

Creating Multibranch Pipelines

While working on a multibranch project, multibranch pipelines will enable you to build different branches besides the default. As we are going to see in this section, these pipelines provide more flexibility by allowing you to perform different actions on different branches and to perform builds on merge requests.

Global Variables

A variable is a name associated to a value. A global variable is accessible in any scope within our program and is not bound to any scope, such as a function.

While working with the scripted Jenkins pipeline, we can configure it to work with different branches. There are pre-defined global variables available on Jenkins that allow us to use conditionals to perform different actions. For instance, there is a `BRANCH_NAME` global variable, which allows you to perform different actions, such as deploying while building the master branch, as opposed to while building a feature or bug branch. In this section, we'll learn how to set up global variables and create a multibranch pipeline in Jenkins.

1. To view the global variable reference, go to the pipeline project configuration page.
2. Go to the text area where you entered the pipeline script.
3. Click on the hyperlink labelled Pipeline Syntax.



4. This will land you on the Pipeline Syntax page.

Jenkins > lesson5-pipeline > Pipeline Syntax

[Back](#)

[Snippet Generator](#)

[Step Reference](#)

[Global Variables Reference](#)

[Online Documentation](#)

[IntelliJ IDEA GDSDL](#)

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 archiveArtifacts: Archive the artifacts

Files to archive

[Advanced...](#)

[Generate Pipeline Script](#)

[Global Variables](#)

There are many features of the Pipeline that are not steps. These are often exposed via global variables, which are not supported by the snippet generator. See the [Global Variables Reference](#) for details.

This page serves as documentation and a quick reference while working with pipelines. There is a snippet generator that helps you while working with the scripted pipeline to generate different portions of the pipeline. This is very important as a remembering aid since some pipeline steps have complicated syntax that might be difficult to master.

5. On the left-hand menu, open the Global Variable Reference item. This opens the reference page, entailing a global variable reference that would take a while to go through. In our case, we are looking for the `BRANCH_NAME` variable. We can find this under the `env` section. The following output displays a portion of the variables:

[env](#)

Environment variables are accessible from Groovy code as `env.VARNAME` or simply as `VARNAME`. You can write to such properties as well (only using the `env.` prefix):

```
env.MYTOOL_VERSION = '1.33'
node {
    sh '/usr/local/mytool-$MYTOOL_VERSION/bin/start'
}
```

These definitions will also be available via the REST API during the build or after its completion, and from upstream Pipeline builds using the `build` step.

However any variables set this way are global to the Pipeline build. For variables with node-specific content (such as file paths), you should instead use the `withEnv` step, to bind the variable only within a node block.

A set of environment variables are made available to all Jenkins projects, including Pipelines. The following is a general list of variable names that are available.

BRANCH_NAME

For a multibranch project, this will be set to the name of the branch being built, for example in case you wish to deploy to production from `master` but not from feature branches; if corresponding to some kind of change request, the name is generally arbitrary (refer to `CHANGE_ID` and `CHANGE_TARGET`).

CHANGE_ID

For a multibranch project corresponding to some kind of change request, this will be set to the change ID, such as a pull request number, if supported; else unset.

CHANGE_URL

For a multibranch project corresponding to some kind of change request, this will be set to the change URL, if supported; else unset.

CHANGE_TITLE

For a multibranch project corresponding to some kind of change request, this will be set to the title of the change, if supported; else unset.

CHANGE_AUTHOR

For a multibranch project corresponding to some kind of change request, this will be set to the username of the author of the proposed change, if supported; else unset.

CHANGE_AUTHOR_DISPLAY_NAME

For a multibranch project corresponding to some kind of change request, this will be set to the human name of the author, if supported; else unset.

CHANGE_AUTHOR_EMAIL

For a multibranch project corresponding to some kind of change request, this will be set to the email address of the author, if supported; else unset.

CHANGE_TARGET

For a multibranch project corresponding to some kind of change request, this will be set to the target or base branch to which the change could be merged, if supported; else unset.

BUILD_NUMBER

The current build number, such as "153"

The first variable is the `BRANCH_NAME`; this is followed by a short description of exactly what it does and how it can be used. We will add a conditional to our pipeline that will just print out a message that our application is being deployed when building the master branch.



Refer to the complete code, which has been placed at <https://bit.ly/2KwNgug>.

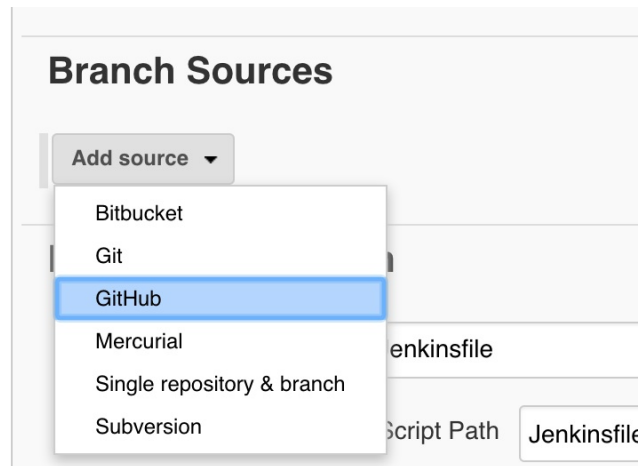
6. Add the following updated script to your pipeline after the testing stage to create a new multibranch project, and add our Jenkinsfile to version control. The final script looks as follows:

```

1 node('master') {
2     stage("Fetch Source Code") {
3         git 'https://github.com/TrainingByPackt/Beginning-Continuous-Delivery-With-Jenkins'
4     }
5
6     dir('Lesson5') {
7         printMessage('Running Pipeline')
8
9         stage("Testing") {
10             sh 'python test_functions.py'
11         }
12
13         stage("Deployment") {
14             if (env.BRANCH_NAME == 'master') {
15                 printMessage('Deploying the master branch')
16             } else {
17                 printMessage('No deployment configured for this branch')
18             }
19         }
20
21         printMessage('Pipeline Complete')
22     }
23 }
24
25 def printMessage(message) {
26     echo "${message}"
27 }
28

```

7. Create this script, add it to the root folder of your project while on the master branch, and name it `Jenkinsfile`. This Jenkinsfile is located in the `Lesson5` folder on the GitHub repository.
8. Add this script to the staging area, create a commit, and push it to the GitHub repository.
9. Create a new project on Jenkins and, under the Project type, select Multibranch pipeline and enter an appropriate name for the project.
10. Under Branch Sources, select Add source and select GitHub, as follows.



This will pop up a configuration form asking us for the origin repository. Configure the form, filling in Owner with your GitHub username or the username of where the repository is hosted, and Repository with the name of the repository. Leave the rest as defaults, just like in the following screenshot:

Branch Sources

GitHub

Credentials

kyaye.arnold@gmail.com/*****

Add

Owner

TrainingByPackt

Repository

Beginning-Continuous-Delivery-With-Jenkins

Behaviors

Discover branches

Strategy

Exclude branches that are also filed as PRs

Discover pull requests from origin

Strategy

Merging the pull request with the current target branch revision

Discover pull requests from forks

Strategy

Merging the pull request with the current target branch revision

Trust

Contributors

Add





Property strategy

All branches get the same properties

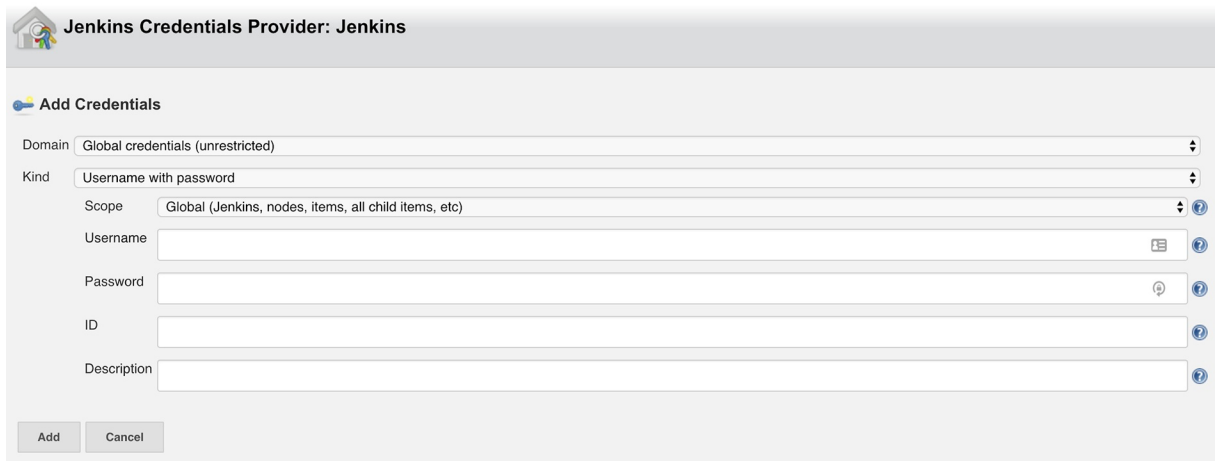
Add property

As we can see from the Credentials section, there is a warning informing us that adding credentials is recommended. We'll need to add GitHub credentials.

11. Select Add Dropdown and select Jenkins, as shown:

Credentials	- none -	
 Credentials are recommended		
Owner	arnoldokoth	<div> lesson-5-pipeline  Jenkins</div>

12. Under Kind, select Username with password.



13. Enter your GitHub username and password in their respective fields and select Add.



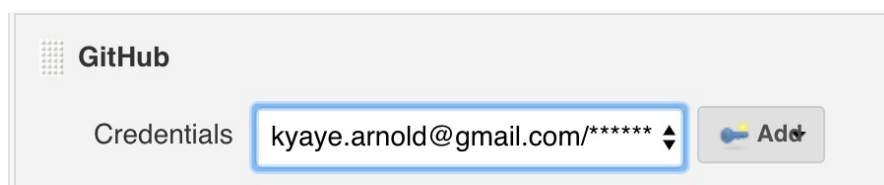
The ID will be automatically generated by Jenkins, and the description is optional.

14. Back on the project configuration page, select the credentials you just created under the Credentials section.



We mentioned the importance of the question mark icon in an earlier section. You can use this icon at this point to discover more about the rest of the configurations under the Behaviors section.

15. Under the Build Configuration section, set up the project as shown.



This configuration informs Jenkins where to find the `Jenkinsfile` we added to our project earlier, which in our case is in the root folder of our project. The rest of the configuration is not relevant for this project, so we can leave it as is.

16. Select Apply and Save to persist the configuration. Going back to the project dashboard, we can see that our pipeline already ran against the master branch.



lesson-5-pipeline

This is a sample repository for a demonstration on Lesson 5

Branches (1)		Pull Requests (0)				
S	W	Name ↓	Last Success	Last Failure	Last Duration	Fav
		master	6 min 9 sec - #1	N/A	2.4 sec	

Icon: [S](#) [M](#) [L](#)

Legend [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

- On the left-hand panel, select Open Blue Ocean. Under the Branches tab, select the first item, which in our case is master. We will view this project using Blue Ocean as follows:

Jenkins

PipelinesAdministrationLogout

lesson-5-pipeline ☆ ⚙

ActivityBranchesPull Requests

HEALTH	STATUS	BRANCH	COMMIT	LATEST MESSAGE	COMPLETED	
		master	—	Branch indexing	11 minutes ago	☆

From the output, we can see that our pipeline ran and added the new stage we defined in our script.

✓ lesson-5-pipeline 1

Pipeline Changes Tests Artifacts Logout X

Branch: master

Commit: 3775b23

2s

14 minutes ago

No changes

Branch indexing

Start

Fetch Source Code

Testing

Deployment

End

Deployment - < 1s

✓ > Deploying the master branch — Print Message

✓ > Pipeline Complete — Print Message

< 1s

< 1s

Looking closely at the deployment stage output, we can see that the deployment message that we configured for the master branch, Deploying the master branch, was displayed.

Deployment - <1s




✓	▼ Deploying the master branch — Print Message	<1s
	1 Deploying the master branch	
✓	> Pipeline Complete — Print Message	<1s

Building Pull Requests

Now that we've covered how to create multibranch projects, we will add a new branch to our project, as well as some functions and tests, and then push our branch to the remote repository. To build pull requests using Jenkins, follow these steps:


1. On your Git Bash in the project root, check out to a new branch and add the following snippet to the `functions.py` file.

```
22 def get_full_name(firstname, lastname):
23     """ Return the full name in the format firstname, lastname
24
25     Arguments:
26     firstname: First name e.g. John
27     lastname: Last name e.g. Doe
28     """
29     return lastname + ", " + firstname
30
```

 This snippet is located in the `Lesson5/functions.py` file. You can copy and paste it directly from this file. Refer to the complete code, which has been placed at <https://bit.ly/2uqgw1I>.

2. In the `test_functions.py` file, add the following unit test for the new function we just added.

```
def test_full_name(self):
    self.assertEqual("Doe, John", get_full_name("John", "Doe"))
```

 This file is already provided in the GitHub repository, under `Lesson5/test_functions.py`. Refer to the complete code, which has been placed at <https://bit.ly/2KYK0mZ>.

Optionally, you can run this test locally (if you have Python installed) to make sure that everything works by using `python test_functions.py`. This is also highlighted in the pipeline script.

3. Once the tests are okay, push the new branch to the remote and create a new pull request.

4. Going back to Jenkins, we can see our new branch and, under the Pull Requests tab, we can see that the Pull Request we created has been built by Jenkins using the same pipeline stages.

lesson-5-pipeline ☆ ⚙						Activity	Branches	Pull Requests
STATUS	PR	SUMMARY	AUTHOR		COMPLETED			
✓	2	Add Full Name Function	arnoldokoth		3 hours ago		⌵	

We can view the output of the build. Digging deeper into the Deployment stage, we can see from the output that no deployment was configured for the branch that ran as a result of the conditional we configured earlier.



Deployment - <1s			🔗 ⬇
✓	✓	No deployment configured for this branch — Print Message	<1s
1	✓	No deployment configured for this branch	
✓	>	Pipeline Complete — Print Message	<1s

At this point, we have managed to create an end-to-end pipeline with GitHub merge checks integrated. If any of the pipeline stages failed, GitHub would report the branch as not mergeable since there are errors in the pipeline.

Activity: Creating a Pipeline

Scenario

You have been provided with a Python code base with test files and custom requirements that the application depends on, and have been tasked with creating a Jenkins pipeline script that will run the tests and perform a mock deployment against the master branch, which will just print a message that the master branch is getting deployed on the console.

Aim

To get familiar with the GitHub build trigger and integrating a version-controlled project into Jenkins

Steps for Completion

1. Log in to GitHub and open the following URL: <https://github.com/TrainingByPackt/Beginning-Jenkins/tree/master/Lesson5/ActivityA>.
2. Create a fork of this repository to your account. You can perform this using the Fork button at the top right of the screen below the navigation bar.
3. After performing a fork, clone the repository (the fork that is now under your account) to your local machine. We can achieve this by using the `git clone <repo-url>` command.
4. Open the repository in a code editor of your choice.
5. Navigate to the `Lesson5/ActivityA` directory. Looking at the code base as it is right now, it has the following files:

```
.
├── Jenkinsfile.outline
├── Makefile
├── README.md
├── app
│   ├── app.py
│   └── tests
│       └── test_endpoints.py
└── requirements.txt
```

6. Create a Jenkinsfile (pipeline script) from the provided template. On the root folder, we will create a Jenkinsfile and copy over the contents of the template to the Jenkinsfile.



Refer to the complete code, which has been placed at <https://bit.ly/2JsE37w>.

After performing this operation, the folder structure will look as follows:

```
.
├── Jenkinsfile
├── Jenkinsfile.outline
├── Makefile
├── README.md
├── app
│   ├── app.py
│   └── tests
│       └── test_endpoints.py
└── requirements.txt
```

Opening our newly created Jenkinsfile, we can see the following code inside it:

```

1 node {
2   printMessage("Pipeline Start")
3
4   stage("Fetch Source Code") {
5     git 'https://github.com/<your-profile>/Beginning-Continuous-Delivery-With-Jenkins'
6   }
7
8   dir('Lesson5/ActivityA') {
9     stage("Install Requirements") {
10      sh '<enter makefile command to install requirements>'
11    }
12
13    stage("Run Tests") {
14      sh '<enter makefile command to run tests>'
15    }
16
17    stage("Deploy") {
18      if (env.BRANCH_NAME == "master") {
19        printMessage("")
20      } else {
21        printMessage("")
22      }
23    }
24  }
25
26  printMessage("Pipeline End")
27 }
28
29 def printMessage(message) {
30   echo "${message}"
31 }
32

```

7. Make changes to this script and replace the parts enclosed in angle brackets <> with the appropriate commands/values (this also includes the angle brackets, as depicted in the previous screenshot).



Refer to the complete code, which has been placed at <https://bit.ly/2LcXhD6>.

For the first stage, which in our case is Fetch Source Code, enter the URL for the forked repository, which is the repository that is now under your account:

```

stage("Fetch Source Code") {
  git "https://github.com/<your-profile>/Beginning-Continuous-Delivery-With-Jenkins"
}

```

8. For the next stage, add the command listed in the Makefile that runs the

tests. Our Makefile is very basic and has the following tasks:

```
M Makefile x
1  install:
2      virtualenv testing
3      . testing/bin/activate
4      pip install --user -r requirements.txt
5
6  jenkins_test:
7      testing/bin/nosetests app/ -v
8
```

The first task creates a virtual environment (since we are working with Python, this creates a virtual environment for our tests to run in) and installs the project requirements using `pip`. `pip` is a Python package manager similar to `npm` for Node.js. Our requirements are listed in the `requirements.txt` file. The second task runs our tests using the `nosetests` test runner.



Refer to the complete code, which has been placed at <https://bit.ly/2Nh3taE>.

9. For the Install Requirements stage, replace the text enclosed in angle brackets with the `make install` command. The final result should look as follows:

```
stage("Install Requirements") {
    sh 'make install'
}
```


10. Repeat this step but for the Run Tests stage, this time replacing the text with the `jenkins_test` command.
11. For the last stage, which in our case is the Deploy stage, we will just print appropriate messages for the conditional we currently have in our pipeline script. The final result should look something like the following:

```
stage("Deploy") {  
    if (env.BRANCH_NAME == "master") {  
        printMessage("deploying master branch")  
    } else {  
        printMessage("no deployment specified for this branch")  
    }  
}
```

At this point, we are done updating our pipeline script and need to push it to the remote repository. Push the changes you made to the remote repository.

12. Go to the Jenkins dashboard and create a project, give it an appropriate name, and select the Multibranch pipeline on the project type.
13. On the project configuration, under Branch Sources, add the repository under your account. For the credentials, you can reuse the ones we created earlier in this chapter. The final Branch Sources configuration should look similar to that in the following screenshot:

Branch Sources

GitHub

Credentials:

Owner:

Repository:

Behaviors

Discover branches

Strategy:

Discover pull requests from origin

Strategy:

Discover pull requests from forks

Strategy:







Trust:

Property strategy:

Leave the Build Configuration as default. The Mode should be by Jenkinsfile and the Script Path should be Jenkinsfile.

- Click Apply and Save to save the pipeline project and open the Blue Ocean dashboard.

If the GitHub webhook was configured correctly, Jenkins will perform a build on the master branch and run the stages that you configured on the pipeline script. Start the build manually by clicking on the play button as follows:

HEALTH	STATUS	BRANCH	COMMIT	LATEST MESSAGE	COMPLETED
		master	bf6d9b4	Started by user Arnold Okoth	a few seconds ago    

Summary

In this chapter, we have learned about branching with Git and how this enables a CI workflow. We then demonstrated working with the Jenkinsfile and how to create pipelines for our projects hosted on GitHub. Lastly, we created a multibranch pipeline and demonstrated how to configure our pipeline to perform different actions while building different branches. In the next chapter, we are going to cover how to set up distributed builds on Jenkins and how to configure our projects to run on our agents, thus creating a faster build environment