

Performance and load testing with k6



What you'll learn

Pre-requisites

- [KPI](#)
 - Define tests using k6
 - [Define KPIs](#)
- [Integrating Validat](#)
 - [Run the test and push the test report to Xray](#)
 - [Validate in Jira that the test results are available](#)
 - [API](#)
 - [Authentication](#)
 - [JUnit results](#)

Tips

Source code for this tutorial

References

- code is available in [GitHub](#)

Overview

[k6](#) is an open source load testing tool that uses Javascript to write tests.

[Checks](#) and [Thresholds](#) are available out of the box for goal-oriented, automation-friendly load testing.

[k6](#) also has a Cloud version that is a commercial SaaS product, positioning itself as a companion for the [k6](#) open source solution.

Pre-requisites

For this example, we will use [k6](#) to define a series of Performance tests.

We will use the [Thresholds](#) to define KPIs, that will fail or pass the tests.

We will need:

- Access to a [demo site](#) to test
- Understand and define Key Performance Indicators (KPI) for our performance tests
- [k6](#) installed

Start by defining a simple load test in [k6](#) that will target a demo site (travel agency) that you can find [here](#).

The test will exercise 3 different endpoints:

- Perform GET requests to the "/login" endpoint
- Perform POST requests to "/reserve" endpoint (where we will attempt to to reserve a flight from Paris to Buenos+Aires)
- Perform POSt requests to "/purchase" endpoint (where we will try to acquire the above reserved flight adding the airline company and the price)

To start using [k6](#) please follow the [documentation](#).

In the documentation you will find that there are several ways to use the tool and to define performance Tests.

In our case we are targeting to have requests that will exercise some endpoints in our application and that will produce a failed or successful output based on the KPIs that are suited for our application. Keep in mind that we also want to execute these Tests in a CI/CD tool and ingest the results back to Xray.

The tests, as we have defined above, will target 3 different endpoints, for that we wrote a *default function()*.

k6Performance.js

```
...  
export default function () {  
...  
}
```

Next we created three objects that are mirroring the operations we want to exercise:

- loginReq
- reserveReq
- purchaseReq

For each, we have defined the endpoint we want to access, the parameters needed to perform the operation; that are previously defined in constants. Finally, set the Tests to run in parallel, in a batch, as you can see below:

k6Performance.js

```
...  
const BASE_URL = 'http://blazedemo.com';  
  
const reserveParams = new URLSearchParams([  
  ['fromPort', 'Paris'],  
  ['toPort', 'Buenos+Aires'],  
]);  
const purchaseParams = new URLSearchParams([  
  ['fromPort', 'Paris'],  
  ['toPort', 'Buenos+Aires'],  
  ['airline', 'Virgin+America'],  
  ['flight', '43'],  
  ['price', '472.56']  
]);  
  
let loginReq = {  
  method: 'GET',  
  url: BASE_URL+'/login',  
};  
  
let reserveReq = {  
  method: 'POST',  
  url: BASE_URL+'/reserve.php',  
  params: {  
    reserveParams,  
  },  
};  
  
let purchaseReq = {  
  method: 'POST',  
  url: BASE_URL+'/purchase.php',  
  params: {  
    purchaseParams,  
  },  
};  
  
let responses = http.batch([loginReq, reserveReq, purchaseReq]);  
...  
}
```

Keep in mind that this is only one of the possibilities to define a load test. **k6** has very different ways to support your performance testing, for more information please check the [documentation](#).

After having all of that defined we need to instruct **k6** on how to use that information to execute the load test, for that **k6** have the *options* function.

We have defined it like below:

```

export let options = {
  stages: [
    { duration: '1m', target: 50 },
    { duration: '30s', target: 50 },
    { duration: '1m', target: 0 },
  ]
};
.....

```

These options will instruct k6 how we want to execute the performance test, in more detail, this means that we will ramp up VUs (virtual users) to reach 50 VUs in one minute, maintain those 50 VUs for 30 seconds and decrease the users until 0 in on minute. The requests will be randomly chosen from the *http* batch entries we have defined earlier.

In order to execute the tests you can use several ways, for our case we are using the command line.

```
k6 run k6Performance.js
```

The command line output will look like this:

```

cristiancunha@k6:~/Documents/Projects/k6$ k6 run --out jsonmy_test_result.json k6Performance.js
M K6
n .io
execution: local
script: k6Performance.js
output: json (my_test_result.json)
scenarios: (100.00%) 1 scenario, 50 max VUs, 300s max duration (incl. graceful stop):
  + default: up to 50 looping VUs for 2m30s over 3 stages (gracefulRampdown: 30s, gracefulStop: 30s)

Starting (2m30.6s), 00/50 VUs, 2098 complete and 0 interrupted iterations
default: [=====] 100/50 VUs, 2m30s
3M0[051] Preparing the end-of-test summary...
  + 2m30s 2098
  + 0s - / - / / 2098

checks .....: 0.00% / 0 / 2098
data_received.....: 48.9M, 307.887/s
data_sent.....: 2.8 MB, 13.18/s
http_req_blocked.....: avg=172.84µs min=0s med=2µs max=173.46µs p(90)=1µs p(95)=1µs
http_req_connecting.....: avg=342.57µs min=0s med=8µs max=3.11ms p(90)=6µs p(95)=6µs
http_req_duration.....: avg=778.96ms min=136.33ms med=258.87ms max=4.96s p(90)=137.71ms p(95)=24.43ms
  + expected_response_time 3.....: avg=778.96ms min=136.33ms med=258.87ms max=4.96s p(90)=137.71ms p(95)=24.43ms
http_req_failed.....: 0.00% / 0 / 1114
http_req_receiving.....: avg=1.03ms min=1µs med=6µs max=38.46ms p(90)=11.33ms p(95)=13.68ms
http_req_sending.....: avg=18.8µs min=5µs med=15µs max=2.11ms p(90)=1µs p(95)=16µs
http_req_tls_handshaking.....: avg=95.26µs min=0s med=1µs max=1.29ms p(90)=0ms p(95)=0ms
http_req_waiting.....: avg=774.84ms min=136.3ms med=252.57ms max=4.95s p(90)=129.5ms p(95)=115.6ms
http_req.....: 1114 / 1114 / 1114
iteration_duration.....: avg=1.69s min=1.41s med=1.59s max=4.11s p(90)=1.96s p(95)=2.39s
iterations.....: 2098 / 17,000/27/s
vus.....: 2 / 0 / 50
vus_max.....: 50 / 0 / 50

```

This will be enough to execute performance tests, however a manual validation of results must always be done in the end to assess if the performance is enough or not, and looking at Json files is not always easy.

We need the ability to:

- Define KPI that will assert the performance results and fail the build if they are not fulfilled in an automated way (this will be useful to integrate in CI/CD tools)
- Convert the KPI result in a way that can be ingested in Xray

In order to do that we will use the [Thresholds](#) available in k6 and use the [handleSummary](#) callback to generate a JUnit Test Result file ready to be imported to Xray. Notice that with this function you can parse and generate the output that is better suited for your tests.

KPI

In order to use performance tests in a pipeline we need them to be able to fail the build if the result is not the expected. We need to have the ability to automatically assess if the performance tests were successful (within the parameters we have defined) or not.

k6 has an out of the box ability to define [Thresholds](#). In our case we want to define the following ones globally:

- the 90 percentile exceeds 500ms an error will be triggered,
- the requests per second will exceed 500ms an error will be generated
- any error appear during the execution an error will be triggered (because of the error rate KPI).

To achieve this we have added the following thresholds in the *options* :

```

export let options = {
  stages: [
    { duration: '1m', target: 50 },
    { duration: '30s', target: 50 },
    { duration: '1m', target: 0 },
  ],
  thresholds: {
    http_req_failed: [{threshold:'rate<0.01', abortOnFail: true,
delayAbortEval: '10s'},], // http errors should be less than 1%
    http_req_duration: [{threshold:'p(90)<500', abortOnFail: true,
delayAbortEval: '10s'},], // 90% of requests should be below 200ms
    http_reqs: [{threshold:'rate<500', abortOnFail: true, delayAbortEval:
'10s'},] // http_reqs rate should be below 500ms
  },
};

```

Once we execute the test again we will notice that now we have information about the assertions and those results can be acted upon:

```

CristianCunha@0218122240001:~/Documents/1701/ks6/16: run --out jsonmy_test_result.json k6Performance.js
MKG
execution: local
script: k6Performance.js
output: json (my_test_result.json)

scenarios: (100.00%) 1 scenario, 50 max VUs, 30s max duration (incl. graceful stop):
 * default: up to 50 looping VUs for 2m30s over 3 stages (gracefulRampdown: 30s, gracefulStop: 30s)

running (0m22.8s), 00/50 VUs, 113 complete and 18 interrupted iterations
default [-----] 17/50 VUs  0m22.8s/2m30.0s
INFO[0022] Preparing the end-of-test summary...      source=console
c  status was 200
c  0% - / 0 / 120
checks ..... 0.00% / 0 / 120
data_received ..... 2.4 MB 138 kB/s
data_sent ..... 114 KB 5.1 kB/s
http_req_blocked ..... avg=1.80ms min=0ms med=0.2us max=152.70ms p(90)=20.2us p(95)=27.07ms
http_req_connecting ..... avg=2.39ms min=0ms med=0ms max=45.75ms p(90)=0ms p(95)=5.84ms
http_req_duration ..... avg=78.25ms min=10.7ms med=248.74ms max=1.2s p(90)=540.21ms p(95)=56.97ms
{ expected_response:true } ..... avg=79.20ms min=10.7ms med=248.74ms max=1.2s p(90)=540.21ms p(95)=56.97ms
http_req_failed ..... 0.00% / 0 / 778
http_req_receiving ..... avg=1.68ms min=170ns med=0.2us max=29.74ms p(90)=1.17ms p(95)=14.22ms
http_req_sending ..... avg=21.46us min=6us med=16.5us max=345us p(90)=37us p(95)=46.14us
http_req_tls_handshaking ..... avg=78.4us min=0ms med=0ms max=34.43ms p(90)=0ms p(95)=0ms
http_req_waiting ..... avg=79.59ms min=10.6ms med=243.26ms max=1.2s p(90)=546.62ms p(95)=48.02ms
http_reqs ..... 778 30.380895/s
iteration_duration ..... avg=1.63s min=1.4s med=1.57s max=2.29s p(90)=1.89s p(95)=1.95s
iterations ..... 113 3.135042/s
VUs ..... 0 / 0 / 50
vus_max ..... 50 / 0 / 50
INFO[0023] some thresholds have failed

```

Generate JUnit

Now we are executing Tests to validate the performance of our application and we are capable of defining KPIs to validate each performance indicator in a build (enable us to add these Tests to CI/CD tools given that the execution time is not long). What we need is to be able to ship these results to Xray and to bring visibility over these types of Tests.

k6 prints a summary report to stdout that contains a general overview of your test results. It includes aggregated values for all **built-in** and **custom** metrics and sub-metrics, **thresholds**, **groups**, and **checks**.

k6 also provides the possibility to use the **handleSummary** callback. In this callback we can define in which way we want the output to be generated, it provides access to the data available in the test and allow to treat that data in the way you see fit for your purpose.

In our case we used pre-defined functions to produce 3 outputs and added code to produce a JUnit report with more information to be imported to Xray:

- **textSummary** - to write in stdout the summary of the execution.
- **JUnit** - to write to a xml file the JUnit results of the Tests.
- **JSON.stringify** - to produce a json file with the summary of the requests and metrics.
- **generateXrayJUnitXML** - to write a xml JUnit file with extra information, such as, more detail information of the thresholds and the ability to add a file as an evidence.

The code added will look like this:


```

is within the expected values('+ thresholdName +'). Actual values: '+
metricName + ' = ' + getMetricValue(metric, thresholdName, mergedOpts)+ ']]
></system-out>' +
    '<properties>' +
        '<property name="testrun_comment"><![CDATA[Value
registered for ' + metricName + ' is within the expected values- ' +
thresholdName + ']]></property>' +
        '<property name="test_description"><![CDATA[Threshold for
'+ metricName + ']]></property>' +
        '<property name="test_summary" value="' + escapeHTML
(metricName) + ' - ' + escapeHTML(thresholdName) + '"/>' +
        '</properties>' +
    '</testcase>'
)
} else {
    failures++
    cases.push(
        '<testcase name="' + escapeHTML(metricName) + ' - ' +
escapeHTML(thresholdName) + '">' +
            '<failure message="Value registered for ' + metricName + '
is not within the expected values('+ escapeHTML(thresholdName) +'). Actual
values: '+ escapeHTML(metricName) + ' = ' + getMetricValue(metric,
thresholdName, mergedOpts) + '" />' +
            '<properties>' +
                '<property name="testrun_comment"><![CDATA[Value
registered for ' + metricName + ' is not within the expected values - '+
thresholdName + ']]></property>' +
                '<property name="test_description"><![CDATA[Threshold for
'+ metricName + ']]></property>' +
                '<property name="test_summary" value="' + escapeHTML
(metricName) + ' - ' + escapeHTML(thresholdName) + '"/>' +
                '<property name="testrun_evidence">' +
                    '<item name="' + fileName + '">' +
                    fileContent +
                    '</item>' +
                '</property>' +
            '</properties>' +
        '</testcase>'
    )
}
})
})

var name = options && options.name ? escapeHTML(options.name) : 'k6
thresholds'

return (
    '<?xml version="1.0"?>\n<testsuites tests="' +
cases.length +
    '" failures="' +
failures +
    '">\n' +
    '<testsuite name="' +
name +
    '" tests="' +
cases.length +
    '" failures="' +
failures +
    '">' +
cases.join('\n') +
    '\n</testsuite >\n</testsuites >'
)
}

```

As you can see we are treating two cases:

- When the threshold is ok, we add properties that will enrich the report in Xray, namely: comment, description and summary.
- When the threshold is not ok, we add the same above properties plus a failure message and an evidence file that will holds the details of the performance Test.

This is just an example of one possible integration, you can reuse it or come up with one that better suites your needs.

Integrating with Xray

As we saw in the above example where we are producing JUnit report with the result of the tests, it is now a matter of importing those results to your Jira instance. This can be done by simply submitting automation results to Xray through the REST API, by using one of the available CI/CD plugins (e.g. for Jenkins), or using the Jira interface to do so.

In this case we will show you how to import via the API.

API

Once you have the report file available you can upload it to Xray through a request to the [REST API endpoint](#). For that, the first step is to follow the instructions in [v1](#) or [v2](#) (depending on your usage) to obtain the token we will be using in the subsequent requests.

Authentication

The request made will look like:

```
curl -H "Content-Type: application/json" -X POST --data '{ "client_id": "CLIENTID", "client_secret": "CLIENTSECRET" }' https://xray.cloud.getxray.app/api/v2/authenticate
```

The response of this request will return the token to be used in the subsequent requests for authentication purposes.

JUnit results

Once you have the token, we will use it in the API request passing the target project and the Test Plan.

```
curl -H "Content-Type: text/xml" -X POST -H "Authorization: Bearer $token" --data @"xrayJUnit.xml" https://xray.cloud.getxray.app/api/v2/import/execution/junit?projectKey=XT&testPlanKey=XT-251
```

With this command we are creating a new Test Execution that will have the results of the Tests that were executed and it will be associated to the Test Plan XT-251.

Once uploaded the Test Execution will look like the example below

References

- <https://k6.io/open-source>
- <https://k6.io/docs/cloud/>
- <https://k6.io/docs/>
- [demo site](#)