Performance and load testing with k6

What you'll learn

- Define tests using k6
- Define KPIs, run the test and push the test report to Xray
- Validate in Jira that the test results are available

Source-code for this tutorial

- code is available in GiHub

Overview

k6 is an open source load testing tool that uses Javascript to write the 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 succeed the tests.

We will need:

- Access to a demo site that we aim to test
- Understand and define Keep 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) supplied by BlazeMeter 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, on 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 three different endpoints, for that we have started by writing a *default function()*.

k6Performance.js

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

Next we have 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. And 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]);
...
```

Notice that this is only one of the possibilities to define a load test, k6 have 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 o*ptions* function.

We have defined it like below:

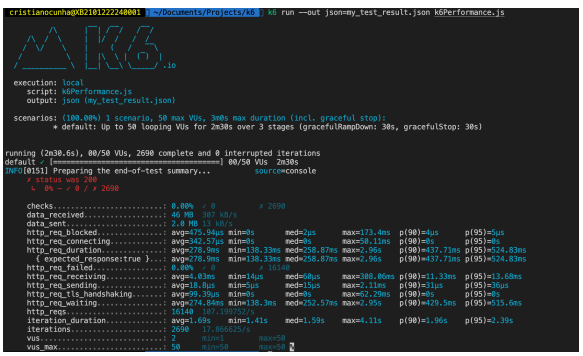
```
export let options = {
  stages: [
    { duration: '1m', target: 50 },
    { duration: '30s', target: 50 },
    { duration: '1m', target: 0 },
  ]
};
....
```

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 below:



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 of 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 in 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 those to be able to fail the build if the result is not the expected, for that we need to have the ability to automatically assess if the performance tests were successful (within the parameters we have defined) or not.

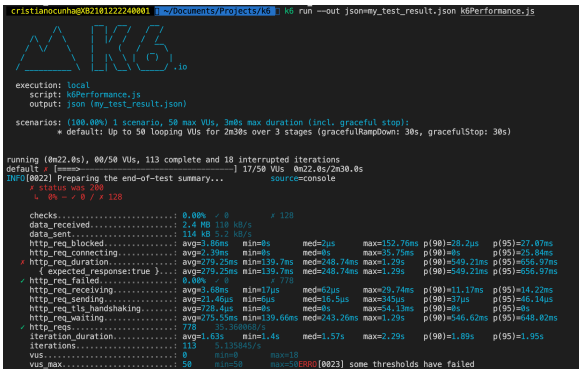
k6 have out of the box the ability to define Thresholds, in our case we want to define the following ones globally:

- the 90 percentile exceed 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:



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), so what we need is to be able to ship these results to Xray to bring visibility over these types of Tests also.

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 have available 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!
  }
}
```

The xrayJunit.xml file generated is:

xrayJunit.xml


```
<?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"
>eyJyb290X2dyb3VwIjp7ImlkIjoiZDQxZDhjZDk4ZjAwYjIwNGU5ODAwOTk4ZWNmODQyN2UiLC
Jncm91cHMiOltdLCJjaGVja3MiOlt7InBhdGgiOiI6OnN0YXR1cyB3YXMgMjAwIiwiaWQiOiIxN
DYxNjYwNzU3YTkxM2Q0ZmI4MmFjNGM1ZTEwMDlkZSIsInBhc3NlcyI6MCwiZmFpbHMiOjI2Niwi
bmFtZSI6InN0YXR1cyB3YXMgMjAwIn1dLCJuYW1lIjoiIiwicGF0aCI6IiJ9LCJvcHRpb25zIjp
7InN1bW1hcnlUcmVuZFN0YXRzIjpbImF2ZyIsIm1pbiIsIm1lZCIsIm1heCIsInAoOTApIiwicC
g5NSkiXSwic3VtbWFyeVRpbWVVbml0IjoiIiwibm9Db2xvciI6ZmFsc2V9LCJzdGF0ZSI6eyJpc
1N0ZE91dFRUWSI6dHJ1ZSwiaXNTdGRFcnJUVFkiOnRydWUsInRlc3RSdW5EdXJhdGlvbk1zIjoz
MjAwMy4xNzZ9LCJtZXRyaWNzIjp7Imh0dHBfcmVxX3dhaXRpbmciOnsidmFsdWVzIjp7Im1heCI
6MTMyOC41NDIsInAoOTApIjo1MjAuMjcxLCJwKDk1KSI6NjA1LjQ2Mzk5OTk5OTk5OTksImF2Zy
I6MjcyLjA1MjU1MzY5MzM1OCwibWluIjoxMzkuOTU4LCJtZWQiOjI0NC45MDN9LCJ0eXBlIjoid
HJlbmQiLCJjb250YWlucyI6InRpbWUifSwiaHR0cF9yZXFzIjp7InR5cGUiOiJjb3VudGVyIiwi
Y29udGFpbnMiOiJkZWZhdWx0IiwidmFsdWVzIjp7InJhdGUiOjUwLjMzODc1Mzg3ODY3NzU0LCJ
jb3VudCI6MTYxMX0sInRocmVzaG9sZHMiOnsicmF0ZTwxMDAiOnsib2siOnRydWV9fX0sImh0dH
BfcmVxX3Rsc19oYW5kc2hha2luZyI6eyJjb250YWlucyI6InRpbWUiLCJ2YWx1ZXMiOnsiYXZnI
```

```
jowLjU1NTQ2MzY4NzE1MDgzNzksIm1pbiI6MCwibWVkIjowLCJtYXgiOjU4LjUzMiwicCg5MCki
OjAsInAoOTUpIjowfSwidHlwZSI6InRyZW5kIn0sImNoZWNrcyI6eyJ0eXBlIjoicmF0ZSIsImN
vbnRhaW5zIjoiZGVmYXVsdCIsInZhbHVlcyI6eyJyYXRlIjowLCJwYXNzZXMiOjAsImZhaWxzIj
oyNjZ9fSwiZGF0YV9yZWNlaXZlZCI6eyJ0eXBlIjoiY291bnRlciIsImNvbnRhaW5zIjoiZGF0Y
SIsInZhbHVlcyI6eyJjb3VudCI6NDg3OTcxNSwicmF0ZSI6MTUyNDc1Ljk2MDUxMDkxOTI4fX0s
Imh0dHBfcmVxX2Nvbm5lY3RpbmciOnsidHlwZSI6InRyZW5kIiwiY29udGFpbnMiOiJ0aW1lIiw
idmFsdWVzIjp7Im1pbiI6MCwibWVkIjowLCJtYXgiOjM0LjY0NCwicCg5MCkiOjAsInAoOTUpIj
oyNC43MTEsImF2ZyI6MS42OTI5MzYwNjQ1NTYxNzY2fX0sImh0dHBfcmVxX2R1cmF0aW9uIjp7I
nR5cGUiOiJ0cmVuZCIsImNvbnRhaW5zIjoidGltZSIsInZhbHVlcyI6eyJwKDkwKSI6NTI1LjU3
LCJwKDk1KSI6NjEzLjkxMjQ5OTk5OTk5OTksImF2ZyI6Mjc1Ljk3NDg0MTA5MjQ4OTYsIm1pbiI
6MTQwLjAwOSwibWVkIjoyNTAuNjg5LCJtYXgiOjEzNDEuMjk3fSwidGhyZXNob2xkcyI6eyJwKD
kwKTw1MDAiOnsib2siOmZhbHNlfX19LCJodHRwX3JlcV9zZW5kaW5nIjp7InR5cGUiOiJ0cmVuZ
CIsImNvbnRhaW5zIjoidGltZSIsInZhbHVlcyI6eyJwKDk1KSI6MC4xMDQ1LCJhdmciOjAuMDQ1
NTcxNjk0NTk5NjI3NzEsIm1pbiI6MC4wMDksIm1lZCI6MC4wMzUsIm1heCI6MC4yNjQsInAoOTA
pIjowLjA4N319LCJkYXRhX3NlbnQiOnsiY29udGFpbnMiOiJkYXRhIiwidmFsdWVzIjp7ImNvdW
50IjoyMjM2MjUsInJhdGUiOjY5ODcuNTg3NzMxOTE3NjA2fSwidHlwZSI6ImNvdW50ZXIifSwia
HR0cF9yZXFfYmxvY2tlZCI6eyJ0eXBlIjoidHJlbmQiLCJjb250YWlucyI6InRpbWUiLCJ2YWx1
ZXMiOnsibWluIjowLCJtZWQiOjAuMDAzLCJtYXgiOjgyLjA5NCwicCg5MCkiOjAuMDA5LCJwKDk
1KSI6MjUuMjIxNSwiYXZnIjoyLjI2MTE0NTg3MjEyOTE1fX0sInZ1c19tYXgiOnsidHlwZSI6Im
dhdWdlIiwiY29udGFpbnMiOiJkZWZhdWx0IiwidmFsdWVzIjp7InZhbHVlIjo1MCwibWluIjo1M
CwibWF4Ijo1MH19LCJ2dXMiOnsidHlwZSI6ImdhdWdlIiwiY29udGFpbnMiOiJkZWZhdWx0Iiwi
dmFsdWVzIjp7InZhbHVlIjoyNywibWluIjoxLCJtYXgiOjI3fX0sImh0dHBfcmVxX3JlY2Vpdml
uZyI6eyJ0eXBlIjoidHJlbmQiLCJjb250YWlucyI6InRpbWUiLCJ2YWx1ZXMiOnsicCg5MCkiOj
ExLjE2MiwicCg5NSkiOjEzLjk4ODUsImF2ZyI6My44NzY3MTU3MDQ1MzEzNDI1LCJtaW4iOjAuM
DE5LCJtZWQiOjAuMTMsIm1heCI6MzIuOTQ2fX0sIml0ZXJhdGlvbnMiOnsidHlwZSI6ImNvdW50
ZXIiLCJjb250YWlucyI6ImRlZmF1bHQiLCJ2YWx1ZXMiOnsiY291bnQiOjI0OCwicmF0ZSI6Ny4
3NDkyMzA4ODg4MzQyODI1fX0sImh0dHBfcmVxX2ZhaWxlZCI6eyJ0eXBlIjoicmF0ZSIsImNvbn
RhaW5zIjoiZGVmYXVsdCIsInZhbHVlcyI6eyJwYXNzZXMiOjAsImZhaWxzIjoxNjExLCJyYXRlI
jowfSwidGhyZXNob2xkcyI6eyJyYXRlPDAuMDEiOnsib2siOnRydWV9fX0sIml0ZXJhdGlvbl9k
dXJhdGlvbiI6eyJ0eXBlIjoidHJlbmQiLCJjb250YWlucyI6InRpbWUiLCJ2YWx1ZXMiOnsibWF
4IjoyNTQzLjAwMTk0NywicCg5MCkiOjE5MDIuMjk1MjA3OCwicCg5NSkiOjE5NjEuMjM3MDI3Mz
UsImF2ZyI6MTY1Mi44NTk3ODE4OTUxNjE5LCJtaW4iOjE0MDMuNDQyNTc5LCJtZWQiOjE2MTUuO
TM1MjYyOTk5OTk5OH19LCJodHRwX3JlcV9kdXJhdGlvbntleHBlY3RlZF9yZXNwb25zZTp0cnVl
fSI6eyJ0eXBlIjoidHJlbmQiLCJjb250YWlucyI6InRpbWUiLCJ2YWx1ZXMiOnsiYXZnIjoyNzU
uOTc0ODQxMDkyNDg5NiwibWluIjoxNDAuMDA5LCJtZWQiOjI1MC42ODksIm1heCI6MTM0MS4yOT
csInAoOTApIjo1MjUuNTcsInAoOTUpIjo2MTMuOTEyNDk5OTk5OTk5OX19fX0=</item><
/property></properties></testcase>
</testsuite >
</testsuites >
```


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 and 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 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 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, and for that the first step is to follow the instructions in v1 or v2 (depending on your usage) to include authentication parameters in the following requests.

Junit results

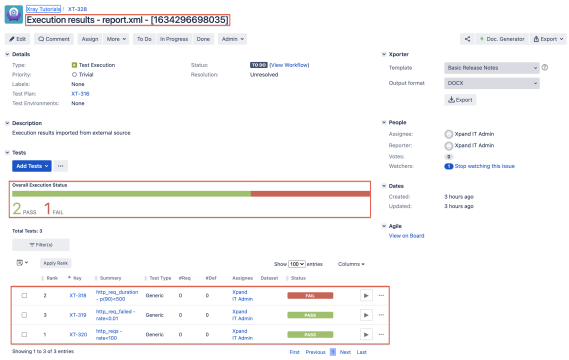
We will use the API request with the addition of some parameters that will set the Project to where the results will be uploaded and the Test Plan that will hold the Execution results.

In the first version of the API, the authentication used a login and password (not the token that is used in Cloud).

```
curl -H "Content-Type: multipart/form-data" -u admin:admin -F
"file=@xrayJunit.xml" http://yourserver/rest/raven/1.0/import/execution
/junit?projectKey=XT&testPlanKey=XT-316
```

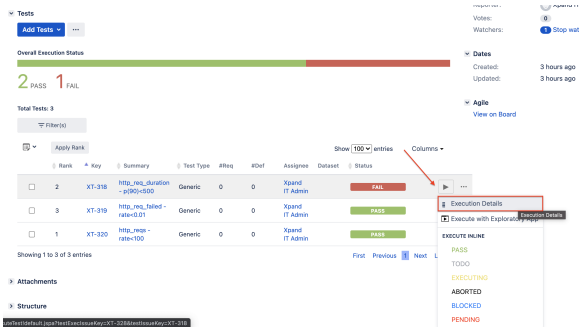
With this command we are creating a new Test Execution that will have the results of the Tests that were executed and it will be associated to the Test Plan XT-316.

Once uploaded the Test Execution will look like the example below

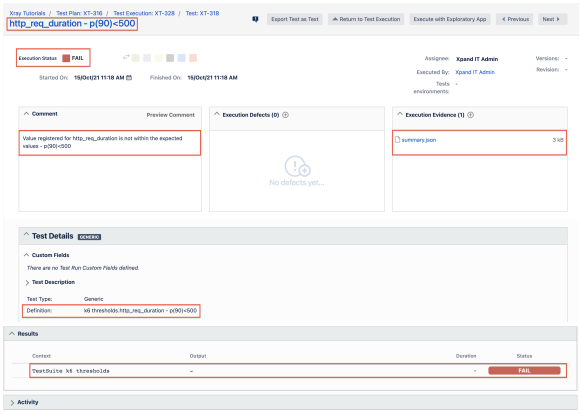


We can see that a new Test Execution was created with a summary automatically generated and 3 Tests were added with the corresponding status and summary (matching the information from the xml report).

In order to check the details we click on the details icon next to each Test



It 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 - Fail, 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 as breached the KPIs (in case of failure).

Bringing the information of performance tests to your project will allow a complete view over the Testing process and bring that visibility up front for the team to have all the elements necessary to deliver a quality product.

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