=114.41ms p(95)=114.41ms
       { expected_response:true }...: avg=114.41ms min=114.41ms med=114.41ms max=114.41ms p(90)=114.41ms p(95)=114.41ms
     http_req_failed................: 0.00% ✓ 0        ✗ 1
     http_req_receiving.............: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_sending...............: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_tls_handshaking.......: avg=153.63ms min=153.63ms med=153.63ms max=153.63ms p(90)=153.63ms p(95)=153.63ms
     http_req_waiting...............: avg=114.41ms min=114.41ms med=114.41ms max=114.41ms p(90)=114.41ms p(95)=114.41ms
     http_reqs......................: 1     2.297122/s
     iteration_duration.............: avg=435.32ms min=435.32ms med=435.32ms max=435.32ms p(90)=435.32ms p(95)=435.32ms
     iterations.....................: 1     2.297122/s


running (00m00.4s), 0/1 VUs, 1 complete and 0 interrupted iterations
default ✓ [======================================] 1 VUs  00m00.4s/10m0s  1/1 iters, 1 per VU
```

4. Virtual Users (VUs) and Test Duration

By default, if we don't specify anything, **k6 test has a duration** of about **one second** and will only use **one virtual user**. This is useful when you're checking if the script is working properly, but not so useful for performance tests. So, in order to specify more virtual users and extend the length of our test, we're going to go back to our script:

- **Define** a **constant** called options. We also need to **export it**.
- This will be an **object** that will **store a few things**:
    - **Specify how many virtual users** we want to use.
    - Also **specify a test duration**.

```
export const options = {
    vus: 10,
    duration: '10s'
};
```

Useful shortcut: CTRL+C stops the execution of the test.

```
     data_received..................: 9.5 MB 935 kB/s
     data_sent......................: 84 kB  8.3 kB/s
     http_req_blocked...............: avg=3.31ms   min=0s       med=0s       max=271.7ms  p(90)=0s       p(95)=0s
     http_req_connecting............: avg=1.44ms   min=0s       med=0s       max=121.8ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=119.27ms min=110.76ms med=119.73ms max=130.47ms p(90)=122.08ms p(95)=122.7ms
       { expected_response:true }...: avg=119.27ms min=110.76ms med=119.73ms max=130.47ms p(90)=122.08ms p(95)=122.7ms
     http_req_failed................: 0.00%  ✓ 0        ✗ 817
     http_req_receiving.............: avg=212.1µs  min=0s       med=0s       max=2.1ms    p(90)=990.86µs p(95)=1.02ms
     http_req_sending...............: avg=13.08µs  min=0s       med=0s       max=948.5µs  p(90)=0s       p(95)=0s
     http_req_tls_handshaking.......: avg=1.65ms   min=0s       med=0s       max=142.78ms p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=119.05ms min=110.76ms med=119.53ms max=130.47ms p(90)=121.74ms p(95)=122.43ms
     http_reqs......................: 817    80.78501/s
     iteration_duration.............: avg=122.95ms min=111.29ms med=120.09ms max=397.17ms p(90)=122.48ms p(95)=123.4ms
     iterations.....................: 817    80.78501/s
     vus............................: 10     min=10     max=10
     vus_max........................: 10     min=10     max=10
```

If we look at the results, why there are so many HTTP request and iterations?!
When we **specify 10 VUs** and **10s duration**, k6 takes these VUs and will **get them through this function as many times as possible within the allocated timeframe**. A real user will not call this address so many times in just 10s. So, we will introduce a short pause. **k6 provides a function, called sleep**:

- **Import** the **sleep function** from the k6 library.
- **Call** the **sleep function** with an **argument of 1**, which will pause the script for 1 second after each HTTP request is made.
- **Check again** the number of the **HTTP requests** and **iterations** after the execution.

```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
    vus: 10,
    duration: '10s'
};

export default function () {
    http.get('https://test.k6.io');
    sleep(1);
}
```

5. Different Types of Performance Testing with k6

5.1. Smoke Testing

The goal of a smoke testing is to check if the application we want to test is working. So, with a **small test**, we are just **sending one to three users** to the application to check if it responds. And we are doing it for **no more than thirty seconds**. While doing so, we are also checking our test script to see if it works as expected. Additionally, it sets preliminary benchmarks for performance, given that such a low number of users represents an ideal scenario concerning infrastructure load. Take a good look at the metrics, as most likely they will not get any better.

- **Duplicate** your **first-script.js**
- **Rename** it to **smoke-test.js**
- Add **1 VU** and **30s test duration**
- In the function, except for the Home page, add **/contacts.php** and **/news.php** (since https://test.k6.io is an entire app, it doesn't have only Home page)
- To make it more realistic, add **different sleep times** after each HTTP request (let say 1s and 2s)

```js
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
    vus: 1,
    duration: '30s'
};

export default function () {
    http.get('https://test.k6.io');
    sleep(1);
    http.get('https://test.k6.io/contacts.php');
    sleep(2);
    http.get('https://test.k6.io/news.php');
    sleep(2);
}
```

Some Metrics

```
data_received..................: 88 kB  2.6 kB/s
data_sent......................: 2.2 kB 66 B/s
http_req_blocked...............: avg=27.79ms  min=0s      med=0s       max=500.31ms p(90)=0s       p(95)=75.04ms
http_req_connecting............: avg=10.87ms  min=0s      med=0s       max=195.83ms p(90)=0s       p(95)=29.37ms
http_req_duration..............: avg=181.64ms min=125.7ms med=174.03ms max=243.33ms p(90)=239.43ms p(95)=240.71ms
  { expected_response:true }...: avg=181.64ms min=125.7ms med=174.03ms max=243.33ms p(90)=239.43ms p(95)=240.71ms
http_req_failed................: 0.00%  ✓ 0       X 18
http_req_receiving.............: avg=121.72µs min=0s      med=0s       max=1.02ms   p(90)=567.5µs  p(95)=687.33µs
http_req_sending...............: avg=28.34µs  min=0s      med=0s       max=510.2µs  p(90)=0s       p(95)=76.52µs
http_req_tls_handshaking.......: avg=14.16ms  min=0s      med=0s       max=254.91ms p(90)=0s       p(95)=38.23ms
http_req_waiting...............: avg=181.49ms min=125.7ms med=174.03ms max=243.33ms p(90)=239.43ms p(95)=240.71ms
http_reqs......................: 18     0.529642/s
iteration_duration.............: avg=5.66s    min=5.52s   med=5.57s    max=6.03s    p(90)=5.88s    p(95)=5.95s
iterations.....................: 6      0.176547/s
vus............................: 1      min=1      max=1
vus_max........................: 1      min=1      max=1
```

- When the execution is done, probably the most important thing that we want to see is the "**http_req_failed**" metrics. Here we have 0% of the requests have failed. Or in absolute numbers: 0 have failed, 18 have been successful. (The check mark is for the failed ones and the X is for the successful ones.)
- Another important metric is **"http_req_duration".** This metric shows how long it took from when we have sent the request until we have received a response.
    - **Percentiles**, such as **P90** and **P95**, provide a clearer picture than averages by showing that 90% or 95% of requests complete within a certain time
- We see here also how many requests we have sent **"http_reqs"**
- Also, the number of **"iterations"**. We started the default function six times (meaning it goes to the home page, waits one second, then goes to the second page, waits two seconds, then goes to the third page…)

Notes:

- Anything **larger than 3 users** and **30 seconds** is **smokeless**
- A **bonus** of having smoke tests is that it can be **used to monitor a production system** by running the tests at regular intervals, typically through a CI/CD server. These tests can be scheduled to execute at short intervals, such as every few minutes. And we can run this on a production system because the kind of load that we're putting on, is almost insignificant.

5.2. Load Testing

Testing the application **under a typical load**. Duration of **30 minutes or more**. **3 stages**:

- **Ramp-up:** Gradually increase the number of users to the desired level. This should take around 10% of the total test time

- **Sustained load:** Once the desired number of users is reached, maintain that level of load. This is the main part of the test where you monitor how the system performs under constant pressure
- **Ramp-down:** Gradually reduce the number of users back down to zero. This also should take about 10% of the total test time, mirroring the ramp-up phase

Our smoke test is a great starting point for load testing:

- **Duplicate** your **smoke-test.js**
- **Rename** it to **load-test.js**

If we **add 100 VUs and** change the **duration to 30m**, the script **will immediately start 100 VUs and keep them active for the entire duration** of the test. So, we introduce **stages.** With stages we inform k6 **how to send the traffic**:

- Inside **const options** add **stages**
- Stages will be an array, meaning that we can have as many stages as we like
- The stage doesn't have the concept of a VU. It has **target**

```
export const options = {
  stages: [
    {
        duration: "5m",
        target: 100,
    },
    {
        duration: "30m",
        target: 100
    },
    {
        duration: "5m",
        target: 0
    },
  ]
};
```

- **Ramp-Up Stage:** The test starts with 0 virtual users (VUs) and over the course of 5 minutes, it ramps up to 100 VUs
- **Sustained Load Stage:** Once at 100 VUs, the test maintains that level of load for 30 minutes. During this period, the system's ability to handle a constant, high-volume load is assessed
- **Ramp-Down Stage:** After the sustained load period, the test enters the ramp-down stage, where it reduces the number of VUs back to 0 over 5 minutes

Notes:

- Remember that we're using k6 website and we really don't want to put much load on their application, just showing you the configuration.
- So, to demonstrate how it works, we will run a simple test with **Total duration of 1m 30s and 10 VUs**:
    - Ramp-up stage - **duration: 15s, target: 10**
    - Sustained load - **duration: 1m, target: 10**
    - Ramp-down: - **duration: 15s, target: 0**

Observe the run:

```
running (0m02.3s), 02/10 VUs, 0 complete and 0 interrupted iterations
default   [--------------------------------------] 02/10 VUs  0m02.3s/1m30.0s
```

```
running (0m17.5s), 10/10 VUs, 13 complete and 0 interrupted iterations
default   [======>-------------------------------] 10/10 VUs  0m17.5s/1m30.0s
```

```
running (1m22.4s), 07/10 VUs, 133 complete and 0 interrupted iterations
default   [==================================>----] 07/10 VUs  1m22.4s/1m30.0s
```

More Notes:

- You can customize the number of stages in a load test to simulate different traffic patterns, like overnight lows and daytime highs
- Starting with a simple configuration, like ramping up from 0 to 100 users, is a good way to begin
- Avoid deploying to production if load testing uncovers performance issues that could affect users
- Typically, perform load tests in an environment that closely resembles the production setting, not on the production system itself

5.3. Stress Testing

Stress Testing **puts a higher-than-average load** on the application. How much more load? There's no hard rule. It can be 20% more, but it can also be 100% more. This really depends. Such data should be anchored in **empirical data as much as possible**.

In terms of implementing Stress Testing in k6, essentially, we should **duplicate the load test** and all we have to do is take the same test, but change the value of the target, right?

- **Duplicate** your **load-test.js**
- **Rename** it to **stress-test.js**

OK, where do we want to go? 1000 users?
It's important to keep in mind that while the **load is higher, it should still be a realistic value**. And just as with the load test, if you have no previous data about the application, don't just start with a very, very high number. **Make sure that the load test is performing as expected and then increase** the load by 10%, 20%, 50%, 100%.
But do it gradually!

- The way k6 constructs the stages and sends the virtual users is absolutely the same as per Load Testing.
- So, try to run the same test for the same amount of time, with 20 VUs.

5.4. Spike Testing

Sometimes an app has to deal with **sudden increases in users**, like during a **big event** or **surprise publicity**. We use a spike test to see if the **app can handle these quick jumps in user numbers**. During a spike, the app gets a lot more users all at once, **way more than usual**, and even **more than what we plan for in stress tests**. Unlike other tests, **there's no gradual increase** in a spike test. The **ramp-up time is practically non-existent**, which means that the way we need to define this scenario is also a bit different from what we did before.

- **Duplicate** your **stress-test.js**
- **Rename** it to **spike-test.js**

The idea of this test is **going somewhere very, very high and then boom, we're coming down again**. The users hear something in a commercial, they just grab their phones and then hit the application. And then, all of a sudden, it goes back to practically zero.

```
import http from "k6/http";
import { sleep } from "k6";

export const options = {
  stages: [
    {
      duration: "2m",
      target: 10000,
    },
    {
      duration: "1m",
      target: 0
    }
  ]
};

export default function () {

  http.get("https://test.k6.io/news.php");
  sleep(1);
}
```

- We have only two stages
- You can configure additional stages simulating, the regular traffic and then spiking from that.
- Most likely, these people will go through only the Home page.

Notes:
We're not going to run this test because we have very high numbers here, but you get the idea.

- You can run the test with 20 VUs, just to see it.

5.5. Soak Testing (Endurance Testing)

Takes **Load Test and stretches it for a longer period of time**. Tests how the application will perform with a **constant load for a long period of time**. Ramp-up and Ramp-down stages are not important here.

```
import http from "k6/http";
import { sleep } from "k6";

export const options = {
  stages: [
    {
      duration: "5m",
      target: 1000,
    },
    {
      duration: "24h",
      target: 1000
    },
    {
      duration: "5m",
      target: 0
    },
  ]
};

export default function () {
  http.get("https://test.k6.io");
  sleep(1);
  http.get("https://test.k6.io/contacts.php");
  sleep(2);
  http.get("https://test.k6.io/news.php");
  sleep(2);
}
```

A soak test helps us find **issues that emerge only after the app runs under load for a long time**. It's great for **spotting things like memory leaks**, where the app uses more and more memory until there's none left, leading to a crash. It also **checks if the app is writing too many logs** or **storing too much data**, which could fill up the disk space. Plus, it can reveal if services tend to crash or go offline after being up for a while.

5.6. Breakpoint Testing

What is the load, that our application can handle without crashing? The idea is to find the point where the application starts breaking.
In a breakpoint test, we only have one stage and a very long duration of the test.
For example, let's say **two hours and even longer** and we also have a **very high number of users**, essentially something that is **way beyond anything that we have tested before**.
This test should not be done on a production environment unless the autoscaling capabilities of a production system need to be tested at a maximum. Apart from this, it only makes sense to run a breakpoint test after the successful completion of a load test and of a stress test.

```
import http from "k6/http";
import { sleep } from "k6";

export const options = {
  stages: [
    {
      duration: "2h",
      target: 100000,
    }
  ]
};

export default function () {

  http.get("https://test.k6.io/news.php");
  sleep(1);
}
```

6. Checks

In k6, **checks are essentially assertions** that you use to validate that your system behaves as expected during a test. The syntax might seem unusual at first, but it's quite powerful once you get used to it.

- **Create new file** and call it **checks.js**
- **Write the following script** and let's see what it does

```
JS checks.js > ...
1   import http from 'k6/http';
2   import { sleep } from 'k6';
3   import { check } from 'k6';
4
5
6   export default function() {
7       const response = http.get('https://test.k6.io');
8       check(response, {
9         'HTTP status is 200': (r) => r.status === 200,
10        'Homepage welcome header present': (r) => r.body.includes("Welcome to the k6.io demo site!")
11          });
12          sleep(1);
13    }
```

Here's a breakdown:

- `check(response, {...});` This is the check function. It takes **two arguments**:
  - The **first argument** (response in this case) is the **response object from an HTTP request**
  - The **second argument** is an **object where each property represents a separate check**
  - **Each property inside the check object is a separate assertion**:
    - The **property name** is a **string describing what the check is validating**. It's like a label that will be used in the output of the test to identify the check
    - The **property value is a function that takes the response object as its parameter and returns a boolean value** (true if the check passes, false if it fails)
- Let's run the test to **observe the result**:

```
  execution: local
     script: .\checks.js
     output: -

  scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
           * default: 1 iterations for each of 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


  ✓ HTTP status is 200
  ✓ Homepage welcome header present

  checks.....................: 100.00% ✓ 2       ✗ 0
```

7. Threshold

After the execution of a test, k6 displays the execution report. Currently, if we want to understand how things went, we need to go through the matrix and manually check if each metric is within acceptable limits. By default, **k6 has no knowledge what is acceptable for us or not**. For example, what would be an acceptable request duration? It really depends from application to application and k6 doesn't really care about this.

Thresholds allow you to **define pass/fail criteria** for performance tests. They let you specify performance goals for the system, and tests will fail if these goals are not met

- **Duplicate** your **checks.js**
- **Rename** it to **threshold.js**

Thresholds are defined within the **options object** under the **thresholds property:**

- **Each threshold consists** of a **metric name** and **one or more criteria strings**.
- The criteria strings specify the condition that must be met for the test to pass.
- So, let's add options to our script, and define thresholds for **`http_req_duration`** and **`http_req_failed`**

threshold.js > ...

```
1   import http from 'k6/http';
2   import { sleep } from 'k6';
3   import { check } from 'k6';
4
5
6   export const options = {
7     vus: 10,
8     duration: '30s',
9     thresholds: {
10      'http_req_duration': ['p(95)<3500','p(90)<3000'],
11      'http_req_failed': ['rate<0.01'],
12        }
13  };
14
15  export default function() {
16      const response = http.get('https://test.k6.io');
17      check(response, {
18        'HTTP status is 200': (r) => r.status === 200,
19        'Homepage welcome header present': (r) => r.body.includes("Welcome to the k6.io demo site!")
20          });
21          sleep(2);
22    }
23
```

- Run the test and observe the results

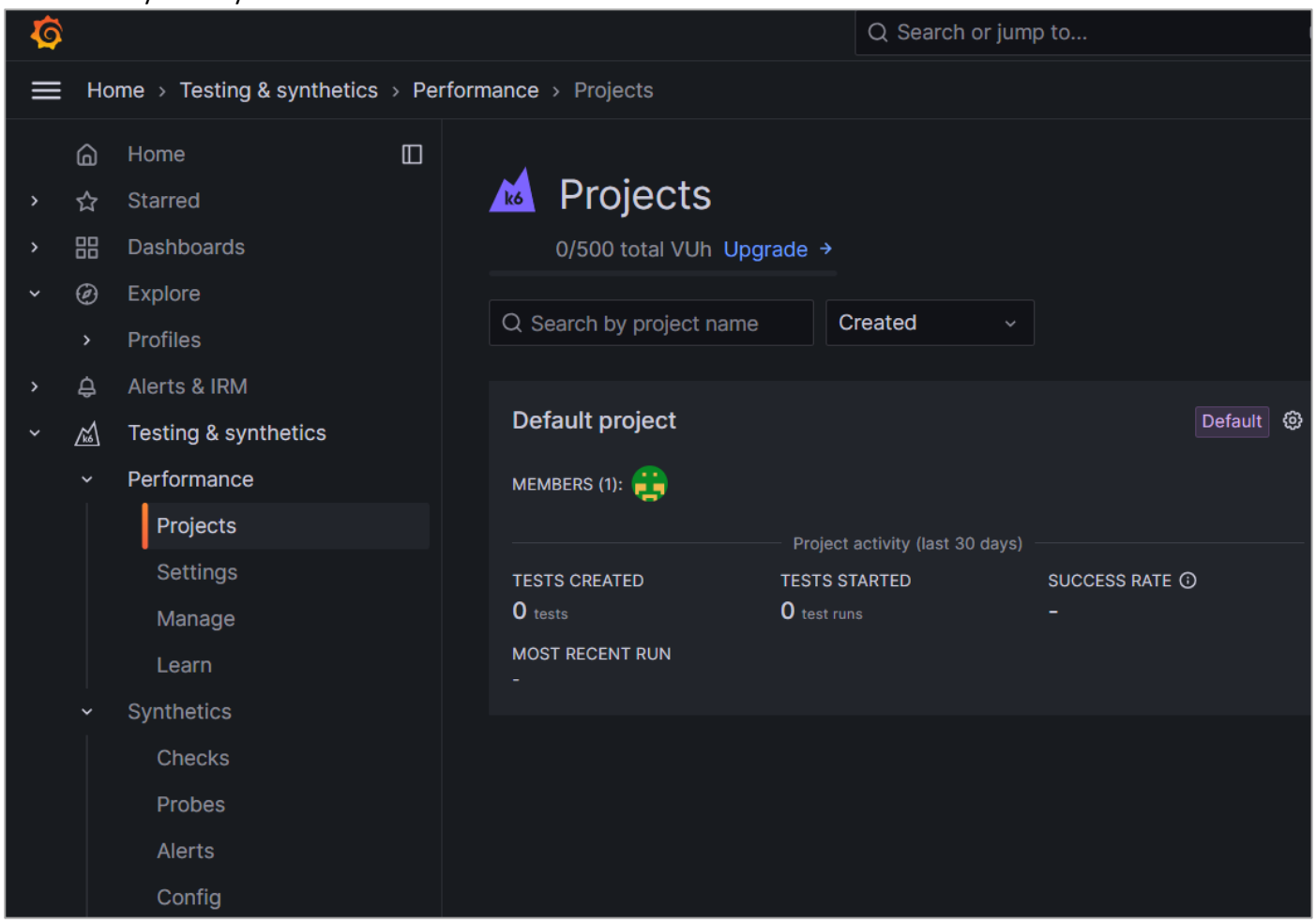
```
   ✓ HTTP status is 200
   ✓ Homepage welcome header present

   checks.....................: 100.00% ✓ 204      ✗ 0
   data_received..............: 1.2 MB  37 kB/s
   data_sent..................: 13 kB   401 B/s
   http_req_blocked...........: avg=29.66ms min=0s       med=0s       max=303.97ms p(90)=58.14µs p(95)=302.89ms
   http_req_connecting........: avg=11.29ms min=0s       med=0s       max=116.91ms p(90)=0s      p(95)=114.32ms
 ✓ http_req_duration..........: avg=1.05s   min=118.43ms med=633.96ms max=3.71s    p(90)=2.85s   p(95)=3.26s
     { expected_response:true }...: avg=1.05s   min=118.43ms med=633.96ms max=3.71s    p(90)=2.85s   p(95)=3.26s
 ✓ http_req_failed............: 0.00%   ✓ 0        ✗ 102
   http_req_receiving.........: avg=6.46ms  min=0s       med=0s       max=140.51ms p(90)=1.1ms   p(95)=73.02ms
   http_req_sending...........: avg=26.48µs min=0s       med=0s       max=829.6µs  p(90)=0s      p(95)=141.39µs
   http_req_tls_handshaking...: avg=14.21ms min=0s       med=0s       max=146.98ms p(90)=0s      p(95)=144.87ms
   http_req_waiting...........: avg=1.05s   min=118.43ms med=611.12ms max=3.71s    p(90)=2.83s   p(95)=3.23s
   http_reqs..................: 102     3.04021/s
   iteration_duration.........: avg=3.09s   min=2.12s    med=2.68s    max=5.86s    p(90)=4.86s   p(95)=5.27s
   iterations.................: 102     3.04021/s
   vus........................: 1       min=1      max=10
   vus_max....................: 10      min=10     max=10
```

- The green checkmark in front indicates that the metrics are within the defined threshold
- Try to run a test with unrealistic threshold, to see what error it will return

8. k6 Cloud

If you're not a big fan of inspecting test execution results in the terminal, this section is for you because we will use k6 Cloud to run the tests and get **some nice reports**. As a side note, the correct name for k6 cloud would be Grafana Cloud k6, but everyone calls it k6 Cloud.

- We need a Grafana cloud account
- When registering, you can use one of the existing login providers or simply use your email and a password
- When you are done with your registration, since k6 is not the only product from Grafana, you will have to find your way to here:



So, how can we run some of the scripts that we wrote in the k6 cloud.
First, we need to log in. From our terminal, we are going to use the k6 CLI to connect to our cloud account. For this we need an API token. To obtain your API token:

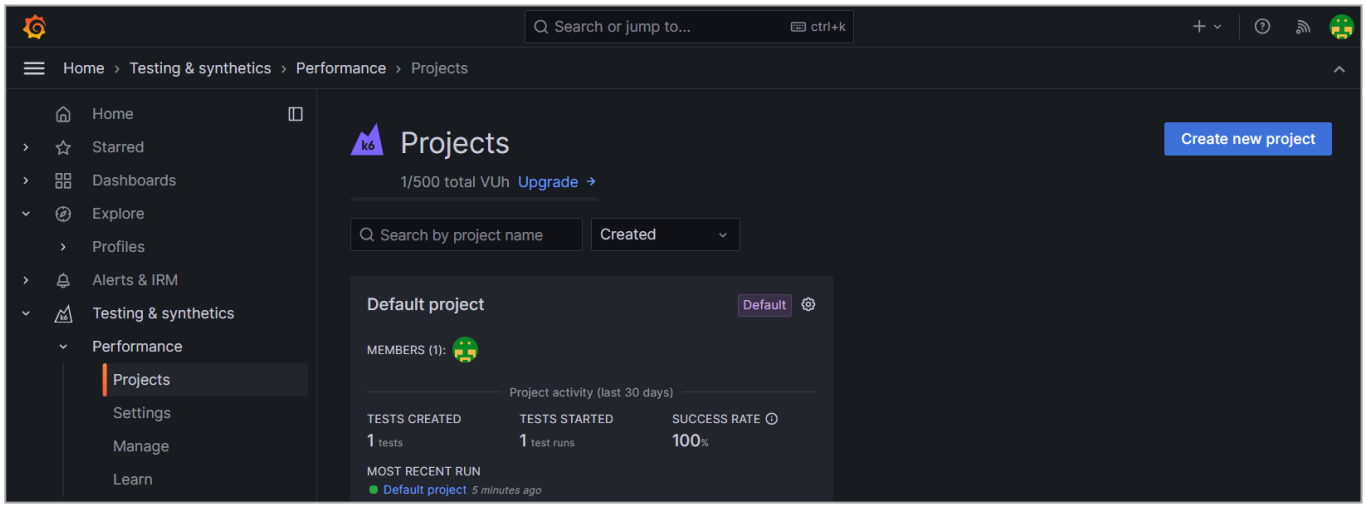
- Go back to k6 cloud
- Navigate to **Testing & synthetics -> Performance -> Settings**
- And there is our personal token
- Go ahead and copy it
- And in the terminal write the following:
  `k6 login cloud --token {your-token}`

```
                              k6 login cloud --token e4c88a35d6c0802da96b8da7cb82a8d95ba8ee
   token: e4c88a35d6c0802da96b8da7cb82a8d9
  Logged in successfully, token saved in C:\Users\mddim\AppData\Roaming\loadimpact\k6\config.json
  PS D:\Program Files\k6\K6_Testing>
```

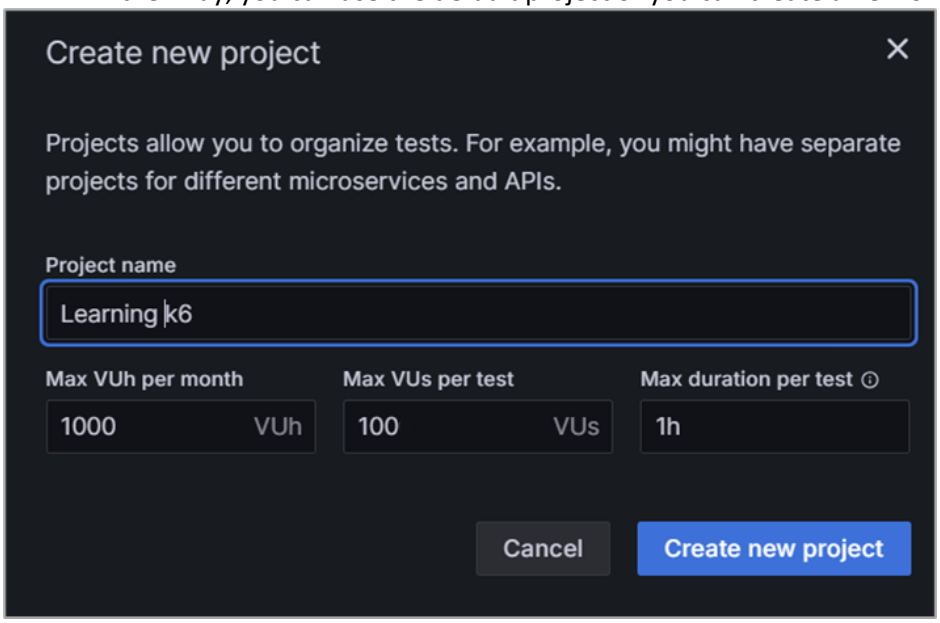
Running the script in the cloud is not much different from running it locally. Essentially, we're replacing the **command run** with the **command cloud**.

Let's say that we want to run our **first-script.js:**

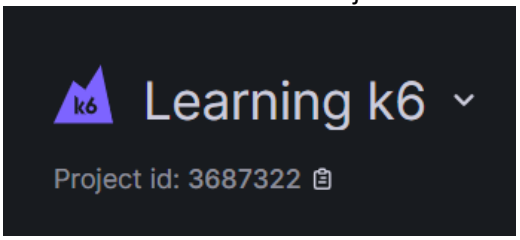
- We need to create a project in order for the results to be displayed
- Head back to your **Grafana account -> Testing & synthetics -> Performance -> Projects**



- Sometimes Grafana creates a Default project for you, when you create your account.
- Either way, you can use the default project or you can create a new one.



- You will need the Project id.



- Add this **id to the options of your first-script.js**, like this:

```
export const options = {
    ext: {
        loadimpact: {
          projectID: '3687208',
          }
    },
    vus: 10,
    duration: '10s'
};
```

Notes: `ext` stands for external, and **`loadimpact`** is the previous name of k6

- Let's run the test
- Type the following in the terminal
  `k6 cloud first-script.js`

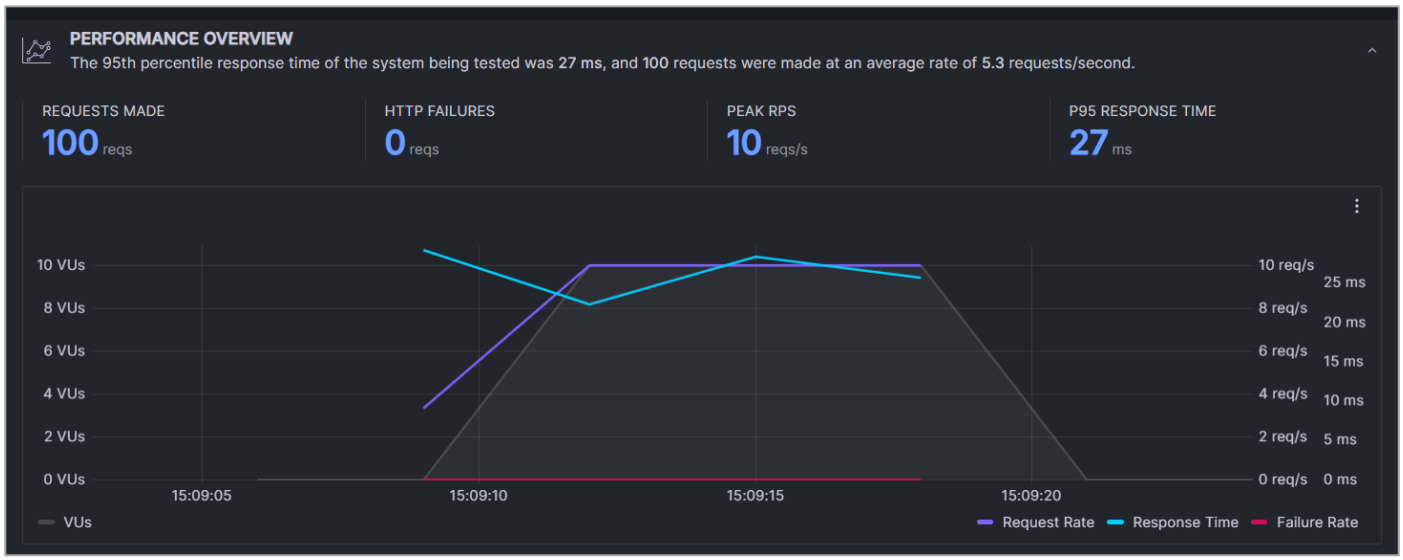
Notes:
You might get the following warning:

```
WARN[0067] The options.ext.loadimpact option is deprecated, please use options.cloud instead
```

Nothing to worry about. **`option.ext.loadimpact`** is not deprecated. It won't be deprecated any time soon. **`options.cloud`** is slowly rolling in, but it is not introduced yet. This is a warning that the user should not see. If you are interested, you can follow the whole saga here **https://github.com/grafana/k6/pull/3348**

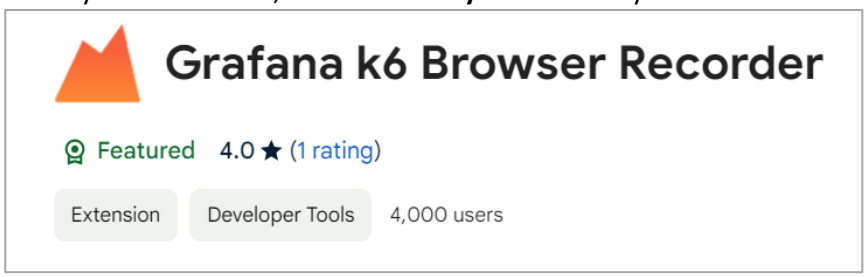
- Assuming that you have managed to run your script
- Let's **see the results in k6 cloud**
- You have the output link in the test results

```
     /\      |‾‾| /‾‾/   /‾‾/
  /\  /  \     |  |/  /   /  /
 /  \/    \    |     (   /   ‾\
/          \   |  |\  \ |  (‾)  |
/ _________ \  |__| \__\ \_____/ .io

  execution: cloud
     script: first-script.js
     output: https://msaum.grafana.net/a/k6-app/runs/2560494
```



9. Grafana k6 Browser Recorder

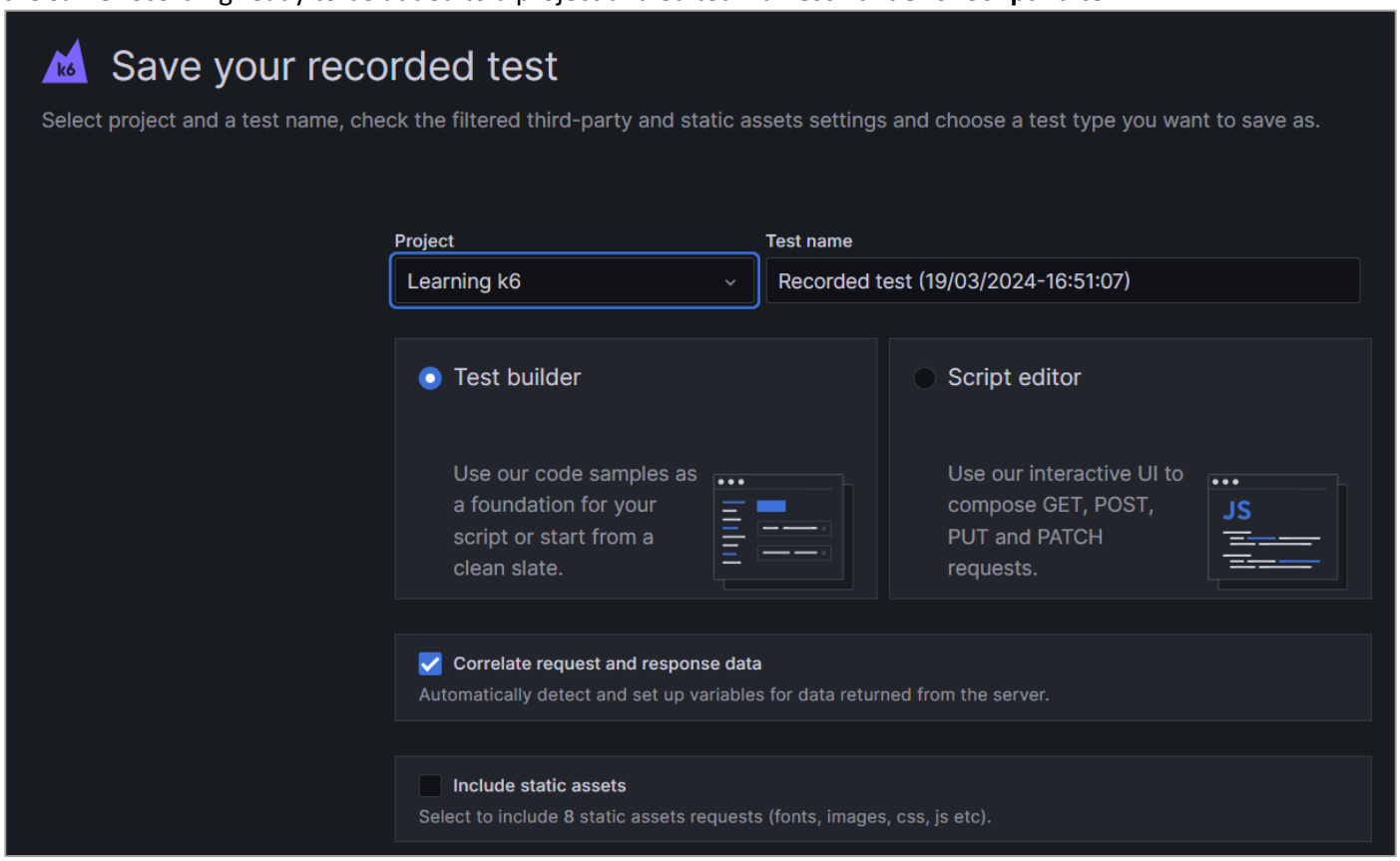
Available at [Chrome Web Store]
When you download it, it **automatically connects** to your Grafana account.



So, let's **try it**:

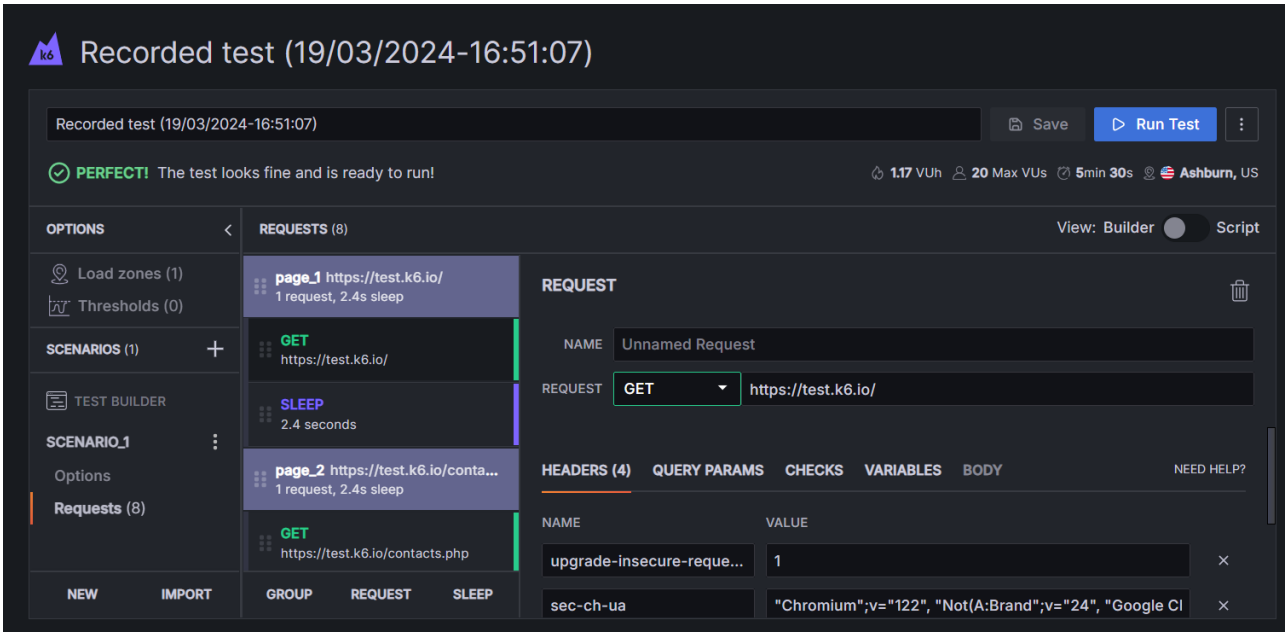
- Click the **record button** and try this **sample scenario**:
  - Open https://test.k6.io/
  - Click on https://test.k6.io/contacts.php
  - Click on Back button
  - Click on https://test.k6.io/my_messages.php
  - Log in with login: admin and pass:123
  - Click Go!
  - Click Logout
  - Click Back
- **Stop the recording**

The recorder will ask you where to save the **.har file** on your computer and also in your **Grafana profile** you will find the same recording ready to be added to a project and edited via **Test Builder** or **Script Editor**.

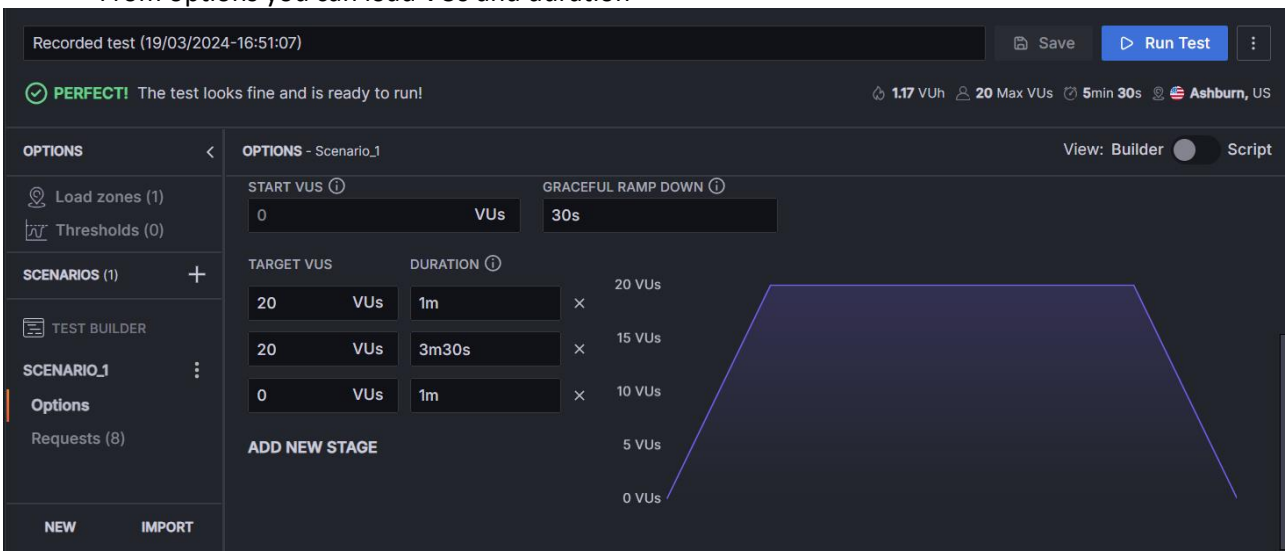


Let's try the **Test Builder**:

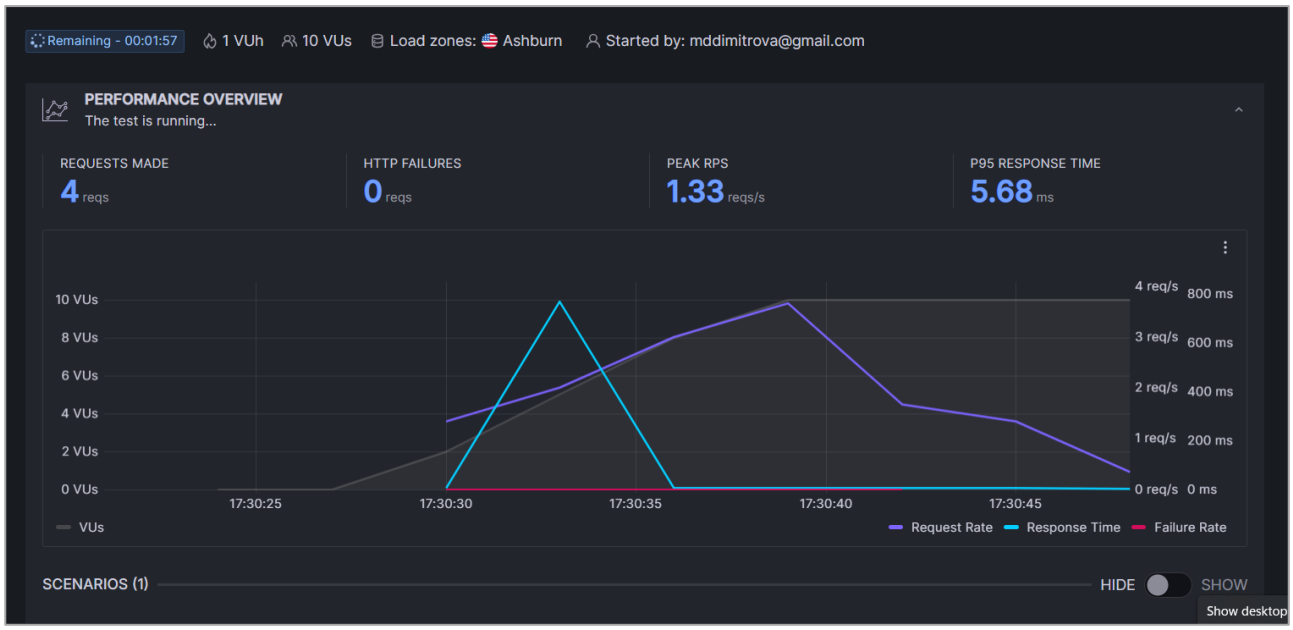
- In the **requests section** you have all the requests that we made, neatly ordered by groups
- We can rename groups/requests. Also, you can check and change almost everything: rearrange requests, add headers, variables, checks, etc.
- **Explore everything** and when you are happy with the result, **save your test**



- From options you can load VUs and duration



- Run the test, and watch it in real time



10. k6 CLI

Let's **imagine** that we have developed a **very complex script**, and we **already have the options**, we have configured **how many VUs we want to use**, and we also have **configured duration**. But we just want to do a **quick smoke test**. We just want **to use one virtual user**.

Of course, the first option that comes to mind is either working with these numbers directly and changing them, or removing this from the code altogether, or commenting them out.

There's a **better way and a fast one**, as well. We will use our **first-test.js script** to demonstrate:

- We have 10 VUs and 10s duration
- In CLI mode we can add flags that change that
- Write the following in the terminal to run the test with only one user, for one iteration:
  `k6 run .\first-script.js --vus 1 --iterations 1`

In **k6 command line flags have the highest order of precedence**, meaning that they can **override essentially anything** that we have **configured**. So, you should remember these:

- The duration flag **`--duration`** or **`-d`** (use -d only with one dash)
  - **`--duration 30s`**
  - You can use **`s`**, **`h`**, and **`m`** to define the duration. The following are valid and equivalent arguments for this flag: **`1h30m10s`**, **`5410s`**, **`90m10s`**
- The iterations flags **`--iterations`** or **`-i`** (use -i only with one dash)
  - **`--iterations 100`**
  - **`-i 100`**
- Virtual user flags **`--vus`** or **`-u`** (use -u only with one dash)
  - **`--vus 10`**
  - **`-u 10`**

11. Further Studies

- Official k6 YouTube channel: **https://www.youtube.com/c/k6test**
- Official k6 Documentation: **https://grafana.com/docs/k6/latest/**