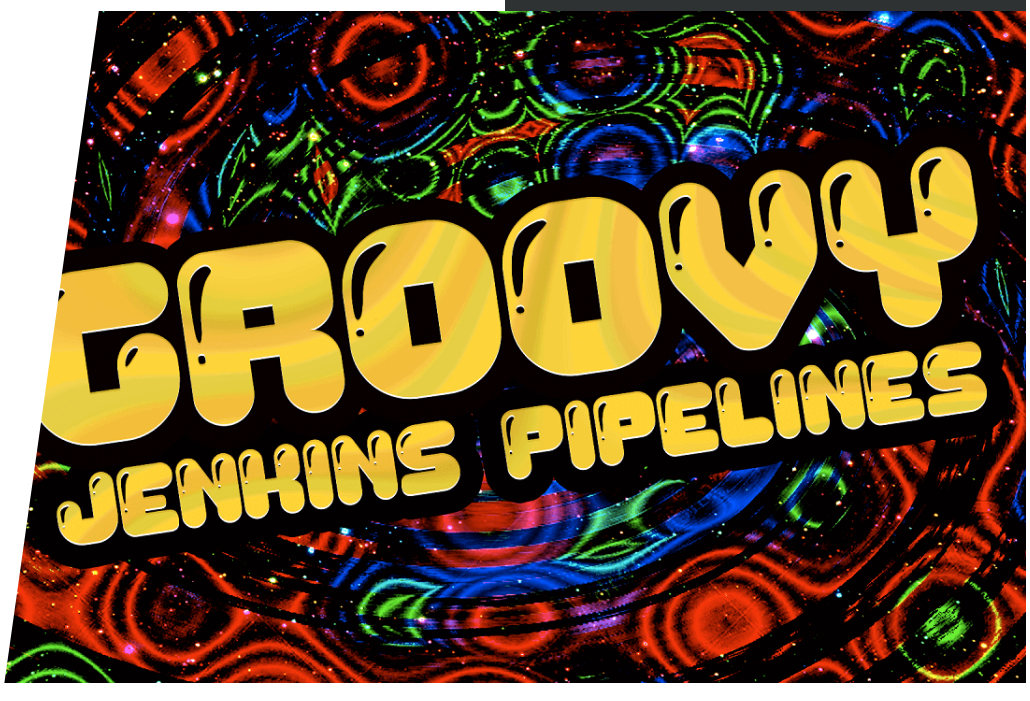
eficode

- Services
- Solutions

Tutorial: Jenkins Pipeline file with Apache Groovy



July 11, 2019 DEVOPS

This Groovy tutorial for Jenkins will show you how to use Apache Groovy script to build a Jenkins pipeline. Groovy is suitable for beginners and is a good choice for uniting teams' scripts.

What is an Apache Groovy Script?

But what is Groovy? Well, Apache Groovy is an object-oriented programming language used for JVM platform. This dynamic language has a lot of features drawing inspiration from Python, Smalltalk & Ruby. It can be used to orchestrate your pipeline in Jenkins and it can glue different languages together meaning that teams in your project can be contributing in different languages.

Groovy can seamlessly interface with the Java language and the syntax of Java and Groovy is very similar. If you forget the syntax when writing Groovy, you can continue to write directly in Java syntax. Groovy can also be used as one of the scripting languages for the Java platform. Groovy scripts can be called in Java, which effectively reduces time spent on Java development.

It also offers many productivity features like DSL support, closures, and dynamic typing. Unlike some other languages, it functions as a companion, not a replacement, for Java. Groovy source code gets compiled to JVM Bytecode so it can run on any platform.

Why use Groovy?

Here are the major reasons why you should consider using Groovy, a commonplace option for creating pipeline files.

1. As we said earlier, Groovy is an agile and dynamic language. It has seamless integration with all existing Java objects and libraries.

Groovy is Java without types, so without defining the basic data types like int, float, String, etc. Basically it’s the same as Java: defining variable without specifying a type. Groovy will judge the type based on the value of the object.

```
def str = "Hello world"
def num =0
```

2. In Groovy, the 'for' loop becomes more concise and easier to read. Groovy loop statements support: while, for, for-in, break, continue, and the whole is consistent with Java.

For example:

0..5 indicates that the integers 0,1,2,3,4,5 are included

0..<5 means 0,1,2,3,5 and a..d means a,b,c,d:

```
for(x in 1..5){
  println x //0,1,2,3,4,5
}
```

Groovy also supports default parameter values:

```
def repeat(val, x=10){
  for(i in 0..<x){
    println val
  }
}
```

3. In Groovy, we can use scopes to define Collections or Arrays.

```
def arg = ["Groovy","Java","Python",”nodeJS”]
println arg.class
```

The search method in Groovy is also more flexible

```
println arg[1]
```

Groovy also allows you to add or remove collections from collections.

```
def no = [1,2,3,4]
def no2 = no +5 //=[1,2,3,4,5]
def no3 = no - [2,3] //=[1,4]
```

When adding elements to a collection, we can use the following methods

```
arg.add("Ruby")
arg << "Smalltalk"
arg[5] = "C++"
```

4. A further benefit to Groovy is its Operators:

Groovy arithmetic operators, logical operators, relational operators and bitwise operators are all consistent with languages like nodeJS. Groovy's == is equivalent to the equals methods in Java .

```
String str1 = "123";
String str2 = new String("123");
if(str1 == str2){
  println(“equal");
}else{
  println("Not equal");
}
```

5. List methods available in Groovy are another benefit.

Add( ) Append the new value to the end of this list.

Get( ) Returns the element at the specified position in this list.

Contains( ) Returns true if this list contains the specified value.

Minus( ) Create a new list of original elements that removes the specified element

Plus ( ) Create a new list of the original list elements and the specified elements.

Pop( ) Remove the last item from this list

Remove( ) Remove elements from the specified position in the list

Reverse() Create a new list that is the opposite of the original list's elements

Size( ) Get the number of elements in this list.

Sort( ) Returns a sorted copy of the original list.

```
def list = [];
list = [1, 2, 3, 4];
list.add(5); //add
list.pop(); //pop
```

6. In any other object-oriented language, there are concepts of objects and classes to represent the object-oriented nature of a programming language. Groovy implicitly creates getters, setter methods, and provides constructors with arguments.

```
class Person {
  String name;
  int ID;
}
class Test {
  static void main(String) {
    def emp = new emp(name: 'name’')
    println emp.getName();
    emp.setYR(2019);
    println emp.getYR(); // 2019
  }
}
```

7. Exception handling is the same as Java, using try, catch to catch exceptions in Groovy

```
Try{
.....
}catch(Exception1 exp){
.....
}finally{
......
}
```

There are lots of Advanced topics (JSON operation, Methods, Maps etc etc) in Groovy which can be referred to in official Apache Documentation

Groovy also combines the features of Python, Ruby and other scripting languages. It does a lot of syntactic sugar in grammar. When it comes to Java development, there is a lot of code that must be written in Java which can be omitted in Groovy as seen from the previous snippets.

To conclude: the key benefit of Groovy is that you can write the same function with less code, which in my mind means more concise and meaningful code compared to Java.

What are the disadvantages of using Groovy?

Some would say that a simpler infrastructure as code with declarative files, which using Groovy may not represent, is the better option to go for in the long term when pursuing DevOps. What's more, Apache Groovy can seem like a hacky language at first.

However I've seen in practice that setting Groovy up has brought teams together in organizations that haven't pursued DevOps explicitly before. It's allowed them to work independently while increasing the amount of shared libraries.

For more detailed learning in Groovy look no further than their documentation.

Jenkins Pipeline in Jenkins File

Let's talk about the Jenkins Pipeline approach in Jenkins and proper pipeline syntax. With Jenkins embedded Groovy support, plus a rich pipeline Step library reference, you can implement a complex build and release process by writing custom pipeline scripts.

Jenkins Pipeline can be created in the following ways:

- Through the classic UI/Blue Ocean - you can directly enter the basic Pipeline into Jenkins using the classic UI (or) using Blue Ocean UI -you can use the Blue Ocean UI to write Pipeline's Jenkinsfile and submit it to source control [here's more information regarding Blue Ocean]
- In SCM - Jenkinsfile can be written by hand and submitted to the project's source control repository.

Instead of relying on creating freestyle jobs and configuring it, Jenkins pipeline has a set of instructions for the Jenkins job to execute.

Jenkinsfile created by the classic UI is saved directly by Jenkins. Pipeline created from the Jenkins classic UI is saved in the root directory of Jenkins and script is executed in Jenkins script console. The Jenkins file is a base code for Jenkins which executes it as a Groovy script

in Jenkins script console.

Anatomy of Jenkins File

The first line *shebang* defines the file as a Groovy language script:

```
#!/usr/bin/env groovy
```

In-line Pipeline files do not have a *shebang* because it is supplied internally.

```
// Jenkinsfile (Declarative Pipeline)
pipeline {
  agent any
  stages {
    stage('Stage 1') {
      steps {
        echo 'Hello world!'
      }
    }
  }
}
```

The same above written (declarative pipeline) example can be written in scripted pipeline code as well.

A scripted pipeline is a groovy-based DSL. It provides greater flexibility and scalability for Jenkins users than the Declarative pipeline.

Groovy scripts are not necessarily suitable for all users, so Jenkins created the Declarative pipeline. The Declarative Pipeline syntax is more stringent. It needs to use Jenkins' predefined DSL structure, which provides a simpler and more assertive syntax for writing Jenkins pipelines.

```
// Jenkinsfile (Scripted Pipeline)
node { // node/agent
  stage('Stage 1') {
    echo 'Hello World' // echo Hello World
  }
}
```

Node/agent

This defines where to execute the code on which machine.

```
pipeline {
  node any
}
```

or

```
pipeline {
  agent { node { label 'labelName' } }
}
```

The section must be defined at the top-level inside the pipeline block, but stage-level usage is optional.

For context: Jenkins has a concept of Master and worker nodes ie. slaves.

Why do we need multiple nodes

1. To distribute the job load on Jenkins

2. We have multiple nodes for different kinds of projects

For example, let's assume we have two different kinds of projects:

- Java
- iOS

For building Java job we need to use a *Java Build Machine* / Node.

Similarly for an *iOS build* we need a *MAC Slave*/Machine/Windows

Stage

We have multiple stages in our build in which each stage section has a different steps and commands to follow. When most of the Jenkins Pipeline's work is performed it contains a sequence of one or more stages. It is recommended that the stages contain at least one stage directive for connecting various delivery processes, such as build, test, and deploy.

```
stages {
  stage('Build') {
    steps {
      sh 'echo "This is my first step"'
    }
  }
  stage('Test') {
    steps
      sh 'echo "This is my Test step"'
    }
  }
  stage('Deploy') {
    steps {
      sh 'echo "This is my Deploy step"'
    }
  }
}
```

Parallel Stages

Stages in pipeline may declare a number of nested stages within a parallel block, which can be used to improve operational efficiency for stages that are time consuming and have no dependencies on each other. In addition, multiple steps in a single parallel can also be executed in parallel.

```
stage("Parallel") {
  steps {
```

```
      parallel (
        "Taskone" : {
          //do some stuff
        },
        "Tasktwo" : {
          // Do some other stuff in parallel
        }
      )
    }
  }
```

Steps

The core and basic part of the pipeline is *step*. Pipelines are made up of multiple steps that allow you to build, test, and deploy applications. Fundamentally, steps are the most basic building blocks of the declarative pipeline and the scripted pipeline syntax to tell Jenkins what to do.

The scripted pipeline does not specifically describe steps as part of its syntax, but in the pipeline steps reference document, the steps involved in the pipeline and its plugins are described in detail.

Jenkins pipeline allows you to compose multiple steps that helps to create any sort of automation process. Think of a *step* like a single command which performs a single action. When a step succeeds, it moves onto the next step. When a step fails to execute correctly the pipeline will fail.

When all the steps in the pipeline have successfully completed, the pipeline is considered to have successfully executed

```
stages {
  stage('Build') {
    steps {
      sh 'echo "Step 1"'
      sh 'echo "Step 2"'
      sh 'echo "Step 3"'
    }
  }
}
```

Timeout and retries

```
pipeline {
  agent any
  stages {
    stage('Dev Deployment') {
      steps {
        retry(x) { // It Retries x number of times mentioned until its
successful
          sh './dev-deploy.sh'
        }
        timeout(time: x, unit: 'MINUTES') { // Time out option will make
```

```
    the step wait for x mins to execute if it takes more than that it will
    fail
              sh './readiness-check.sh'
            }
          }
        }
      }
    }
```

Triggers

The triggers directive defines the automated way how Pipeline automates triggers. According to the Jenkins documentation, the triggers currently available are cron, pollSCM and upstream.

Cron

Accepts a cron-style string to define the regular interval that the pipeline triggers, for example:

```
triggers { cron('H */2 * * 1-3') }
```

pollSCM

Accepts a cron-style string to define Jenkins to check SCM source changes Regular interval. If there are new changes, the pipeline will be retriggered.

```
pipeline {
  agent any
  triggers {
    cron('H */2 * * 1-3')
  }
  stages {
    stage('Hello World') {
      steps {
        echo 'Hello World'
      }
    }
  }
}
```

When

The *when* command allows the pipeline to determine whether to perform this phase based on the given conditions. The *when* command must contain at least one condition. More complex pipeline conditional structures can be built using nested conditions: **not**, **allOf** or **anyOf**. If the *when* directive contains multiple conditions, then all sub-conditions must return true for stage execution. Nested conditions can be nested to any depth.

Mostly it's been used to skip or execute a step/stage

```
when { anyOf { branch 'develop'; branch 'test' } }
```

```
    stage('Sample Deploy') {
      when {
        branch 'production'
        anyOf {
          environment name: 'DEPLOY_TO', value: 'prod'
          environment name: 'DEPLOY_TO', value: 'test'
        }
      }
      steps {
        echo 'Deploying application'
      }
    }
```

Post-build sections

Define the operation at the end of the pipeline or stage run. You may need to run clean-up steps or perform some actions based on the outcome of the pipeline. The post-condition block supports the following components: always, changed, failure, success, unstable, and aborted. These blocks allow the steps to be performed at the end of the pipeline or stage run, depending on the status of the pipeline.

```
pipeline {
  agent any
  stages {
    stage('Test') {
      steps {
        sh 'echo "Fail!"; exit 1'
      }
    }
  }
  post {
    always {
      echo 'always runs regardless of the completion status of the
Pipeline run'
    }
    success {
      echo 'step will run only if the build is successful'
    }
    failure {
      echo 'only when the Pipeline is currently in a "failed" state run,
usually expressed in the Web UI with the red indicator.'
    }
    unstable {
      echo 'current Pipeline has "unstable" state, usually by a failed
test, code violations and other causes, in order to run. Usually
represented in a web UI with a yellow indication.'
    }
    changed {
      echo 'can only be run if the current Pipeline is running at a
different state than the previously completed Pipeline'
    }
  }
}
```

Environment variables

Environment variables can be set globally, like the example below, or per stage. As you might expect, setting environment variables per stage means they will only apply to the stage in which they're defined.

```
pipeline {
  agent any

  environment {
    USE_JDK = 'true'
    JDK ='c:/Java/jdk1.8'
  }

  stages {
    stage('Build') {
      steps {
        sh 'printenv'
      }
    }
  }
}
```

Exception handling in Jenkins Pipeline

For exception handling you can now use the try catch block in order to execute the step and catch exceptions.

```
node {
  stage('SampleTryCatch') {
    try {
      sh 'exit 1'
    }
    catch (exc) {
      echo 'Something didn't work and got some exceptions'
      throw
    }
  }
}
```

Recording tests and artifacts

Jenkins can record and aggregate test results so long as your test runner can output test result files. Jenkins typically comes bundled with the JUnit step, but if your test tools cannot output JUnit-style XML reports, then there are additional plugins which process practically any widely-used test report format.

To collect our test results and artifacts, we will use the post section.

```
pipeline {
  agent any
```

```
    stages {
      stage('Test') {
        steps {
          sh './gradlew test'
        }
      }
    }
    post {
      always {
        junit 'build/reports/**/*.xml'    }  } }
```

Sending notifications

When the pipeline succeeds or fails the current status notification needs to be sent to the concerned team. We can add some notification or other steps in the pipeline to perform finalization, notification, or other end-of-pipeline tasks.

Email notification

```
post {
  failure {
    mail to: 'team@test.com',
      subject: "Pipeline has failed: ${currentBuild.fullDisplayName}",
      body: "Error in ${env.BUILD_URL}"
  }
}
```

Slack notification

```
post {
  success {
    slackSend channel: '#Devops-team',
      color: 'green',
      message: "The pipeline ${currentBuild.fullDisplayName} completed
successfully."
}
```

Jenkins Pipeline Groovy: Shared libraries

As the pipeline is adopted for more and more projects in an organization, common patterns are likely to emerge. Oftentimes it is useful to share parts of pipelines between various projects to reduce redundancies.

Pipeline has support for creating *shared libraries* which can be defined in external source control repositories and loaded into existing pipelines

A shared library is defined with a name, a source code retrieval method such as by SCM, and optionally a default version. The name should be a short identifier as it will be used in scripts.

Consider the below following folder structure: (straight from the jenkins documentations)

```
(root)
+- src # Groovy source files
|
+- vars
|
+- test # unit test files
|
+- resources # resource files
|
```

The *src* directory should look like standard Java source directory structure. This directory is added to the classpath when executing pipelines.

The *vars* directory hosts script files that are exposed as a variable in pipelines. The name of the file is the name of the variable in the pipeline. For a more detailed walk-through of defining shared library in Jenkins. There are more options and other available snippets which we can try creating and running on the Jenkins system using Jenkins file.

The *resources* directory all the non-Groovy files/helper scripts required for the pipeline can be managed in this folder, from the above structure the sample. JSON template is stored in the resources folder and can be called in the shared library using libraryResource function.

You can use or call the library you've made by

```
@Library('my-shared-library') _
```

Sample of writing functions in the Groovy shared library

```
// src/org/foo/Zot.groovy
#!/usr/bin/groovy
package org.foo

def myFunction(repo) {
  sh "echo this is my function in Shared Library"
}

return this
```

Defining shared library global variables

The vars directory hosted scripts that define global variables accessible from the pipeline.

```
+- vars
| +- myVariables.groovy
```

Internally, scripts in the vars directory are instantiated on-demand as singletons. This allows multiple methods to be defined in a single .Groovy file for convenience.

Accessing steps

Shared library Groovy classes cannot directly call steps such as sh or git. Explicitly, we can pass a specific global variables env (contains all current environment variables) and steps (contains all standard pipeline steps) into a method of the class. They can however implement methods, outside of the scope of an enclosing class, which in turn invoke pipeline steps.

For example:

```
// src/org/mycode/mors.groovy
#!/usr/bin/groovy
package org.mycode

def checkOutFrom(repo) {
  git url: "git@github.com:jenkinsci/${repo}"
}
return this
```

Which can then be called from a scripted pipeline:

```
def z = new org.mycode.mors()
z.checkOutFrom(repo)
```

A complete Jenkins Groovy Script example Groovy file structure looks like

```
@Library(['github.com/shared-library']) _

pipeline {
  agent { label 'labelname' }

  options {
    timeout(time: 60, unit: 'MINUTES')
    timestamps()
    buildDiscarder(logRotator(numToKeepStr: '5'))
  }

  stages {

    stage('Clean Workspace') {
      steps {
        // Clean the workspace
      }
    }

    stage('Checkout') {
      steps {
        // clone your project from Git/SVN etc
      }
    }

    stage('Build') {
      steps {
        // build, build stages can be made in parallel aswell
        // build stage can call other stages
```

```
        // can trigger other jenkins pipelines and copy artifact from
that pipeline
      }
    }

    stage('Test') {
      steps {
        // Test (Unit test / Automation test(Selenium/Robot framework) /
etc.)
      }
    }

    stage('Code Analysis') {
      steps {
        // Static Code analysis (Coverity/ SonarQube /openvas/Nessus
etc.)
      }
    }

    stage('Generate Release Notes') {
      steps {
        // Release note generation .
      }
    }

    stage('Tagging') {
      steps {
        // Tagging specific version number
      }
    }

    stage('Release') {
      steps {
        // release specific versions(Snapshot / release / etc.)
      }
    }


  stage('Deploy') {
    steps {
      // Deploy to cloud providers /local drives /artifactory etc.
      // Deploy to Deploy/prod /test/ etc
    }
  }
}

post {
  success {
    echo "SUCCESS"
  }
  failure {
    echo "FAILURE"
  }
  changed {
```

```
        echo "Status Changed: [From: $currentBuild.previousBuild.result,
  To: $currentBuild.result]"
      }
      always {
        script {
          def result = currentBuild.result
          if (result == null) {
            result = "SUCCESS"
          }
        }
      }
    }
  }
```

For more snippets in Jenkins-Groovy please refer to Dennyzhang GitHub repo.

I hope this has helped you on your journey with Groovy!

Sandeep Kinayath

SEE MORE FROM AUTHOR →

Related