Test Your Website Performance NOW! Enter Your URL to Get Star|

http://www.blazedemo.com

Start The Test

Alejandro Berardinelli

Alejandro Berardinelli is a Systems Engineer, with more than 3 years of experience in Performance Testing at Abstracta. He has solid experience in all roles involved in the performance testing cycle, from scripting to automating, executing and gathering results, to analyzing and obtaining conclusions and improvement recommendations. Alejandro also worked for 10 years at IBM as a Storage specialist, coauthoring several IBM

Redbooks.

JENKINS . May. 10th, 2018

How to Use the Jenkins Declarative Pipeline

Jenkins provides you with two ways of developing your pipeline code: Scripted and Declarative. Scripted pipelines, also known as “traditional” pipelines, are based on Groovy as their Domain-specific language. On the other hand, Declarative pipelines provide a simplified and more friendly syntax with specific statements for defining them, without needing to learn Groovy.

Jenkins pipeline plugin version 2.5 introduces support for Declarative pipelines. More information on how to write Scripted pipelines can be found at my previous blog post ”How to Use the Jenkins Scripted Pipeline”.

In this blog post we will cover all the directives available to develop your Declarative pipeline script, which will provide a clear picture on its functionality.

Declarative Pipelines Syntax

A valid Declarative pipeline must be defined with the “pipeline” sentence and include the next required sections:

- agent
- stages
- stage
- steps

Also, these are the available directives:

- environment (Defined at stage or pipeline level)

- input (Defined at stage level)
- options (Defined at stage or pipeline level)
- parallel
- parameters
- post
- script
- tools
- triggers
- when

We will now describe each of the listed directives/sections, starting with the required ones.

Agent

Jenkins provides the ability to perform distributed builds by delegating them to “agent” nodes. Doing this allows you to execute several projects with only one instance of the Jenkins server, while the workload is distributed to its agents. Details on how to configure a master/agent mode are out of the scope of this blog. Please refer to Jenkins Distributed builds for more information.

Agents should be labeled so they can be easily identified from each other. For example, nodes can be labeled by their platform (Linux, Windows, etc), by their versions or by their location, among others.
The “agent” section configures on which nodes the pipeline can be run. Specifying “agent any” means that Jenkins will run the job on any of the available nodes.

An example of its usage could be:

```
pipeline {
  agent any
  ...
}
```

Stages

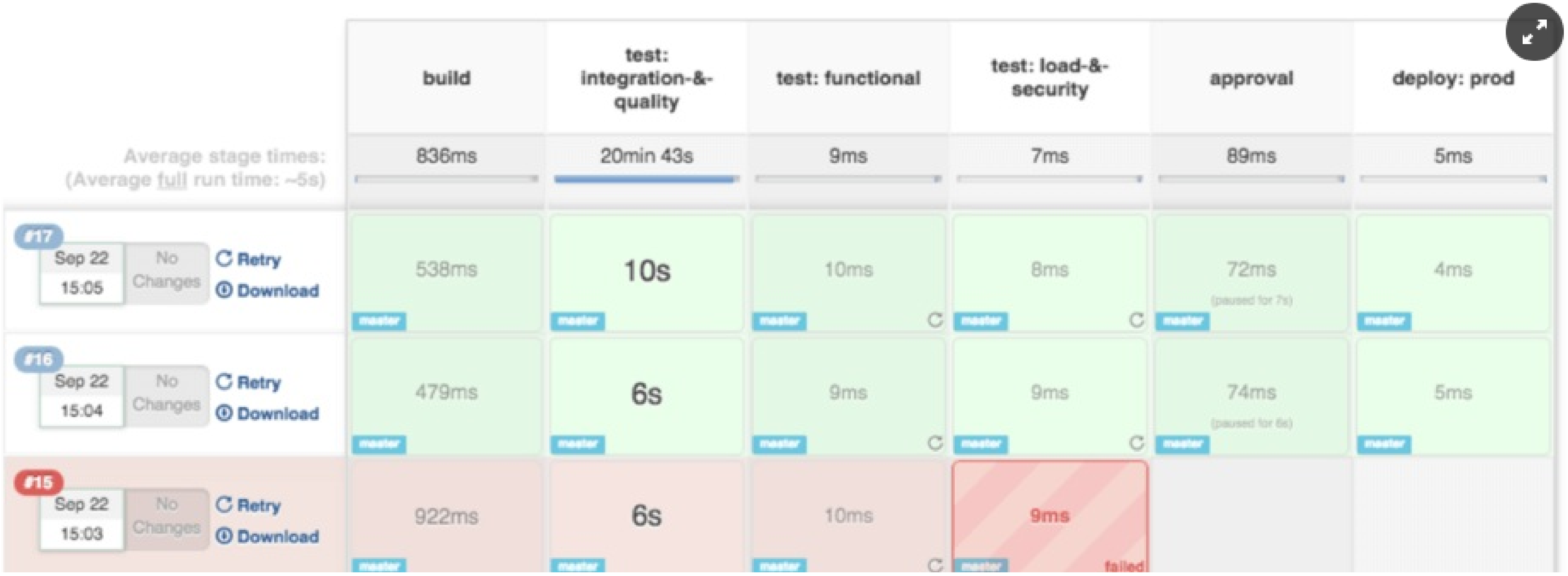
This section allows to generate different stages on your pipeline that will be visualized as different segments when the job is run.

A sample pipeline including the stages sentence is provided:

```
pipeline {
  agent any
  stages {
    ...
  }
}
```

Stage

At least one “stage” section must be defined on the “stages” section. It will contain the work that the pipeline will execute. Stages must be named accordingly since Jenkins will display each of them on its interface, as shown here:



	build	test: integration-&-quality	test: functional	test: load-&-security	approval	deploy: prod
Average stage times: (Average full run time: ~5s)	836ms	20min 43s	9ms	7ms	89ms	5ms
#17 Sep 22 15:05 No Changes Retry Download	538ms master	10s master	10ms master	8ms master	72ms (paused for 7s) master	4ms master
#16 Sep 22 15:04 No Changes Retry Download	479ms master	6s master	9ms master	9ms master	74ms (paused for 6s) master	5ms master
#15 Sep 22 15:03 No Changes Retry Download	922ms master	6s master	10ms master	9ms master failed		

Jenkins graphically splits pipeline execution based on the defined stages and displays their duration and whether it was successful or not.

The pipeline script for the previous image looks like the following:

```
pipeline {
 agent any
 stages {
  stage ('build') {
   ...
  }
  stage ('test: integration-&-quality') {
   ...
  }
  stage ('test: functional') {
   ...
  }
  stage ('test: load-&-security') {
   ...
  }
  stage ('approval') {
   ...
  }
  stage ('deploy:prod') {
   ...
  }
 }
}
```

Steps

The last required section is “steps”, which is defined into a “stage”. At least one step must be defined in the “steps” section.

For Linux and MacOS, sh is supported. Here is an example:

```
steps {
sh 'echo "A one line step"'
sh '''
echo "A multiline step"'
cd /tests/results
ls -lrt
'''
}
```

For Windows, bat or powershell can be used, as shown:

```
steps {
bat "mvn clean test -Dsuite=SMOKE_TEST -Denvironment=QA"
powershell ".\funcional_tests.ps1"
}
```

The other non required directives will be explained in the following paragraphs.

Environment

This directive can be both defined at stage or pipeline level, which will determine the scope of its definitions. When "environment" is used at the "pipeline" level, its definitions will be valid for all of the pipeline steps. If instead it is defined within a "stage", it will only be valid for the particular stage.

Sample uses of this directive:

At the “pipeline” level:

```
pipeline {
 agent any
 environment {
  OUTPUT_PATH = './outputs/'
 }
 stages {
  stage ('build') {
    ...
  }
 ...
 }
}
```

Here, “environment” is used at a “stage” level:

```
pipeline {
  agent any
  stages {
    stage ('build') {
  environment {
    OUTPUT_PATH = './outputs/'
  }
  ...
    }
    ...
  }
}
```

Input

The "input" directive is defined at a stage level and provides the functionality to prompt for an input. The stage will be paused until a user manually confirms it.

The following configuration options can be used for this directive:

- message: This is a required option where the message to be displayed to the user is specified.
- id: Optional identifier for the input. By default the "stage" name is used.
- ok: Optional text for the Ok button.
- submitter: Optional list of users or external group names who are allowed to submit the input. By default any user is allowed.
- submitterParameter: Optional name of an environment variable to set with the submitter name, if present.
- parameters: Optional list of parameters to be provided by the submitter.

Here is a sample pipeline containing this directive:

```
pipeline {
  agent any
  stages {
    stage ('build') {
  input{
    message "Press Ok to continue"
    submitter "user1,user2"
    parameters {
      string(name:'username', defaultValue: 'user', description: 'Username of the user pressing Ok')
    }
  }
  steps {
    echo "User: ${username} said Ok."
  }
    }
  }
}
```

Options

Defined at pipeline level, this directive will group the specific options for the whole pipeline. The available options are:

- buildDiscarder
- disableConcurrentBuilds
- overrideIndexTriggers
- skipDefaultCheckout
- skipStagesAfterUnstable
- checkoutToSubdirectory
- newContainerPerStage

- timeout
- retry
- timestamps

Please refer to Jenkins Declarative pipeline options for a full reference on this.

For example, you can configure your pipeline to be retried 3 times before failing by writing:

```
pipeline {
  agent any
  options {
    retry(3)
  }
  stages {
    ...
  }
}
```

Parallel

Jenkins pipeline Stages can have other stages nested inside that will be executed in parallel. This is done by adding the “parallel” directive to your script. An example on how to use it is provided:

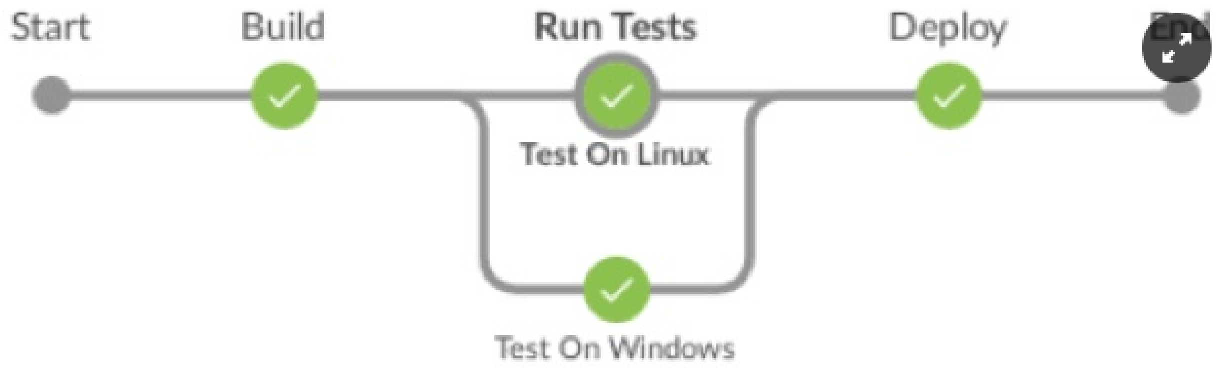
```
stage('run-parallel-branches') {
steps {
parallel(
a: {
echo "Tests on Linux"
},
b: {
echo "Tests on Windows"
}
)
}
}
```

Starting with Declarative Pipeline version 1.2, a new syntax was introduced, making the use of the parallel syntax much more declarative-like.

The previous script rewritten with this new syntax will look like:

```
pipeline {
agent none
stages {
stage('Run Tests') {
parallel {
stage('Test On Windows') {
agent { label "windows" }
steps {
bat "run-tests.bat"
}
}
stage('Test On Linux') {
agent { label "linux" }
steps {
sh "run-tests.sh"
}
}
}
}
}
}
```

Any of the previous pipelines will look like this:



Both scripts will run the tests on different nodes since they run specific platform tests. Parallelism can also be used to simultaneously run stages on the same node by the use of multithreading, if your Jenkins server has enough CPU.

Some restrictions apply when using parallel stages:

- A stage directive can have either a parallel or steps directive but not both.
- A stage directive inside a parallel one cannot nest another parallel directive, only steps are allowed.
- Stage directives that have a parallel directive inside cannot have "agent" or "tools" directives defined.

Parameters

This directive allows you to define a list of parameters to be used in the script. Parameters should be provided once the pipeline is triggered. It should be defined at a "pipeline" level and only one directive is allowed for the whole pipeline.

String and boolean are the valid parameter types that can be used.

```
pipeline {
agent any
parameters {
string(name: 'user', defaultValue: 'John', description: 'A user that triggers the pipeline')
}
stages {
stage('Trigger pipeline') {
steps {
echo "Pipeline triggered by ${params.USER}"
}
}
}
}
```

Post

Post sections can be added at a pipeline level or on each stage block and sentences included in it are executed once the stage or pipeline completes. Several post-conditions can be used to control whether the post executes or not:

- always: Steps are executed regardless of the completion status.
- changed: Executes only if the completion results in a different status than the previous run.
- fixed: Executes only if the completion is successful and the previous run failed
- regression: Executes only if current execution fails, aborts or is unstable and the previous run was successful.
- aborted: Steps are executed only if the pipeline or stage is aborted.
- failure: Steps are executed only if the pipeline or stage fails.
- success: Steps are executed only if the pipeline or stage succeeds.
- unstable: Steps are executed only if the pipeline or stage is unstable.

Since sentences included in a pipeline post block will be run at the end of the script, cleanup tasks or notifications, among others, can be performed here.

```
pipeline {
agent any
stages {
stage('Some steps') {
steps {
...
}
}
}
post {
always {
  echo "Pipeline finished"
  bat ./performCleanUp.bat
}
}
}
```

Script

This step is used to add Scripted Pipeline sentences into a Declarative one, thus providing even more functionality. This step must be included at "stage" level.

Several times blocks of scripts can be utilized on different projects. These blocks allow you to extend Jenkins functionalities and can be implemented as shared libraries. More information on this can be found at Jenkins shared libraries. Also, shared libraries can be imported and used into the "script" block, thus extending pipeline functionalities.

Next we will provide sample pipelines. The first one will only have a block with a piece of Scripted pipeline text, while the second one will show how to import and use shared libraries:

```
pipeline {
agent any
stages {
stage('Sample') {
steps {
echo "Scripted block"
script {

}
}
}
}
}
```

Please refer to our post about Scripted pipelines at How to Use the Jenkins Scripted Pipeline for more information on this topic.

Tools

The "tools" directive can be added either at a pipeline level or at the stage one. It allows you to specify which maven, jdk or gradle version to use on your script. Any of these tools, the three supported at the time of writing, must be configured on the "Global tool configuration" Jenkins menu.

Also, Jenkins will attempt to install the listed tool (if it is not installed yet). By using this directive you can make sure a specific version required for your project is installed.

```
pipeline {
agent any
tools {
  maven 'apache-maven-3.0.1'
}
stages {
...
}
}
```

Triggers

Triggers allows Jenkins to automatically trigger pipelines by using any of the available ones:

- cron: By using cron syntax, it allows to define when the pipeline will be re-triggered.
- pollSCM: By using cron syntax, it allows you to define when Jenkins will check for new source repository updates. The Pipeline will be re-triggered if changes are detected. (Available starting with Jenkins 2.22).
- upstream: Takes as input a list of Jenkins jobs and a threshold. The pipeline will be triggered when any of the jobs on the list finish with the threshold condition.

Sample pipelines with the available triggers are shown next:

```
pipeline {
agent any
triggers {
  //Execute weekdays every four hours starting at minute 0
cron('0 */4 * * 1-5')
}
stages {
```

```
...
}
}



pipeline {
agent any
triggers {
  //Query repository weekdays every four hours starting at minute 0
pollSCM('0 */4 * * 1-5')
}
stages {
...
}
}


pipeline {
agent any
triggers {
  //Execute when either job1 or job2 are successful
upstream(upstreamProjects: 'job1, job2', threshold: hudson.model.Result.SUCCESS)
}
stages {
...
}
}
```

When

Pipeline steps could be executed depending on the conditions defined in a “when” directive. If conditions match, the steps defined in the corresponding stage will be run. It should be defined at a stage level.

For a full list of the conditions and its explanations refer to Jenkins declarative pipeline ”when” directive.

For example, pipelines allow you to perform tasks on projects with more than one branch. This is known as multibranched pipelines, where specific actions can be taken depending on the branch name like “master”, “feature*”, “development”, among others. Here is a sample pipeline that will run the steps for the master branch:

```
pipeline {
agent any
stages {
  stage ('Deploy stage') {
when {
branch 'master'
}
steps {
echo 'Deploy master to stage'
...
}
      }
}
}
```

2 Final Jenkins Declarative Pipeline Tips:

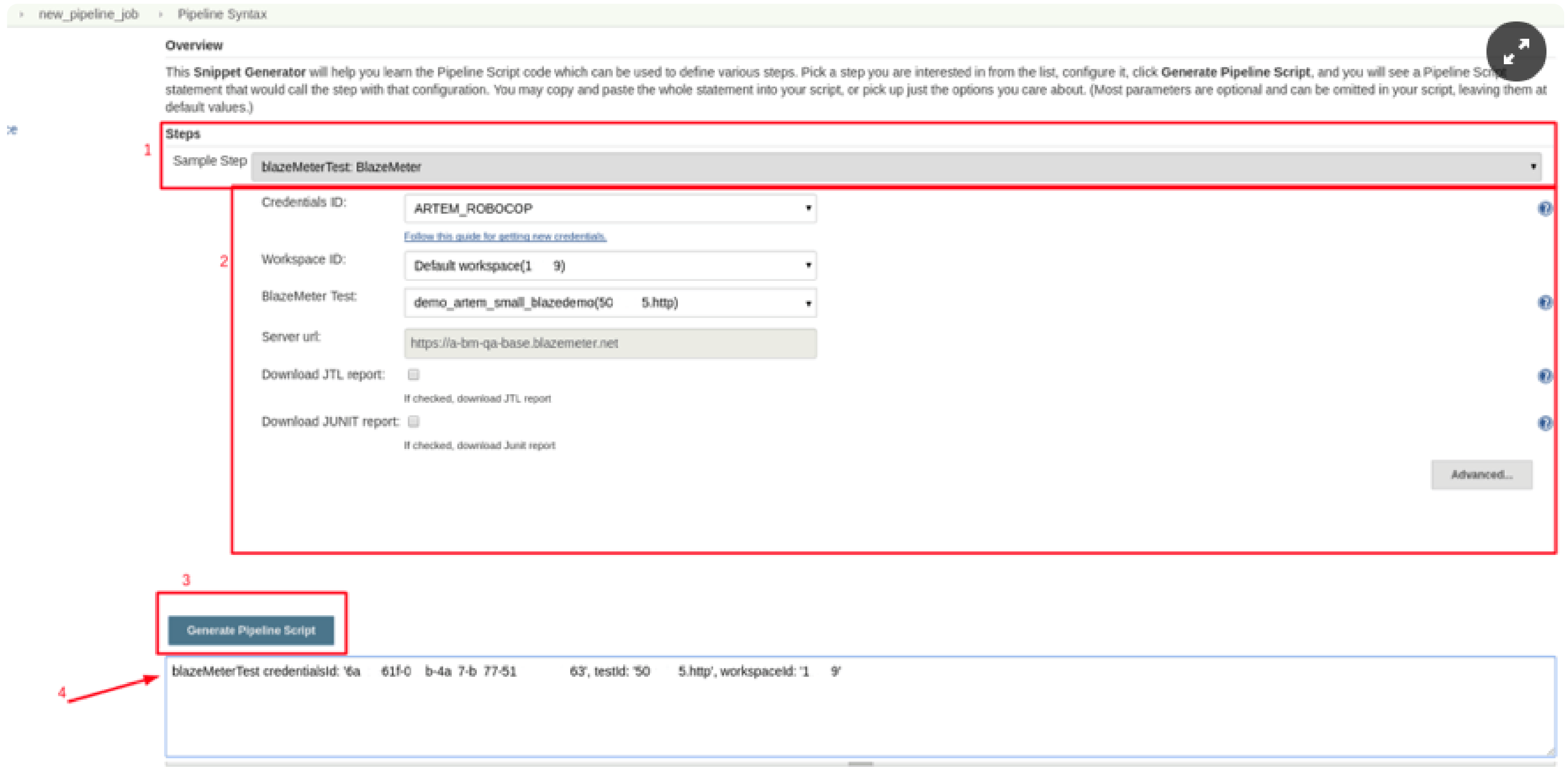
Declarative pipelines syntax errors are reported right at the beginning of the script. This is a nice functionality since you will not waste time until a step fails to realize there is a typo or misspelling.

As already mentioned, pipelines can be written either declarative or scripted. Indeed, the declarative way is built on top of the scripted way making it easy to extend as explained, by adding script steps.

Jenkins pipelines are being widely used on CI/CD environments. Using either declarative or scripted pipelines has several advantages. In this post we presented all the syntax elements to write your declarative script along with samples. As we already stated, the declarative way offers a much more friendly syntax with no Groovy knowledge required.

Running BlazeMeter in Your Jenkins Pipeline

Your performance tests should be part of your Jenkins Pipeline, to ensure that any code change doesn't degrade performance. Add BlazeMeter to Jenkins with the BlazeMeter Jenkins Plugin, run your tests and analyze with BlazeMeter's insightful reports.



› new_pipeline_job › Pipeline Syntax

Overview

This **Snippet Generator** will help you learn the Pipeline Script code which can be used to define various steps. Pick a step you are interested in from the list, configure it, click **Generate Pipeline Script**, and you will see a Pipeline Script statement that would call the step with that configuration. You may copy and paste the whole statement into your script, or pick up just the options you care about. (Most parameters are optional and can be omitted in your script, leaving them at default values.)

Steps

Sample Step	blazeMeterTest: BlazeMeter
Credentials ID:	ARTEM_ROBOCOP
	Follow this guide for getting new credentials.
Workspace ID:	Default workspace(1 9)
BlazeMeter Test:	demo_artem_small_blazedemo(50 5.http)
Server url:	https://a-bm-qa-base.blazemeter.net
Download JTL report:	☐ If checked, download JTL report
Download JUNIT report:	☐ If checked, download Junit report

Advanced...

Generate Pipeline Script

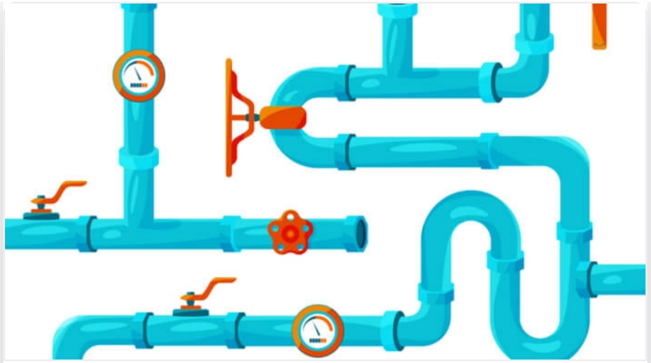
```
blazeMeterTest credentialsId: '6a    61f-0    b-4a  7-b  77-51          63', testId: '50      5.http', workspaceId: '1     9'
```

Try out BlazeMeter's abilities by putting your URL in the box below, and your test will start in minutes.

Enter a URL

Start The Test

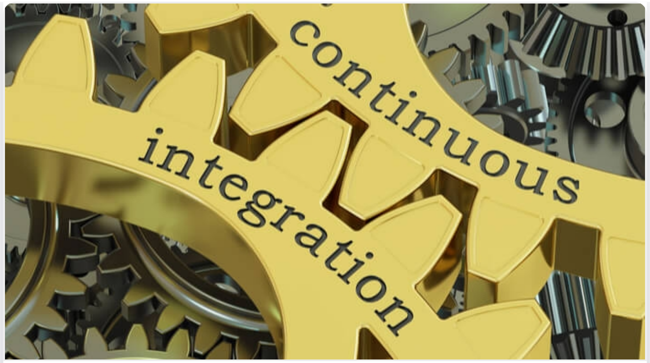
You might also find these useful:



How to Use the Jenkins Scripted Pipeline

Mar. 22nd, 2018

Alejandro Berardinelli



What's New in the BlazeMeter Jenkins Plugin 4.0

Apr. 11th, 2018

Artem Fedorov

Tags:

Jenkins Continuous Integration continuous testing open source Groovy language jenkins pipelines Declarative pipelines scripted pipelines Jenkins CI

Interested in writing for our Blog? Send us a pitch!

Follow Us

Subscribe to our blog

Use Cases Try Resources Company

JMeter™

Jenkins CI

Open Source Tools

API Functional Testing

LoadRunner Vs. BlazeMeter

Black Friday

Case Studies

Selenium Functional Testing

Shift Left Testing

Mainframe Testing

Run a Test

Try

Interactive Features

Pricing

Professional Services

Blog

Webinars

Knowledge Base

Changelog

JMeter Academy

BlazeMeter Labs

Continuous Testing

About

Become a Partner

Careers

News and Media

Need Help?

BlazeMeter Intro

Support

API Documentation

Status

JMeter Slack

CT Slack