

# Performance and load testing with Gatling



## What you'll learn

### Pre-requisites

- [KPI](#)
  - Define tests using Gatling
  - Define KPIs in the test and push the test report to Xray
- [Integrating with Jira](#)
  - Validate that the test results are available in Jira
- [Authentication](#)
- [Xray Json results](#)

### Tips

### Source code for this tutorial

### References

- code is available in [GitHub](#)

## Overview

Locust is a load testing tool that uses Scala to write tests.

Using an expressive DSL and having scenarios defined in code makes its code suitable to be kept in a version control system.

Gatling also has an [Enterprise](#) version, that was formerly known as Gatling FrontLine, it is a management interface for Gatling, that includes advanced metrics and advanced features for integration and automation.

## Pre-requisites

For this example, we will use [Gatling](#) to define a series of Performance tests using the [Maven plugin](#) available.

We will use the [assertions](#) to define KPIs that will fail or succeed the tests.

You will need:

- Access to a [demo site](#) that we will test
- Understanding and defining Key Performance Indicators (KPI) for our performance tests
- Maven with Scala environment and Gatling installed

To start, define a simple load test in [Gatling](#) 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 [Gatling](#) please follow the [documentation](#).

In the documentation you will find that there are several ways to use the tool, in our case we are using the [Maven Plugin](#) available, since we will use this code in a CI/CD tool further ahead.

Before jumping into the code you can find below the *pom.xml* file content for the project after it has been configured with the Maven plugin.

## pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.xpandit.xray.tutorials</groupId>
  <artifactId>gatling-perf</artifactId>
  <version>3.6.1</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <gatling.version>${project.version}</gatling.version>
    <gatling-maven-plugin.version>3.1.2</gatling-maven-plugin.version>
    <maven-jar-plugin.version>3.2.0</maven-jar-plugin.version>
    <scala-maven-plugin.version>4.4.1</scala-maven-plugin.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>io.gatling.highcharts</groupId>
      <artifactId>gatling-charts-highcharts</artifactId>
      <version>${gatling.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <testSourceDirectory>src/test/scala</testSourceDirectory>
    <sourceDirectory>main/scala</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>${maven-jar-plugin.version}</version>
      </plugin>
      <plugin>
        <groupId>net.alchim31.maven</groupId>
        <artifactId>scala-maven-plugin</artifactId>
        <version>${scala-maven-plugin.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>testCompile</goal>
            </goals>
            <configuration>
              <jvmArgs>
                <jvmArg>-Xss100M</jvmArg>
              </jvmArgs>
              <args>
                <arg>-target:jvm-1.8</arg>
                <arg>-deprecation</arg>
                <arg>-feature</arg>
                <arg>-unchecked</arg>
                <arg>-language:implicitConversions</arg>
                <arg>-language:postfixOps</arg>
              </args>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>io.gatling</groupId>
        <artifactId>gatling-maven-plugin</artifactId>
        <version>${gatling-maven-plugin.version}</version>
      </plugin>
    </plugins>
  </build>
</project>
```

```
    </plugins>
  </build>
</project>
```

The tests, as we have defined above, will target 3 different endpoints. We started by extending the *Simulation* class of Gatling signalling that this class will hold our simulation.

#### blazemeterPerf.scala

```
...
class MySimulation extends Simulation {
...

```

Next we have created 3 objects that are mirroring the operations we want to exercise:

- Login
- Reserve
- Purchase

For each we have defined the endpoint we want to access, the parameters needed to perform the operation and a waiting time at the end to simulate a real user, as you can see below:

#### blazemeterPerf.scala

```
...object Login {
val login = exec(http("Access Reserve").post("http://blazedemo.com/reserve.
php")
.formParam("fromPort", "Paris")
.formParam("toPort", "Buenos+Aires"))
.pause(2, 3)
}

object Reserve {
val reserve = exec(http("Access Reserve").post("http://blazedemo.com
/reserve.php")
.formParam("fromPort", "Paris")
.formParam("toPort", "Buenos+Aires"))
.pause(2, 3)
}

object Purchase {
val purchase = exec(http("Access Purchase").post("http://blazedemo.com
/purchase.php")
.formParam("fromPort", "Paris")
.formParam("toPort", "Buenos+Aires")
.formParam("airline", "Virgin+America")
.formParam("flight", "43")
.formParam("price", "472.56"))
.pause(2, 3)
}...
```

Another thing we will need is the protocol definition for the simulation, in our case we are using HTTP protocol and have defined it with some default values, notice nevertheless the *baseUrl* pointing to the endpoint of the application.

#### blazemeterPerf.scala

```
...
val httpProtocol = http
    .baseUrl("http://blazedemo.com")
    .acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8")
    .doNotTrackHeader("1")
    .acceptLanguageHeader("en-US,en;q=0.5")
    .acceptEncodingHeader("gzip, deflate")
    .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101 Firefox/16.0")
...
```

Finally we need to define the user scenarios, in our case it will be a one-to-one correspondence, we have created one user per operation thus creating 3 different scenarios:

- LoginUsers - Users that will simulate login operations
- ReserveUsers - Users that will simulate reserve operations
- PurchaseUsers - Users that will simulate purchase operations

This will help define the different definitions of profile injections of users we want to simulate.

#### blazemeterPerf.scala

```
...
val loginUsers = scenario("LoginUsers").exec(Login.login)
val reserveUsers = scenario("ReserveUsers").exec(Reserve.reserve)
val purchaseUsers = scenario("PurchaseUsers").exec(Purchase.purchase)
...
```

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

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

In our case we have defined it like below:

```
setUp(
    loginUsers.inject(atOnceUsers(10)),
    reserveUsers.inject(rampUsers(2).during(10.seconds)),
    purchaseUsers.inject(rampUsers(1).during(10.seconds))
).protocols(httpProtocol)
```

In more detail:

- loginUsers.inject(atOnceUsers(10)) - We are inserting 10 loginUsers simultaneously
- reserveUsers.inject(rampUsers(2).during(10.seconds)) - We are ramping up 2 reserveUsers for 10 seconds
- purchaseUsers.inject(rampUsers(1).during(10.seconds)) - We are ramping up 1 purchaseUsers for 10 seconds
- All of the above injections will occur in parallel
- .protocols(httpProtocol) - We will perform all of the above using the HttpProtocol defined earlier

To execute the tests you can use several ways, in our case we are using the command line.

```
mvn gatling:test
```

The command line output will look like this:

[illegible]

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's 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 (generate Xray Json results)

In order to do that we will use the assertions available in Gatling and build a converter function that will convert the assertions.json produced by Gatling into an Xray Json Test Result file ready to be imported to Xray.

## KPI

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

Gatling has out of the box the ability to define assertions, in our case we want to define the following ones globally:

- the 90 percentile exceed 5000ms 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 assertions in the `setUp`:

```
setUp(
    loginUsers.inject(atOnceUsers(10)),
    reserveUsers.inject(rampUsers(2).during(10.seconds)),
    purchaseUsers.inject(rampUsers(1).during(10.seconds))
).assertions(
    global.responseTime.percentile(90).lt(5000),
    global.failedRequests.count.lte(0),
    global.requestsPerSec.lt(500)
).protocols(httpProtocol)
```

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

```
Simulation performance: MySimulation completed in 8 seconds
Parsing log file(s):
Parsing log file(s) done
Generating reports...

-- Global Information --
> request count 36 (0m0s) N/A
> min response time 108 (0m0s) N/A
> max response time 3328 (0m0s) N/A
> mean response time 1772 (0m0s) N/A
> std deviation 1433 (0m0s) N/A
> response time 50th percentile 532 (0m0s) N/A
> response time 75th percentile 1248 (0m0s) N/A
> response time 90th percentile 1968 (0m0s) N/A
> response time 95th percentile 2400 (0m0s) N/A
> mean requests/sec 2.880 (0m0s) N/A
-- Response Time Distribution --
> < 400 ms 14 ( 38%)
> 400 ms <= 1200 ms 8 ( 22%)
> > 1200 ms 12 ( 34%)
> total 36

Reports generated in 0s.
Please open the following file: /Users/cristianomendes/Documents/projects/gatling/tutorial-maven-scala-gatling/target/gatling/mysimulation-2021100910272918/index.html
Global: 95th percentile of response time is less than 20000 ms : true
Global: count of failed events is less than or equal to 0.0 : true
Global: mean requests per second is less than 5000.0 : true
-----
[Info] SUCCESS
[Info] Total time: 22.460 s
[Info] Finished at: 2021-10-09T17:27:30+01:00
[Info]
```

## Generate Xray Json

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.

Gatling produces HTML files with a detailed report of the tests and also some Json files with the details of the assertions and the stats of the respective tests (valuable to perform a post analysis but hard to convert into proper pass or fail result).

We need to produce a result that will hold all the information produced and will bring value to our project. To do so, we are going to create an [Xray Json](#) report to hold these results.

In Xray we have defined one Test and one Test Plan:

- XT-245 - Test Plan that will hold all executions of the performance Tests
- XT-246 - Performance Test, associated to the above Test Plan, that we will use to link the results back to

This will serve to centralize all the results of the performance executions in each sprint and bring visibility to the team of those results providing an overall view of the status of the project.

Gatling generates several files but for this example we will convert the assertions.json (in the ./target/gatling/mysimulation-DATE/js, you will find two files: assertions.json and stats.json).

The assertions file will have the details regarding the assertions we have defined in the Test, so we are going to convert that into a valid Xray Json file with a small script developed in python.

### conver2XrayJson.py

```
import json, argparse
from base64 import b64encode

class convert2XrayJson:
    def injectFile(self, fileName):
        with open(str(fileName), 'rb') as open_file:
            byte_content = open_file.read()

        return b64encode(byte_content).decode('utf-8')

    def appendToXrayResult(self, data, testkey, metric, name, value,
comment, status, projectkey, testplankey, evidencefile):
        done = False
        if len(data['tests']) > 0:
            for tests in data['tests']:
                for key, value in tests.items():
                    if key == 'testKey' and value == testkey:
                        tests['results'].append({
                            'name': metric + ' for ' + name,
                            'log': comment,
                            'status': 'PASSED' if status else 'FAILED'
                        })
                    done = True

        if not done:
            info = {
                'info': {
                    'summary': ' Perf test',
                    'description': 'Perf test',
```

```

        'project': projectkey,
        'testPlanKey': testplankey,
    },
}

data['tests'].append({
    'testKey': testkey,
    'comment': metric,
    'status': 'PASSED' if status else 'FAILED',
    'results': [
        {
            'name': metric + ' for ' + name,
            'log': comment,
            'status': 'PASSED' if status else 'FAILED'
        }
    ],
    'evidences': [
        {
            'data': self.injectFile(evidencefile),
            'filename': evidencefile.rsplit('/', 1)[-1],
            'contentType': 'application/json'
        }
    ]
})

data.update(info)

```

## \_\_\_\_\_

```

parser = argparse.ArgumentParser(description='Helper to convert Gatling
assertions output to Xray Json')
parser.add_argument('--gatlingFile', dest='gatlingfile', type=str,
help='Path of the Gatling assertion file')
parser.add_argument('--outputFile', dest='outputfile', type=str,
help='Name of the Xray Json output file')
parser.add_argument('--testKey', dest='testkey', type=str, help='Key of
the test to associate in Xray')
parser.add_argument('--testPlan', dest='testplan', type=str, help='Test
Plan key to associate in Xray')
parser.add_argument('--jiraProject', dest='jiraproject', type=str,
help='Jira project key')
parser.add_argument('--evidenceFile', dest='evidencefile', type=str,
help='File to add as an evidence')

```

```
args = parser.parse_args()
```

```

gatlingfile = args.gatlingfile
outputfile = args.outputfile
testkey = args.testkey
testplan = args.testplan
jiraproj = args.jiraproject
evidencefile = args.evidencefile

```

```

data = {}
data['tests'] = []
cXray = convert2XrayJson()

```

```

with open(gatlingfile) as json_file:
    filedata = json.load(json_file)
    for p in filedata['assertions']:
        cXray.appendToXrayResult(data, testkey, p['target'], p['path'],
testplan, p['message'], p['result'], jiraproj, testplan, evidencefile)

```

```

with open(outputfile, 'w') as outfile:
    json.dump(data, outfile)

```

The usage is straight forward and can be explained with the helper function available if you use:

```
python convert2XrayJson.py -h
usage: convert2XrayJson.py [-h] [--gatlignFile
GATLINGFILE]                        [--outputFile OUTPUTFILE] [--
testKey TESTKEY]                    [--testPlan TESTPLAN] [--jiraProject
JIRAPROJECT]                       [--evidenceFile EVIDENCEFILE]
```

Helper to convert Gatling assertions output to Xray Json

optional arguments:

```
-h, --help          show this help message and exit
--gatlingFile GATLINGFILE
                    Path of the Gatling assertion file
--outputFile OUTPUTFILE
                    Name of the Xray Json output file
--testKey TESTKEY   Key of the test to associate in Xray
--testPlan TESTPLAN Test Plan key to associate in Xray
--jiraProject JIRAPROJECT
                    Jira project key
--evidenceFile EVIDENCEFILE
                    File to add as an evidence
```

One example of the execution of the tool is:

```
python convert2XrayJson.py --gatlingFile /target/gatling/mysimulation-20211007103948126/js/assertions.json --outputFile xrayJson.json --testKey 'XT-246' --testPlan 'XT-245' --jiraProject XT --evidenceFile /target/gatling/mysimulation-20211007103948126/js/stats.json
```

The Xray Json file generated is:

xrayResults.json

[illegible]



[illegible]

```
sIjogMTAxLAogICAgICAgICJvayI6IDeWMSwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzMSI6IHsKICAgICAgICAidG90YWwiOiAlNzAsCiAgICAgICAgIm9rIjogNTcwLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAicGVyY2VudGlsZXMyIjogewogICAgICAgICJ0b3RhbCI6IDU5NiWkICAgICAgICAib2siOiAlOTYsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJwZXJjZW50aWxlczMioiB7CiAgICAgICAgInRvdGFsIjogNjM2LAogICAgICAgICJvayI6IDYzNiWkICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzNCI6IHsKICAgICAgICCAidG90YWwiOiA2NDYsCiAgICAgICAgIm9rIjogNjQ2LAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAiZ3JvdXAxiJogewogICAgIm5hbWUiOiAidCA8IDgwMCBtcyIsCiAgICAiY291bnQiOiAxMiwKICAgICJwZXJjZW50YWdlIjogMTAwCn0sCiAgICAiZ3JvdXAyIjogewogICAgIm5hbWUiOiAiODAwIGlzIDwgdCA8IDeYMDAgbXMiLAogICAgImNvdW50IjogMCwKICAgICJwZXJjZW50YWdlIjogMAP9LAogICAgImdyb3VwMyI6IHsKICAgICJuYWllIjogInQgPiAxMjAwIGlzIiwKICAgICJjb3VudCI6IDAsCiAgICAicGVyY2VudGFzSI6IDAKfSwKICAgICJncm9lcDQiOiB7CiAgICAibmFtZSI6IDCMjYwLsZWQilLAogICAgImNvdW50IjogMCwKICAgICJwZXJjZW50YWdlIjogMAP9LAogICAgIm1lYW50dWllZXJPZjJlcXVlc3RzUGVyU2Vjb25kIjogewogICAgICAgICJ0b3RhbCI6IDeUnSwKICAgICAgICAgICAib2siOiAxLjUsCiAgICAgICAgImtvIjogMAogICAgfQp9CiAgICB9LCJyZXFFYWNjZXNzLXB1cmNoYXNlLWMxNGJmIjogewogICAgICAgICJ0eXBlIjogIlJFUUVFU1QiLAogICAgICAgICJuYWllIjogIkFjY2VzcyBQdXJjaGFzZSBSZWRpcmVjdCAxiIiwKInBhdGgiOiAiQWNjZXNzIFB1cmNoYXNlIFJlZGlyZWNOIDeilaOicGF0aEZhcm1hdHRlZCI6ICJyZXFFYWNjZXNzLXB1cmNoYXNlLWMxNGJmIiwKInN0YXRzIjogewogICAgIm5hbWUiOiAiQWNjZXNzIFB1cmNoYXNlIFJlZGlyZWNOIDeilaOgICAgIm5lbWlck9mUmVxdWVzdHMiOiB7CiAgICAgICAgInRvdGFsIjogMSwKICAgICAgICAib2siOiAxLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAibWluUmVzcG9uc2VUaWllIjogewogICAgICAgICJ0b3RhbCI6IDQ4MCwKICAgICAgICAib2siOiA0ODAsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJtYXhSZXNwb25zZVRpbWUiOiB7CiAgICAgICAgInRvdGFsIjogNDgwLAogICAgICAgICJvayI6IDQ4MCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgIm1lYW5SZXNwb25zZVRpbWUiOiB7CiAgICAgICAgInRvdGFsIjogNDgwLAogICAgICAgICJvayI6IDQ4MCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzMSI6IHsKICAgICAgICAidG90YWwiOiA0ODAsCiAgICAgICAgIm9rIjogNDgwLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAicGVyY2VudGlsZXMyIjogewogICAgICAgICJ0b3RhbCI6IDQ4MCwKICAgICAgICAib2siOiA0ODAsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJwZXJjZW50aWxlczMioiB7CiAgICAgICAgInRvdGFsIjogNDgwLAogICAgICAgICJvayI6IDQ4MCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzNCI6IHsKICAgICAgICAidG90YWwiOiA0ODAsCiAgICAgICAgIm9rIjogNDgwLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAiZ3JvdXAxiJogewogICAgIm5hbWUiOiAidCA8IDgwMCBtcyIsCiAgICAiY291bnQiOiAxLAogICAgInBlcmNlbnRpbGVzNCI6IHsKICAgICAgICAidG90YWwiOiA0ODAsCiAgICAgICAgIm9rIjogNDgwLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAiZ3JvdXAyIjogewogICAgIm5hbWUiOiAidCA8IDeYMDAgbXMiLAogICAgImNvdW50IjogMCwKICAgICJwZXJjZW50YWdlIjogMAP9LAogICAgImdyb3VwNCI6IHsKICAgICJuYWllIjogImZhaWxlc3RzSI6ICAgICAiY291bnQiOiAwLAogICAgInBlcmNlbnRhZ2UiOiAwCn0sCiAgICAibWVhbk51bWJlck9mUmVxdWVzdHMQZXRJTZWNVbmQiOiB7CiAgICAgICAgInRvdGFsIjogMC4xMjUsCiAgICAgICAgIm9rIjogMC4xMjUsCiAgICAgICAgImtvIjogMAogICAgfQp9CiAgICB9Cn0KCn0=\", \"filename\": \"stats.json\"}}]}}
```

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 produced Xray Json reports with the result of the tests, it is now a matter of importing those results to your Jira instance. You can do this 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.

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](#). To do that, 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.

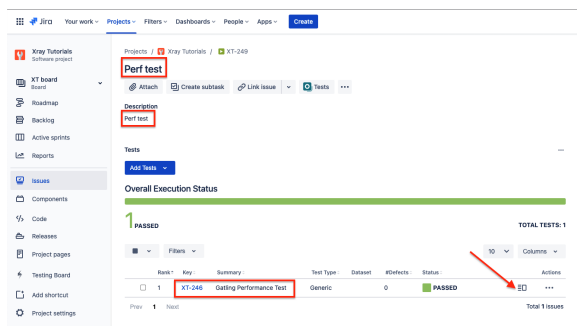
## Xray Json results

Once you have the token we will use it in the API request with the definition of some common fields on the Test Execution, such as the target project, project version, etc.

```
curl -H "Content-Type: application/json" -X POST -H "Authorization: Bearer $token" --data @xrayResults.json https://xray.cloud.getxray.app/api/v2/import/execution
```

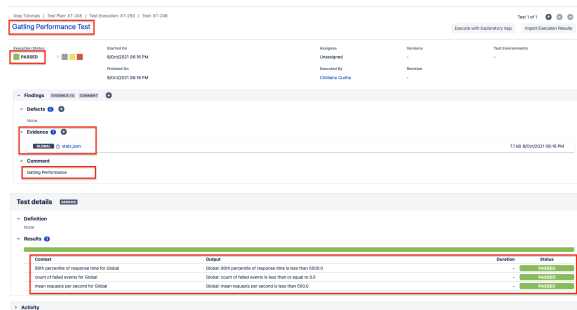
With this command we are creating a new Test Execution that will have the results of the Tests that were executed.

Once uploaded the Test Execution will look like the example below



With Title and Description we have defined in the code and linked to the Tests we have created beforehand to hold Performance results.

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



In the details we have the following relevant information:

- Execution Status - Passed, this indicates the overall status of the execution of the Performance Tests
- Evidence - Json file produced by Gatling of the stats of the Test for future analysis
- Comment - Shows the performance Test that was executed
- Results - Detailed results with information of the KPI's defined and why they were considered successful.

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 by that environment later on. A Test Environment can be a testing stage (e.g. dev, staging, pre-production, production) or an identifier of the device/application used to interact with the system (e.g. browser, mobile OS).

---

## References

- <https://gatling.io/>
- <https://gatling.io/docs/gatling/>
- <https://gatling.io/docs/gatling/tutorials/installation/>
- <https://blazedemo.com/>