Performance and load testing with Gatling

What you'll learn

- Define tests using Gatling
- 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

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

We will need:

- Access to a demo site that we aim to test
- Understand and define Keep Performance Indicators (KPI) for our performance tests
- Maven with Scala environment and Gatling installed

We will start to 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, on our case we are using the Maven Plugin available, thinking that 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 all the configuration as been done 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>
```

The tests, as we have defined above, will target three different endpoints, for that we have 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 three 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 three 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 defining 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 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 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)
```

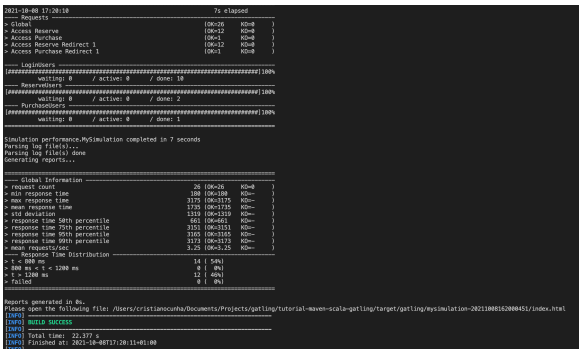
In more detail:

- 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

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

```
mvn gatling:test
```

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 (generate Xray Json results)

In order to do that we will use the assertions available in Gatling and build a converter functions 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 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.

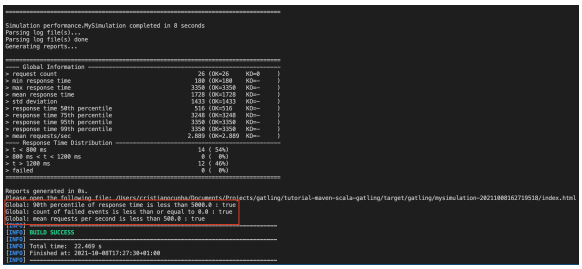
Gatling have out of the box the ability to define assertions, in our case we want to define the following ones globally:

- 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:



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.

First let us explain the approach we are taking towards these performance Tests, in Xray we have defined one Test and one TestPlan:

- XT-329 - TestPlan that will hold all executions of the performance Tests
- XT-330 - 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 in 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 is explained with an helper function available if you use:

```
python convert2XrayJson.py -h
usage: convert2XrayJson.py [-h] [--gatlingFile
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-330' --testPlan 'XT-329' --jiraProject XT --evidenceFile /target
/gatling/mysimulation-20211007103948126/js/stats.json
```

The Xray Json file generated is:

xrayResults.json


```
{"info": {"project": "XT", "testPlanKey": "XT-245", "description": "Perf
test", "summary": " Perf test"}, "tests": [{"comment": "90th percentile of
response time", "status": "PASSED", "results": [{"status": "PASSED",
"name": "90th percentile of response time for Global", "log": "Global:
90th percentile of response time is less than 5000.0"}, {"status":
"PASSED", "name": "count of failed events for Global", "log": "Global:
count of failed events is less than or equal to 0.0"}, {"status":
"PASSED", "name": "mean requests per second for Global", "log": "Global:
mean requests per second is less than 500.0"}], "testKey": "XT-246",
"evidences": [{"contentType": "application/json", "data":
"ewogICJ0eXBlIjogIkdST1VQIiwKIm5hbWUiOiAiR2xvYmFsIEluZm9ybWF0aW9uIiwKInBhdG
giOiAiIiwKInBhdGhGb3JtYXR0ZWQiOiAiZ3JvdXBfbWlzc2luZy1uYW1lLWIwNmQxIiwKInN0Y
XRzIjogewogICAgIm5hbWUiOiAiR2xvYmFsIEluZm9ybWF0aW9uIiwKICAgICJudW1iZXJPZlJl
cXVlc3RzIjogewogICAgICAgICJ0b3RhbCI6IDI2LAogICAgICAgICJvayI6IDI2LAogICAgICA
gICJrbyI6IDAKICAgIH0sCiAgICAibWluUmVzcG9uc2VUaW1lIjogewogICAgICAgICJ0b3RhbC
I6IDE3OSwKICAgICAgICAib2siOiAxNzksCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJtY
XhSZXNwb25zZVRpbWUiOiB7CiAgICAgICAgInRvdGFsIjogMzIzNywKICAgICAgICAib2siOiAz
MjM3LAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAibVhblJlc3BvbnNlVGltZSI6IHsKICA
gICAgICAidG90YWwiOiAxNzMxLAogICAgICAgICJvayI6IDE3MzEsCiAgICAgICAgImtvIjogMA
ogICAgfSwKICAgICJzdGFuZGFyZERldmlhdGlvbiI6IHsKICAgICAgICAidG90YWwiOiAxMzM0L
AogICAgICAgICJvayI6IDEzMzQsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJwZXJjZW50
aWxlczEiOiB7CiAgICAgICAgInRvdGFsIjogNjM4LAogICAgICAgICJvayI6IDYzOCwKICAgICA
gICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzMiI6IHsKICAgICAgICAidG90YWwiOi
AzMTU3LAogICAgICAgICJvayI6IDMxNTcsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJwZ
XJjZW50aWxlczMiOiB7CiAgICAgICAgInRvdGFsIjogMzIxOCwKICAgICAgICAib2siOiAzMjE4
LAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAicGVyY2VudGlsZXM0IjogewogICAgICAgICJ
0b3RhbCI6IDMyMzcsCiAgICAgICAgIm9rIjogMzIzNywKICAgICAgICAia28iOiAwCiAgICB9LA
```

ogICAgImdyb3VwMSI6IHsKICAgICJuYW1lIjogInQgPCA4MDAgbXMiLAogICAgImNvdW50IjogM
TQsCiAgICAicGVyY2VudGFnZSI6IDU0Cn0sCiAgICAiZ3JvdXAyIjogewogICAgIm5hbWUiOiAi
ODAwIG1zIDwgdCA8IDEyMDAgbXMiLAogICAgImNvdW50IjogMCwKICAgICJwZXJjZW50YWdlIjo
gMAp9LAogICAgImdyb3VwMyI6IHsKICAgICJuYW1lIjogInQgPiAxMjAwIG1zIiwKICAgICJjb3
VudCI6IDEyLAogICAgInBlcmNlbnRhZ2UiOiA0Ngp9LAogICAgImdyb3VwNCI6IHsKICAgICJuY
W1lIjogImZhaWxlZCIsCiAgICAiY291bnQiOiAwLAogICAgInBlcmNlbnRhZ2UiOiAwCn0sCiAg
ICAibWVhbk51bWJlck9mUmVxdWVzdHNQZXJTZWNvbmQiOiB7CiAgICAgICAgInRvdGFsIjogMy4
yNSwKICAgICAgICAib2siOiAzLjI1LAogICAgICAgICJrbyI6IDAKICAgIH0KfSwKImNvbnRlbn
RzIjogewoicmVxX2FjY2Vzcy1yZXNlcnZlLTQ2MzU0IjogewogICAgICAgICJ0eXBlIjogIlFU
VVFU1QiLAogICAgICAgICJuYW1lIjogIkFjY2VzcyBSZXNlcnZlIiwKInBhdGgiOiAiQWNjZXNz
IFJlc2VydmUiLAoicGF0aEZvcm1hdHRlZCI6ICJyZXFfYWNjZXNzLXJlc2VydmUtNDYzNTQiLAo
ic3RhdHMiOiB7CiAgICAibmFtZSI6ICJBY2Nlc3MgUmVzZXJ2ZSIsCiAgICAibnVtYmVyT2ZSZX
F1ZXN0cyI6IHsKICAgICAgICAiaG90YWwiOiAxMiwKICAgICAgICAib2siOiAxMiwKICAgICAgI
CAia28iOiAwCiAgICB9LAogICAgIm1pblJlc3BvbnNlVGltZSI6IHsKICAgICAgICAiaG90YWwi
OiAxNzksCiAgICAgICAgIm9rIjogMTc5LAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAibWF
4UmVzcG9uc2VUaW1lIjogewogICAgICAgICJ0b3RhbCI6IDMyMzcsCiAgICAgICAgIm9rIjogMz
IzNywKICAgICAgICAia28iOiAwCiAgICB9LAogICAgIm1lYW5SZXNwb25zZVRpbWUiOiB7CiAgI
CAgICAgInRvdGFsIjogMjkxOSwKICAgICAgICAib2siOiAyOTE5LAogICAgICAgICJrbyI6IDAK
ICAgIH0sCiAgICAic3RhbmRhcmREZXZpYXRpb24iOiB7CiAgICAgICAgInRvdGFsIjogODI3LAo
gICAgICAgICJvayI6IDgyNywKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbG
VzMSI6IHsKICAgICAgICAidG90YWwiOiAzMTU2LAogICAgICAgICJvayI6IDMxNTYsCiAgICAgI
CAgImtvIjogMAogICAgfSwKICAgICJwZXJjZW50aWxlcziIiOiB7CiAgICAgICAgInRvdGFsIjog
MzE2MSwKICAgICAgICAib2siOiAzMTYxLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAicGV
yY2VudGlsZXMzIjogewogICAgICAgICJ0b3RhbCI6IDMyMzcsCiAgICAgICAgIm9rIjogMzIzNy
wKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzNCI6IHsKICAgICAgICAid
G90YWwiOiAzMjM3LAogICAgICAgICJvayI6IDMyMzcsCiAgICAgICAgImtvIjogMAogICAgfSwK
ICAgICJncm91cDEiOiB7CiAgICAibmFtZSI6ICJ0IDwgODAwIG1zIiwKICAgICJjb3VudCI6IDE
sCiAgICAicGVyY2VudGFnZSI6IDgKfSwKICAgICJncm91cDIiOiB7CiAgICAibmFtZSI6ICI4MD
AgbXMgPCB0IDwgMTIwMCBtcyIsCiAgICAiY291bnQiOiAwLAogICAgInBlcmNlbnRhZ2UiOiAwC
n0sCiAgICAiZ3JvdXAzIjogewogICAgIm5hbWUiOiAidCA+IDEyMDAgbXMiLAogICAgImNvdW50
IjogMTEsCiAgICAicGVyY2VudGFnZSI6IDkyCn0sCiAgICAiZ3JvdXA0IjogewogICAgIm5hbWU
iOiAiZmFpbGVkIiwKICAgICJjb3VudCI6IDAsCiAgICAicGVyY2VudGFnZSI6IDAKfSwKICAgIC
JtZWFuTnVtYmVyT2ZSZXF1ZXN0c1BlclNlY29uZCI6IHsKICAgICAgICAidG90YWwiOiAxLjUsC
iAgICAgICAgIm9rIjogMS41LAogICAgICAgICJrbyI6IDAKICAgIH0KfQogICAgfSwicmVxX2Fj
Y2Vzcy1wdXJjaGFzZS0zZGE0YiI6IHsKICAgICAgICAidHlwZSI6ICJSRVFVRVNUIiwKICAgICA
gICAibmFtZSI6ICJBY2Nlc3MgUHVyY2hhc2UiLAoicGF0aCI6ICJBY2Nlc3MgUHVyY2hhc2UiLA
oicGF0aEZvcm1hdHRlZCI6ICJyZXFfYWNjZXNzLXB1cmNoYXNlLTNkYTRiIiwKInN0YXRzIjoge
wogICAgIm5hbWUiOiAiQWNjZXNzIFB1cmNoYXNlIiwKICAgICJudW1iZXJPZlJlcXVlc3RzIjog
ewogICAgICAgICJ0b3RhbCI6IDEsCiAgICAgICAgIm9rIjogMSwKICAgICAgICAia28iOiAwCiA
gICB9LAogICAgIm1pblJlc3BvbnNlVGltZSI6IHsKICAgICAgICAidG90YWwiOiAzMTU4LAogIC
AgICAgICJvayI6IDMxNTgsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJtYXhSZXNwb25zZ
VRpbWUiOiB7CiAgICAgICAgInRvdGFsIjogMzE1OCwKICAgICAgICAib2siOiAzMTU4LAogICAg
ICAgICJrbyI6IDAKICAgIH0sCiAgICAibWVhblJlc3BvbnNlVGltZSI6IHsKICAgICAgICAidG9
0YWwiOiAzMTU4LAogICAgICAgICJvayI6IDMxNTgsCiAgICAgICAgImtvIjogMAogICAgfSwKIC
AgICJzdGFuZGFyZERldmlhdGlvbiI6IHsKICAgICAgICAidG90YWwiOiAwLAogICAgICAgICJva
yI6IDAsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJwZXJjZW50aWxlcEiOiB7CiAgICAg
ICAgInRvdGFsIjogMzE1OCwKICAgICAgICAib2siOiAzMTU4LAogICAgICAgICJrbyI6IDAKICA
gIH0sCiAgICAicGVyY2VudGlsZXMyIjogewogICAgICAgICJ0b3RhbCI6IDMxNTgsCiAgICAgIC
AgIm9rIjogMzE1OCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzNCI6I
HsKICAgICAgICAidG90YWwiOiAzMTU4LAogICAgICAgICJvayI6IDMxNTgsCiAgICAgICAgImtv
IjogMAogICAgfSwKICAgICJwZXJjZW50aWxlcyI6IHsKICAgICAgICAidG90YWwiOiAxMTgsCi
AgICAgICAgIm9rIjogMTE4LAogICAgICAgICJrbyI6IDAKICAgIH0KfSwKImNvbnRlbnRzIjog
ewoicmVxX2FjY2Vzcy1yZXNlcnZlLTQ2MzU0IjogewogICAgICAgICJ0eXBlIjogIlFUVVFU1QiL
AogICAgICAgICJuYW1lIjogIkFjY2VzcyBSZXNlcnZlIiwKInBhdGgiOiAiQWNjZXNzIFJlc2
VydmUiLAoicGF0aEZvcm1hdHRlZCI6ICJyZXFfYWNjZXNzLXJlc2VydmUtNDYzNTQiLAoic3R
hdHMiOiB7CiAgICAibmFtZSI6ICJBY2Nlc3MgUmVzZXJ2ZSBSZXNlcnZlLS0wYzgzMyI
sCiJzdGF0cyI6IHsKICAgICJuYW1lIjogIkFjY2VzcyBSZXNlcnZlIFJlZGlyZWN0IDEiLAogIC
AgIm51bWJlck9mUmVxdWVzdHMiOiB7CiAgICAgICAgInRvdGFsIjogMTIsCiAgICAgICAgIm9rI
jogMTIsCiAgICAgICAgImtvIjogMAogICAgfSwKICAgICJtaW5SZXNwb25zZVRpbWUiOiB7CiAg
ICAgICAgInRvdGFsIjogMzUyLAogICAgICAgICJvayI6IDM1MiwKICAgICAgICAia28iOiAwCiA
gICB9LAogICAgIm1heFJlc3BvbnNlVGltZSI6IHsKICAgICAgICAidG90YWwiOiA2NDgsCiAgIC
AgICAgICJvayI6IDY0OCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInAxMjcsCiAgICAgI
CAgIm9rIjogNTI3LAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAibWVhblJlc3BvbnNlV
GltZSI6IHsKICAgICAgICAidG90YWwiOiA1MjcsCiAgICAgICAgIm9rIjogNTI3LAogICAgICAg
ICJrbyI6IDAKICAgIH0sCiAgICAic3RhbmRhcmREZXZpYXRpb24iOiB7CiAgICAgICAgInRvdGF

```
sIjogMTAxLAogICAgICAgICJvayI6IDEwMSwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgIn
BlcmNlbnRpbGVzMSI6IHsKICAgICAgICAidG90YWwiOiA1NzAsCiAgICAgICAgIm9rIjogNTcwL
AogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAicGVyY2VudGlsZXMyIjogewogICAgICAgICJ0
b3RhbCI6IDU5NiwKICAgICAgICAib2siOiA1OTYsCiAgICAgICAgImtvIjogMAogICAgfSwKICA
gICJwZXJjZW50aWxlczMiOiB7CiAgICAgICAgInRvdGFsIjogNjM2LAogICAgICAgICJvayI6ID
YzNiwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzNCI6IHsKICAgICAgI
CAidG90YWwiOiA2NDYsCiAgICAgICAgIm9rIjogNjQ2LAogICAgICAgICJrbyI6IDAKICAgIH0s
CiAgICAiZ3JvdXAxIjogewogICAgIm5hbWUiOiAidCA8IDgwMCBtcyIsCiAgICAiY291bnQiOiA
xMiwKICAgICJwZXJjZW50YWdlIjogMTAwCn0sCiAgICAiZ3JvdXAyIjogewogICAgIm5hbWUiOi
AiODAwIG1zIDwgdCA8IDEyMDAgbXMiLAogICAgImNvdW50IjogMCwKICAgICJwZXJjZW50YWdlI
jogMAp9LAogICAgImdyb3VwMyI6IHsKICAgICJuYW1lIjogInQgPiAxMjAwIG1zIiwKICAgICJj
b3VudCI6IDAsCiAgICAicGVyY2VudGFnZSI6IDAKfSwKICAgICJncm91cDQiOiB7CiAgICAibmF
tZSI6ICJmYWlsZWQiLAogICAgImNvdW50IjogMCwKICAgICJwZXJjZW50YWdlIjogMAp9LAogIC
AgIm1lYW5OdW1iZXJPZlJlcXVlc3RzUGVyU2Vjb25kIjogewogICAgICAgICJ0b3RhbCI6IDEuN
SwKICAgICAgICAib2siOiAxLjUsCiAgICAgICAgImtvIjogMAogICAgfQp9CiAgICB9LCJyZXFf
YWNjZXNzLXB1cmNoYXNlLWMxNGJmIjogewogICAgICAgICJ0eXBlIjogIlFUVVFU1QiLAogICA
gICAgICJuYW1lIjogIkFjY2VzcyBQdXJjaGFzZSBSZWRpcmVjdCAxIiwKInBhdGgiOiAiQWNjZX
NzIFB1cmNoYXNlIFJlZGlyZWN0IDEiLAoicGF0aEZvcm1hdHRlZCI6ICJyZXFfYWNjZXNzLXB1c
mNoYXNlLWMxNGJmIiwKInN0YXRzIjogewogICAgIm5hbWUiOiAiQWNjZXNzIFB1cmNoYXNlIFJl
ZGlyZWN0IDEiLAogICAgIm51bWJlck9mUmVxdWVzdHMiOiB7CiAgICAgICAgInRvdGFsIjogMSw
KICAgICAgICAib2siOiAxLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAibWluUmVzcG9uc2
VUaW1lIjogewogICAgICAgICJ0b3RhbCI6IDQ4MCwKICAgICAgICAib2siOiA0ODAsCiAgICAgI
CAgImtvIjogMAogICAgfSwKICAgICJtYXhSZXNwb25zZVRpbWUiOiB7CiAgICAgICAgInRvdGFs
IjogNDgwLAogICAgICAgICJvayI6IDQ4MCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgIm1
lYW5SZXNwb25zZVRpbWUiOiB7CiAgICAgICAgInRvdGFsIjogNDgwLAogICAgICAgICJvayI6ID
Q4MCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInN0YW5kYXJkRGV2aWF0aW9uIjogewogI
CAgICAgICJ0b3RhbCI6IDAsCiAgICAgICAgIm9rIjogMCwKICAgICAgICAia28iOiAwCiAgICB9
LAogICAgInBlcmNlbnRpbGVzMSI6IHsKICAgICAgICAidG90YWwiOiA0ODAsCiAgICAgICAgIm9
rIjogNDgwLAogICAgICAgICJrbyI6IDAKICAgIH0sCiAgICAicGVyY2VudGlsZXMyIjogewogIC
AgICAgICJ0b3RhbCI6IDQ4MCwKICAgICAgICAib2siOiA0ODAsCiAgICAgICAgImtvIjogMAogI
CAgfSwKICAgICJwZXJjZW50aWxlczMiOiB7CiAgICAgICAgInRvdGFsIjogNDgwLAogICAgICAg
ICJvayI6IDQ4MCwKICAgICAgICAia28iOiAwCiAgICB9LAogICAgInBlcmNlbnRpbGVzNCI6IHs
KICAgICAgICAidG90YWwiOiA0ODAsCiAgICAgICAgIm9rIjogNDgwLAogICAgICAgICJrbyI6ID
AKICAgIH0sCiAgICAiZ3JvdXAxIjogewogICAgIm5hbWUiOiAidCA8IDgwMCBtcyIsCiAgICAiY
291bnQiOiAxLAogICAgInBlcmNlbnRhZ2UiOiAxMDAKfSwKICAgICJncm91cDIiOiB7CiAgICAi
bmFtZSI6ICI4MDAgbXMgPCB0IDwgMTIwMCBtcyIsCiAgICAiY291bnQiOiAwLAogICAgInBlcmN
lbnRhZ2UiOiAwCn0sCiAgICAiZ3JvdXAzIjogewogICAgIm5hbWUiOiAidCA+IDEyMDAgbXMiLA
ogICAgImNvdW50IjogMCwKICAgICJwZXJjZW50YWdlIjogMAp9LAogICAgImdyb3VwNCI6IHsKI
CAgICJuYW1lIjogImZhaWxlZCIsCiAgICAiY291bnQiOiAwLAogICAgInBlcmNlbnRhZ2UiOiAw
Cn0sCiAgICAibWVhbk51bWJlck9mUmVxdWVzdHNQZXJTZWNvbmQiOiB7CiAgICAgICAgInRvdGF
sIjogMC4xMjUsCiAgICAgICAgIm9rIjogMC4xMjUsCiAgICAgICAgImtvIjogMAogICAgfQp9Ci
AgICB9LCJuYW1lIjogInN0YXRzLmpzb24ifV19]}
```


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 Xray Json reports 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) and add authentication parameters to the subsequent requests.

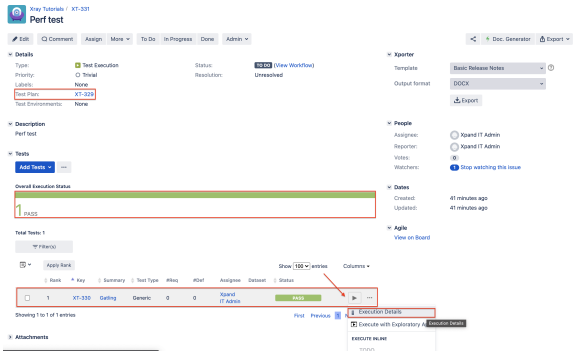
Xray Json results

Using the API we will perform a request like the following:

```
curl -H "Content-Type: application/json" -X POST -u admin:admin --data
'@xrayJson.json' 'https://yourserver/rest/raven/1.0/import/execution'
```

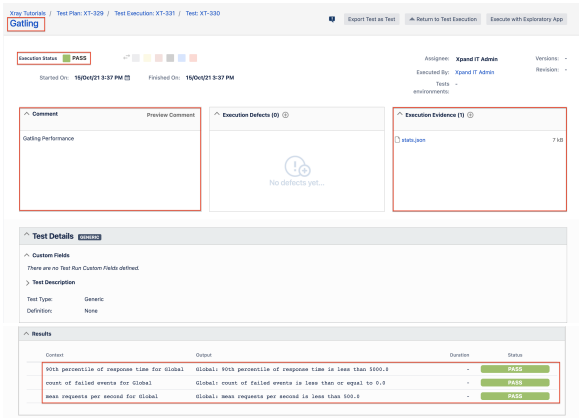
With this command we are creating a new Test Execution that will have the results of the Tests that were executed.

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 we click on the details icon next to each Test (below the red arrow in the screenshot), this will take us 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, prepod, prod) or a 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/