

Performance and load testing with k6



What you'll learn

- [Pre-requisites](#)
- [KPI](#)
 - Define tests using k6
 - [Define KPIs](#)
- [Integrating Validating](#)
 - [Validate in Jira](#) that the test results are available
 - [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:

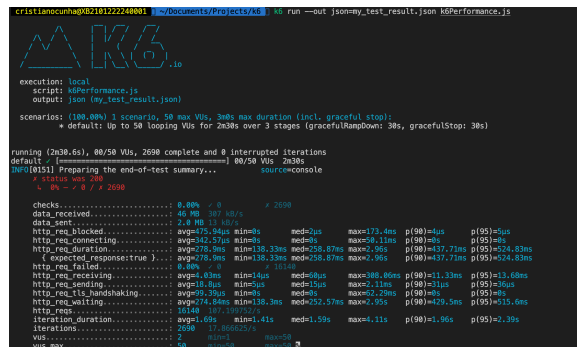
```
export let options = {
  stages: [
    { duration: '1m', target: 50 },
    { duration: '30s', target: 50 },
    { duration: '1m', target: 0 },
  ]
};
.....
```

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@01222240001: /home/cristiancunha/projects/k6 $ k6 run --out jsonmy_test_result.json k6Performance.js

  M K6  .js
  ~~~~

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 2h30s over 3 stages (gracefulRampdown: 30s, gracefulStop: 30s)

running (2h30.6s), 00/50 VUs, 2000 complete and 0 interrupted iterations
default: (=====) 00/50 VUs 2h30s
[0151] Preparing the end-of-test summary... source=console

  2.31min 2000
  1.04s / 0 / 2000

checks.....: 0.00% / 0 / 2000
data_received.....: 0.00 MB / 0.0 MB/s
data_sent.....: 2.8 MB / 11.0 MB/s
http_req_blocked.....: avg=172.24µs min=0s med=2µs max=73.4ms p(90)=1µs p(95)=1µs
http_req_connecting.....: avg=342.57µs min=0s med=8µs max=8.11ms p(90)=µs p(95)=µs
http_req_duration.....: avg=278.5ms min=136.33ms med=258.67ms max=1.96s p(90)=137.71ms p(95)=124.83ms
  ↳ expected response time 3...
http_req_failed.....: 0.00% / 0 / 15154
http_req_receiving.....: avg=1.83ms min=1µs med=6µs max=385.80ms p(90)=11.33ms p(95)=13.68ms
http_req_sending.....: avg=18.8µs min=5µs med=15µs max=2.11ms p(90)=11µs p(95)=16µs
http_req_tls_handshaking.....: avg=95.20µs min=0s med=9µs max=1.27ms p(90)=µs p(95)=µs
http_req_waiting.....: avg=274.84ms min=136.3ms med=252.57ms max=1.95s p(90)=129.5ms p(95)=115.0ms
http_req.....: avg=1.60s min=1.41s med=1.59s max=4.11s p(90)=1.06s p(95)=2.39s
iteration_duration.....: avg=1.60s min=1.41s med=1.59s max=4.11s p(90)=1.06s p(95)=2.39s
iterations.....: 2000 17.00002/s max=0
vus.....: 2 max=0
vus_max.....: 50 max=0

```

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:

```
Cristianocunha@k6:18122240001 [~/Documents/kyto/kyto] $ k6 run --out jsonmy_test_result.json k6Performance.js

M K6

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 2038s over 3 stages (gracefulRampDown: 30s, gracefulStop: 30s)

running (0m22.8s), 00/50 VUs, 113 complete and 18 interrupted iterations
default [0022] Preparing the end-of-test summary... 17/50 VUs 0m22.8s/2038.8s
INFO[0022] state was 200
C 0% - / 0 / 120

checks..... 0.00% / 0 / 120
data_received..... 2.4 MB 128 MB/s
data_sent..... 114 KB 5.1 KB/s
http_req_blocked..... avg=1.80ms min=0ms med=0ms max=152.70ms p(90)=20.2us p(95)=27.87ms
http_req_connecting..... avg=2.39ms min=0ms med=0ms max=55.75ms p(90)=0ms p(95)=5.84ms
http_req_duration..... avg=78.25ms min=10.7ms med=48.74ms max=1.29s p(90)=440.21ms p(95)=56.97ms
  { expected_responsetime }..... avg=79.2ms min=10.7ms med=48.74ms max=1.29s p(90)=440.21ms p(95)=56.97ms
http_req_failed..... 0.00% / 0 / 170
http_req_receiving..... avg=1.68ms min=170s med=0ms max=29.74ms p(90)=11.17ms p(95)=14.22ms
http_req_resolving..... avg=21.40us min=0us med=0us max=345us p(90)=37us p(95)=46.14us
http_req_tls_handshaking..... avg=28.4us min=0ms med=0ms max=24.83ms p(90)=0ms p(95)=0ms
http_req_waiting..... avg=75.59ms min=19.66ms med=43.26ms max=1.29s p(90)=546.62ms p(95)=48.82ms
iteration_duration..... avg=143s min=1.4s med=1.57s max=2.23s p(90)=1.89s p(95)=1.95s
iterations..... 113 5.138445/s
VUs..... 0 0ms 0ms 0ms 0ms
vus_max..... 50 100ms 0ms 0ms 0ms
INFO[0022] 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:

```

k6Performance.js

export function handleSummary(data) {
  console.log('Preparing the end-of-test summary...');

  return {
    'stdout': textSummary(data, { indent: ' ', enableColors: true}), //
    Show the text summary to stdout...
    './junit.xml': junit(data), // but also transform it and save it as
    a JUnit XML...
    './summary.json': JSON.stringify(data), // and a JSON with all the
    details...
    './xrayJUnit.xml': generateXrayJUnitXML(data, 'summary.json',
    encoding.b64encode(JSON.stringify(data))),
    // And any other JS transformation of the data you can think of,
    // you can write your own JS helpers to transform the summary data
    however you like!
  }
}

```

```
<?xml version="1.0"?>
<testsuites tests="3" failures="1">
<testsuite name="k6 thresholds" tests="3" failures="1"><testcase name="
http_req_failed - rate<0.01"><system-out><![CDATA[Value registered for
http_req_failed is within the expected values(rate<0.01). Actual values:
http_req_failed = 0.00%]]></system-out><properties><property name="
testrun_comment"><![CDATA[Value registered for http_req_failed is within
the expected values- rate<0.01]]></property><property name="
test_description"><![CDATA[Threshold for http_req_failed]]><
/property><property name="test_summary" value="http_req_failed - rate<0.01"
/></properties></testcase>
<testcase name="http_reqs - rate<100"><system-out><![CDATA[Value
registered for http_reqs is within the expected values(rate<100). Actual
values: http_reqs = 50.33875387867754/s]]></system-
out><properties><property name="testrun_comment"><![CDATA[Value registered
for http_reqs is within the expected values- rate<100]]><
/property><property name="test_description"><![CDATA[Threshold for
http_reqs]]></property><property name="test_summary" value="http_reqs -
rate<100"/></properties></testcase>
<testcase name="http_req_duration - p(90)<500"><failure message="Value
registered for http_req_duration is not within the expected values(p(90)
<500). Actual values: http_req_duration = 525.57" /><properties><property
name="testrun_comment"><![CDATA[Value registered for http_req_duration is
not within the expected values - p(90)<500]]></property><property name="
test_description"><![CDATA[Threshold for http_req_duration]]><
/property><property name="test_summary" value="http_req_duration - p(90)
<500"/><property name="testrun_evidence"><item name="summary.json"
>eyJyb290Xzdyb3VwIjpb7ImIkIjoiciZDQxZXdhZjZkd4ZjAwYyJlWNGU5ODAwOTk4ZWNmODQyN2UiLC
Jncm9lcHMiOltldCJjaGVva3MiOlt7InBhdGgiOiI6OnN0YXRlc3B3YXMgMjAwIiwiaWF0IjoiIXN
DYxNjYwNzU3YTkwM2Q0ZmI4MmFjNGMlZTEwEWMdlkZSIsInBhc3NlcyI6MCwiZWZpbHMioIjI2Niwi
bmFtZSI6InN0YXRlc3B3YXMgMjAwInldldCJuYWllIjoiciIiwicGF0aCI6IiJ9LCLJvcHRpb25zIjpb
7InFlbnhcnlucmVudGF0aWUuIjozMzkuOTU0LCLJctZWQoIjI0NC45MDNBNDJC70eXB1Ijoici
c5NSkiXSwic3VtbWFWfVEVRpbWVVbml0IjoiciIiwibm9Db2xvciI6ZmFsc2V9LCLJzdGF0ZSI6eyJpc
1N0ZE91dFRUWSI6dHU1ZSwiaXNTdGRFcnuJUUVkioNRydWUSInRlc3RSdw5EdXJhdGlvbklzIjozi
MjAwMy4xNzZ9LCLJtZXRYaWNzIjpb7Imh0dHBfcmlvX3dhhaXRpbmcioiOndmFsdWVzIjpb7ImIheCI
6MTMYocY4lNDISInAoATApIjoimJAmUjcXLCJWkdKlKSINjAjlJCLJ2mkz5OTk5OTk5OTksImF2Zy
IGMjYyZjAlMjYlMjY5MzlmZm91bWUuIjozMzkuOTU0LCLJctZWQoIjI0NC45MDNBNDJC70eXB1Ijoici
HJlbmQiLCJjb250YWLlucyI6InRpbWUifSwiaHR0cF9yZXZfZiJpb7InR5cGUoIjoIjbb3VudGVyIiwi
Y29udGFpbniMiOiJkZWZhdx0IiwiwidmFsdWVzIjpb7InJhdGUoIjoUwLjMzODClMzg3ODY3NZUOLCLJ
jb3VudCI6MTYxMX0sInRocmVzaG9sZHMIOnsicmfOZTwMDAIOnsiB2siOnRydWV9fX0sImh0dH
BfcmlvX3Rsc19oY5kc2hha2luZyI6eyJjb250YWLlucyI6InRpbWUilCLJ2YwLzXMIOnsiYXZNi
```

[illegible]

With the extra code we have added extra information to the JUnit report, it is based in the default JUnit available in k6 with extra fields added.

To achieve it, we have added an extra file: `junitXray.js` that will handle these new informations.

The main method is the one that will generate the JUnit report.

junitXray.js

```

...
export function generateXrayJUnitXML(data, fileName, fileContent, options)
{
    var failures = 0
    var cases = []
    var mergedOpts = Object.assign({}, defaultOptions, data.options,
options);

    forEach(data.metrics, function (metricName, metric) {
        if (!metric.thresholds) {
            return
        }
        forEach(metric.thresholds, function (thresholdName, threshold) {
            if (threshold.ok) {
                cases.push(
                    '<testcase name="' + escapeHTML(metricName) + ' - ' +
escapeHTML(thresholdName) + '">' +
                    '<system-out><![CDATA[Value registered for ' + metricName + '

```

```

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 hold 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 suits 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.

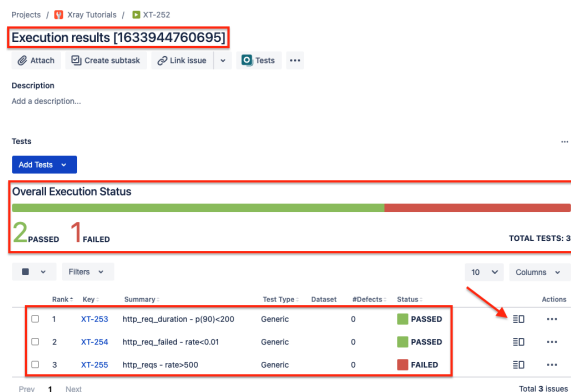
JUnit results

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
```

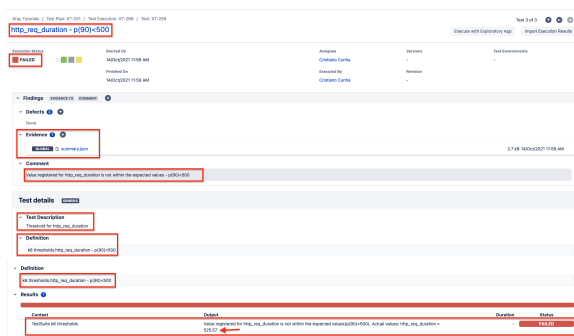
With this command we are creating a new Test Execution that will have the results of the Tests that were executed and it will be associated to the Test Plan XT-251.

Once uploaded the Test Execution will look like the example below



We can see that a new Test Execution was created with a summary automatically generated and 3 Tests were added with the corresponding status and summary (matching the information from the json report).

In order to check the details we click on the details icon next to each Test (below the red arrow in the above screenshot), this will take us to the Test Execution Details Screen



In the Test Execution details we have the following relevant information:

- Summary - Combination of the metric and the threshold defined in the KPIs.
- Execution Status - Passed, this indicates the overall status of the execution of the performance tests.
- Evidence - Holding the file with more detailed information about the performance Test.
- Comment - Showing the comment we have defined in the *XrayJUnit.xml* file.
- Test Description - Allowing adding a specific description for the Test Execution.
- Definition - A unique identifier generated by Xray to uniquely identify this automated Test
- Results - Detailed results with information of the KPI's defined and the value that has breached the KPIs (in case of failure).

Bringing the information of performance tests to your project will give you an overview of the entire testing process and bring that visibility up front for the team to have all the elements necessary to deliver quality products.

Tips

- after results are imported in Jira, Tests can be linked to existing requirements/user stories, so you can track the impacts on their coverage.
- results from multiple builds can be linked to an existing Test Plan, to facilitate the analysis of test result trends across builds.
- results can be associated with a Test Environment, in case you want to analyze coverage and test results using that environment later on. A Test Environment can be a testing stage (e.g. dev, staging, preprod, prod) or an identifier of the device/application used to interact with the system (e.g. browser, mobile OS).

References

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