Jenkins User Documentation Home

Guided Tour

- Getting started
- Creating your first Pipeline
- Running multiple steps
- Defining execution environments
- Using environment variables
- Recording test results and artifacts
- Cleaning up and notifications
- Deployment

Tutorials

- Overview
- Build a Java app with Maven
- Build a Node.js and React app with npm
- Build a Python app with PyInstaller
- Build a LabVIEW app
- Create a Pipeline in Blue Ocean
- End-to-End Multibranch Pipeline Project Creation

Handbook

- User Handbook overview
- Installing Jenkins

- Using Jenkins
- Pipeline
  - Getting started with Pipeline
  - Using a Jenkinsfile
  - Running Pipelines
  - Branches and Pull Requests
  - Using Docker with Pipeline
  - Extending with Shared Libraries
  - Pipeline Development Tools
  - Pipeline Syntax
  - Pipeline Best Practices
  - Scaling Pipelines
- Blue Ocean
- Managing Jenkins
- System Administration
- Scaling Jenkins
- Appendix
- Glossary

Resources

- Pipeline Syntax reference
- Pipeline Steps reference
- LTS Upgrade Guide

Tutorial Blog Posts

- Introducing Tutorials in the Jenkins User Documentation
- Pipeline Development Tools
- Getting Started with the Blue Ocean Dashboard

View all tutorial blog posts

⇐ Getting started with Pipeline
⇑ Pipeline
Index
Running Pipelines ⇒

Using a Jenkinsfile

Table of Contents

- Creating a Jenkinsfile
  - Build
  - Test
  - Deploy
- Working with your Jenkinsfile
  - String interpolation
  - Using environment variables
    - Setting environment variables
    - Setting environment variables dynamically
  - Handling credentials
    - For secret text, usernames and passwords, and secret files
      - Secret text

        - Usernames and passwords
        - Secret files
    - For other credential types
        - Combining credentials in one step
  - Handling parameters
  - Handling failure
  - Using multiple agents
  - Optional step arguments
  - Advanced Scripted Pipeline
    - Parallel execution

This section builds on the information covered in Getting started with Pipeline and introduces more useful steps, common patterns, and demonstrates some non-trivial `Jenkinsfile` examples.

Creating a `Jenkinsfile`, which is checked into source control [1], provides a number of immediate benefits:

- Code review/iteration on the Pipeline
- Audit trail for the Pipeline
- Single source of truth [2] for the Pipeline, which can be viewed and edited by multiple members of the project.

Pipeline supports two syntaxes, Declarative (introduced in Pipeline 2.5) and Scripted Pipeline. Both of which support building continuous delivery pipelines. Both may be used to define a Pipeline in either the web UI or with a `Jenkinsfile`, though it's generally considered a best practice to create a `Jenkinsfile` and check the file into the source control repository.

Creating a Jenkinsfile

As discussed in the Defining a Pipeline in SCM, a `Jenkinsfile` is a text file that contains the definition of a Jenkins Pipeline and is checked into source control. Consider the following Pipeline which implements a basic three-stage continuous delivery pipeline.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

Not all Pipelines will have these same three stages, but it is a good starting point to define them for most projects. The sections below will demonstrate the creation and execution of a simple Pipeline in a test installation of Jenkins.

> It is assumed that there is already a source control repository set up for the project and a Pipeline has been defined in Jenkins following these instructions.

Using a text editor, ideally one which supports Groovy syntax highlighting, create a new `Jenkinsfile` in the root directory of the project.

The Declarative Pipeline example above contains the minimum necessary structure to implement a continuous delivery pipeline. The agent directive, which is required, instructs Jenkins to allocate an executor and workspace for the Pipeline. Without an `agent` directive, not only is the Declarative Pipeline not valid, it would not be capable of doing any work! By default the `agent` directive ensures that the source repository is checked out and made available for steps in the subsequent stages`

The stages directive, and steps directives are also required for a valid Declarative Pipeline as they instruct Jenkins what to execute and in which stage it should be executed.

For more advanced usage with Scripted Pipeline, the example above `node` is a crucial first step as it allocates an executor and workspace for the Pipeline. In essence, without `node`, a Pipeline cannot do any work! From within `node`, the first order of business will be to checkout the source code for this project. Since the `Jenkinsfile` is being pulled directly from source control, Pipeline provides a quick and easy way to access the right revision of the source code

Jenkinsfile (Scripted Pipeline)

```
node {
    checkout scm (1)
    /* .. snip .. */
}
```

1 The `checkout` step will checkout code from source control; `scm` is a special variable which instructs the `checkout` step to clone the specific revision which triggered this Pipeline run.

Build

For many projects the beginning of "work" in the Pipeline would be the "build" stage. Typically this stage of the Pipeline will be where source code is assembled, compiled, or packaged. The `Jenkinsfile` is **not** a replacement for an existing build tool such as GNU/Make, Maven, Gradle, etc, but rather can be viewed as a glue layer to bind the multiple phases of a project's development lifecycle (build, test, deploy, etc) together.

Jenkins has a number of plugins for invoking practically any build tool in general use, but this example will simply invoke `make` from a shell step (`sh`). The `sh` step assumes the system is Unix/Linux-based, for Windows-based systems the `bat` could be used instead.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Build') {
```

```
            steps {
                sh 'make' (1)
                archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true (2)
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

1. The `sh` step invokes the `make` command and will only continue if a zero exit code is returned by the command. Any non-zero exit code will fail the Pipeline.
2. `archiveArtifacts` captures the files built matching the include pattern (`**/target/*.jar`) and saves them to the Jenkins master for later retrieval.

Archiving artifacts is not a substitute for using external artifact repositories such as Artifactory or Nexus and should be considered only for basic reporting and file archival.

Test

Running automated tests is a crucial component of any successful continuous delivery process. As such, Jenkins has a number of test recording, reporting, and visualization facilities provided by a number of plugins. At a fundamental level, when there are test failures, it is useful to have Jenkins record the failures for reporting and visualization in the web UI. The example below uses the `junit` step, provided by the JUnit plugin.

In the example below, if tests fail, the Pipeline is marked "unstable", as denoted by a yellow ball in the web UI. Based on the recorded test reports, Jenkins can also provide historical trend analysis and visualization.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Test') {
            steps {
                /* `make check` returns non-zero on test failures,
                * using `true` to allow the Pipeline to continue nonetheless
                */
                sh 'make check || true' (1)
                junit '**/target/*.xml' (2)
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

1. Using an inline shell conditional (`sh 'make check || true'`) ensures that the `sh` step always sees a zero exit code, giving the `junit` step the opportunity to capture and process the test reports. Alternative approaches to this are covered in more detail in the Handling failure section below.
2. `junit` captures and associates the JUnit XML files matching the inclusion pattern (`**/target/*.xml`).

Deploy

Deployment can imply a variety of steps, depending on the project or organization requirements, and may be anything from publishing built artifacts to an Artifactory server, to pushing code to a production system.

At this stage of the example Pipeline, both the "Build" and "Test" stages have successfully executed. In essence, the "Deploy" stage will only execute assuming previous stages completed successfully, otherwise the Pipeline would have exited early.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Deploy') {
            when {
              expression {
                currentBuild.result == null || currentBuild.result == 'SUCCESS' (1)
              }
            }
            steps {
                sh 'make publish'
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

1 Accessing the `currentBuild.result` variable allows the Pipeline to determine if there were any test failures. In which case, the value would be `UNSTABLE`.

Assuming everything has executed successfully in the example Jenkins Pipeline, each successful Pipeline run will have associated build artifacts archived, test results reported upon and the full console output all in Jenkins.

A Scripted Pipeline can include conditional tests (shown above), loops, try/catch/finally blocks and even functions. The next section will cover this advanced Scripted Pipeline syntax in more detail.

Working with your Jenkinsfile

The following sections provide details about handling:

- specific Pipeline syntax in your `Jenkinsfile` and
- features and functionality of Pipeline syntax which are essential in building your application or Pipeline project.

String interpolation

Jenkins Pipeline uses rules identical to Groovy for string interpolation. Groovy's String interpolation support can be confusing to many newcomers to the language. While Groovy supports declaring a string with either single quotes, or double quotes, for example:

```
def singlyQuoted = 'Hello'
def doublyQuoted = "World"
```

Only the latter string will support the dollar-sign (`$`) based string interpolation, for example:

```
def username = 'Jenkins'
echo 'Hello Mr. ${username}'
echo "I said, Hello Mr. ${username}"
```

Would result in:

```
Hello Mr. ${username}
I said, Hello Mr. Jenkins
```

Understanding how to use string interpolation is vital for using some of Pipeline's more advanced features.

Using environment variables

Jenkins Pipeline exposes environment variables via the global variable `env`, which is available from anywhere within a `Jenkinsfile`. The full list of environment variables accessible from within Jenkins Pipeline is documented at ${YOUR_JENKINS_URL}/pipeline-syntax/globals#env and includes:

BUILD_ID

The current build ID, identical to BUILD_NUMBER for builds created in Jenkins versions 1.597+

BUILD_NUMBER

The current build number, such as "153"

BUILD_TAG

String of jenkins-${JOB_NAME}-${BUILD_NUMBER}. Convenient to put into a resource file, a jar file, etc for easier identification

BUILD_URL

The URL where the results of this build can be found (for example http://buildserver/jenkins/job/MyJobName/17/)

EXECUTOR_NUMBER

The unique number that identifies the current executor (among executors of the same machine) performing this build. This is the number you see in the "build executor status", except that the number starts from 0, not 1

JAVA_HOME

If your job is configured to use a specific JDK, this variable is set to the JAVA_HOME of the specified JDK. When this variable is set, PATH is also updated to include the bin subdirectory of JAVA_HOME

JENKINS_URL

Full URL of Jenkins, such as https://example.com:port/jenkins/ (NOTE: only available if Jenkins URL set in "System Configuration")

JOB_NAME

Name of the project of this build, such as "foo" or "foo/bar".

NODE_NAME

The name of the node the current build is running on. Set to 'master' for master node.

WORKSPACE

The absolute path of the workspace

Referencing or using these environment variables can be accomplished like accessing any key in a Groovy Map, for example:

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

Setting environment variables

Setting an environment variable within a Jenkins Pipeline is accomplished differently depending on whether Declarative or Scripted Pipeline is used.

Declarative Pipeline supports an environment directive, whereas users of Scripted Pipeline must use the `withEnv` step.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    environment { (1)
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment { (2)
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

1 An `environment` directive used in the top-level `pipeline` block will apply to all steps within the Pipeline.

2 An `environment` directive defined within a `stage` will only apply the given environment variables to steps within the `stage`.

Setting environment variables dynamically

Environment variables can be set at run time and can be used by shell scripts (`sh`), Windows batch scripts (`bat`) and PowerShell scripts (`powershell`). Each script can either `returnStatus` or `returnStdout`. More information on scripts.

Below is an example in a declarative pipeline using `sh` (shell) with both `returnStatus` and `returnStdout`.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any (1)
    environment {
        // Using returnStdout
        CC = """${sh(
                returnStdout: true,
                script: 'echo "clang"'
            )}""" (2)
        // Using returnStatus
        EXIT_STATUS = """${sh(
                returnStatus: true,
                script: 'exit 1'
            )}"""
    }
    stages {
        stage('Example') {
            environment {
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

1 An `agent` must be set at the top level of the pipeline. This will fail if agent is set as `agent none`.

2 When using `returnStdout` a trailing whitespace will be appended to the returned string. Use `.trim()` to remove this.

Handling credentials

Credentials configured in Jenkins can be handled in Pipelines for immediate use. Read more about using credentials in Jenkins on the Using credentials page.

For secret text, usernames and passwords, and secret files

Jenkins' declarative Pipeline syntax has the `credentials()` helper method (used within the `environment` directive) which supports secret text, username and password, as well as secret file credentials. If you want to handle other types of credentials, refer to the For other credential types section (below).

Secret text

The following Pipeline code shows an example of how to create a Pipeline using environment variables for secret text credentials.

In this example, two secret text credentials are assigned to separate environment variables to access Amazon Web Services (AWS). These credentials would have been configured in Jenkins with their respective credential IDs `jenkins-aws-secret-key-id` and `jenkins-aws-secret-access-key`.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent {
        // Define agent details here
    }
    environment {
        AWS_ACCESS_KEY_ID     = credentials('jenkins-aws-secret-key-id')
        AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')
```

```
    }
    stages {
        stage('Example stage 1') {
            steps {
                // (1)
            }
        }
        stage('Example stage 2') {
            steps {
                // (2)
            }
        }
    }
}
```

1 You can reference the two credential environment variables (defined in this Pipeline's `environment` directive), within this stage's steps using the syntax `$AWS_ACCESS_KEY_ID` and `$AWS_SECRET_ACCESS_KEY`. For example, here you can authenticate to AWS using the secret text credentials assigned to these credential variables.
To maintain the security and anonymity of these credentials, if the job displays the value of these credential variables from within the Pipeline (e.g. `echo $AWS_SECRET_ACCESS_KEY`), Jenkins only returns the value "****" to reduce the risk of secret information being disclosed to the console output and any logs. Any sensitive information in credential IDs themselves (such as usernames) are also returned as "****" in the Pipeline run's output.
This only reduces the risk of **accidental exposure**. It does not prevent a malicious user from capturing the credential value by other means. A Pipeline that uses credentials can also disclose those credentials. Don't allow untrusted Pipeline jobs to use trusted credentials.

2 In this Pipeline example, the credentials assigned to the two `AWS_…` environment variables are scoped globally for the entire Pipeline, so these credential variables could also be used in this stage's steps. If, however, the `environment` directive in this Pipeline were moved to a specific stage (as is the case in the Usernames and passwords Pipeline example below), then these `AWS_…` environment variables would only be scoped to the steps in that stage.

Usernames and passwords

The following Pipeline code snippets show an example of how to create a Pipeline using environment variables for username and password credentials.

In this example, username and password credentials are assigned to environment variables to access a Bitbucket repository in a common account or team for your organization; these credentials would have been configured in Jenkins with the credential ID `jenkins-bitbucket-common-creds`.

When setting the credential environment variable in the `environment` directive:

```
environment {
    BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-creds')
}
```

this actually sets the following three environment variables:

- `BITBUCKET_COMMON_CREDS` - contains a username and a password separated by a colon in the format `username:password`.
- `BITBUCKET_COMMON_CREDS_USR` - an additional variable containing the username component only.
- `BITBUCKET_COMMON_CREDS_PSW` - an additional variable containing the password component only.

By convention, variable names for environment variables are typically specified in capital case, with individual

words separated by underscores. You can, however, specify any legitimate variable name using lower case characters. Bear in mind that the additional environment variables created by the `credentials()` method (above) will always be appended with `_USR` and `_PSW` (i.e. in the format of an underscore followed by three capital letters).

The following code snippet shows the example Pipeline in its entirety:

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent {
        // Define agent details here
    }
    stages {
        stage('Example stage 1') {
            environment {
                BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-creds')
            }
            steps {
                // (1)
            }
        }
        stage('Example stage 2') {
            steps {
                // (2)
            }
        }
    }
}
```

The following credential environment variables (defined in this Pipeline's `environment` directive) are available within this stage's steps and can be referenced using the syntax:

- `$BITBUCKET_COMMON_CREDS`
- `$BITBUCKET_COMMON_CREDS_USR`
- `$BITBUCKET_COMMON_CREDS_PSW`

1 For example, here you can authenticate to Bitbucket with the username and password assigned to these credential variables.
To maintain the security and anonymity of these credentials, if the job displays the value of these credential variables from within the Pipeline the same behavior described in the Secret text example above applies to these username and password credential variable types too.
This only reduces the risk of **accidental exposure**. It does not prevent a malicious user from capturing the credential value by other means. A Pipeline that uses credentials can also disclose those credentials. Don't allow untrusted Pipeline jobs to use trusted credentials.

2 In this Pipeline example, the credentials assigned to the three `COMMON_BITBUCKET_CREDS…` environment variables are scoped only to `Example stage 1`, so these credential variables are not available for use in this `Example stage 2` stage's steps. If, however, the `environment` directive in this Pipeline were moved immediately within the `pipeline` block (as is the case in the Secret text Pipeline example above), then these `COMMON_BITBUCKET_CREDS…` environment variables would be scoped globally and could be used in any stage's steps.

Secret files

As far as Pipelines are concerned, secret files are handled in exactly the same manner as secret text (above).

Essentially, the only difference between secret text and secret file credentials are that for secret text, the credential itself is entered directly into Jenkins whereas for a secret file, the credential is originally stored in a file which is then uploaded to Jenkins.

Unlike secret text, secret files cater for credentials that are:

- too unwieldy to enter directly into Jenkins, and/or
- in binary format, such as a GPG file.

For other credential types

If you need to set credentials in a Pipeline for anything other than secret text, usernames and passwords, or secret files (above) - i.e SSH keys or certificates, then use Jenkins' **Snippet Generator** feature, which you can access through Jenkins' classic UI.

To access the **Snippet Generator** for your Pipeline project/item:

1. From the Jenkins home page (i.e. the Dashboard of Jenkins' classic UI), click the name of your Pipeline project/item.
2. On the left, click **Pipeline Syntax** and ensure that the **Snippet Generator** link is in bold at the top-left. (If not, click its link.)
3. From the **Sample Step** field, choose **withCredentials: Bind credentials to variables**.
4. Under **Bindings**, click **Add** and choose from the dropdown:
    - **SSH User Private Key** - to handle SSH public/private key pair credentials, from which you can specify:
        - **Key File Variable** - the name of the environment variable that will be bound to these credentials. Jenkins actually assigns this temporary variable to the secure location of the private key file required in the SSH public/private key pair authentication process.
        - **Passphrase Variable** (*Optional*) - the name of the environment variable that will be bound to the passphrase associated with the SSH public/private key pair.
        - **Username Variable** (*Optional*) - the name of the environment variable that will be bound to username associated with the SSH public/private key pair.
        - **Credentials** - choose the SSH public/private key credentials stored in Jenkins. The value of this field is the credential ID, which Jenkins writes out to the generated snippet.
    - **Certificate** - to handle PKCS#12 certificates, from which you can specify:
        - **Keystore Variable** - the name of the environment variable that will be bound to these credentials. Jenkins actually assigns this temporary variable to the secure location of the certificate's keystore required in the certificate authentication process.
        - **Password Variable** (*Optional*) - the name of the environment variable that will be bound to the password associated with the certificate.
        - **Alias Variable** (*Optional*) - the name of the environment variable that will be bound to the unique alias associated with the certificate.

        - **Credentials** - choose the certificate credentials stored in Jenkins. The value of this field is the credential ID, which Jenkins writes out to the generated snippet.

    - **Docker client certificate** - to handle Docker Host Certificate Authentication.

5. Click **Generate Pipeline Script** and Jenkins generates a `withCredentials( … ) { … }` Pipeline step snippet for the credentials you specified, which you can then copy and paste into your Declarative or Scripted Pipeline code.
   Notes:

   - The **Credentials** fields (above) show the names of credentials configured in Jenkins. However, these values are converted to credential IDs after clicking **Generate Pipeline Script**.

   - To combine more than one credential in a single `withCredentials( … ) { … }` Pipeline step, see Combining credentials in one step (below) for details.

SSH User Private Key example

```
withCredentials(bindings: [sshUserPrivateKey(credentialsId: 'jenkins-ssh-key-for-abc', \
                                             keyFileVariable: 'SSH_KEY_FOR_ABC', \
                                             passphraseVariable: '', \
                                             usernameVariable: '')]) {
  // some block
}
```

The optional `passphraseVariable` and `usernameVariable` definitions can be deleted in your final Pipeline code.

Certificate example

```
withCredentials(bindings: [certificate(aliasVariable: '', \
                                       credentialsId: 'jenkins-certificate-for-xyz', \
                                       keystoreVariable: 'CERTIFICATE_FOR_XYZ', \
                                       passwordVariable: 'XYZ-CERTIFICATE-PASSWORD')]) {
  // some block
}
```

The optional `aliasVariable` and `passwordVariable` variable definitions can be deleted in your final Pipeline code.

The following code snippet shows an example Pipeline in its entirety, which implements the **SSH User Private Key** and **Certificate** snippets above:

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent {
        // define agent details
    }
    stages {
        stage('Example stage 1') {
            steps {
                withCredentials(bindings: [sshUserPrivateKey(credentialsId: 'jenkins-ssh-key-for-abc', \
                                                             keyFileVariable: 'SSH_KEY_FOR_ABC')]) {
                  // (1)
                }
                withCredentials(bindings: [certificate(credentialsId: 'jenkins-certificate-for-xyz', \
                                                       keystoreVariable: 'CERTIFICATE_FOR_XYZ', \
                                                       passwordVariable: 'XYZ-CERTIFICATE-PASSWORD')]) {
                  // (2)
                }
            }
        }
        stage('Example stage 2') {
```

```
            steps {
                // (3)
            }
        }
    }
}
```

1. Within this step, you can reference the credential environment variable with the syntax `$SSH_KEY_FOR_ABC`. For example, here you can authenticate to the ABC application with its configured SSH public/private key pair credentials, whose **SSH User Private Key** file is assigned to `$SSH_KEY_FOR_ABC`.
2. Within this step, you can reference the credential environment variable with the syntax `$CERTIFICATE_FOR_XYZ` and `$XYZ-CERTIFICATE-PASSWORD`. For example, here you can authenticate to the XYZ application with its configured certificate credentials, whose **Certificate**'s keystore file and password are assigned to the variables `$CERTIFICATE_FOR_XYZ` and `$XYZ-CERTIFICATE-PASSWORD`, respectively.
3. In this Pipeline example, the credentials assigned to the `$SSH_KEY_FOR_ABC`, `$CERTIFICATE_FOR_XYZ` and `$XYZ-CERTIFICATE-PASSWORD` environment variables are scoped only within their respective `withCredentials( … ) { … }` steps, so these credential variables are not available for use in this `Example stage 2` stage's steps.

To maintain the security and anonymity of these credentials, if you attempt to retrieve the value of these credential variables from within these `withCredentials( … ) { … }` steps, the same behavior described in the Secret text example (above) applies to these SSH public/private key pair credential and certificate variable types too. This only reduces the risk of **accidental exposure**. It does not prevent a malicious user from capturing the credential value by other means. A Pipeline that uses credentials can also disclose those credentials. Don't allow untrusted Pipeline jobs to use trusted credentials.

- When using the **Sample Step** field's **withCredentials: Bind credentials to variables** option in the **Snippet Generator**, only credentials which your current Pipeline project/item has access to can be selected from any **Credentials** field's list. While you can manually write a `withCredentials( … ) { … }` step for your Pipeline (like the examples above), using the **Snippet Generator** is recommended to avoid specifying credentials that are out of scope for this Pipeline project/item, which when run, will make the step fail.
- You can also use the **Snippet Generator** to generate `withCredentials( … ) { … }` steps to handle secret text, usernames and passwords and secret files. However, if you only need to handle these types of credentials, it is recommended you use the relevant procedure described in the section above for improved Pipeline code readability.
- The use of **single-quotes** instead of **double-quotes** to define the `script` (the implicit parameter to `sh`) in Groovy above. The single-quotes will cause the secret to be expanded by the shell as an environment variable. The double-quotes are potentially less secure as the secret is interpolated by Groovy, and so typical operating system process listings (as well as Blue Ocean, and the pipeline steps tree in the classic UI) will accidentally disclose it :

```
node {
  withCredentials([string(credentialsId: 'mytoken', variable: 'TOKEN')]) {
    sh /* WRONG! */ """
      set +x
      curl -H 'Token: $TOKEN' https://some.api/
    """
    sh /* CORRECT */ '''
      set +x
      curl -H 'Token: $TOKEN' https://some.api/
    '''
  }
}
```

Combining credentials in one step

Using the **Snippet Generator**, you can make multiple credentials available within a single `withCredentials( … ) { … }` step by doing the following:

1. From the Jenkins home page (i.e. the Dashboard of Jenkins' classic UI), click the name of your Pipeline project/item.
2. On the left, click **Pipeline Syntax** and ensure that the **Snippet Generator** link is in bold at the top-left. (If not, click its link.)
3. From the **Sample Step** field, choose **withCredentials: Bind credentials to variables**.
4. Click **Add** under **Bindings**.
5. Choose the credential type to add to the `withCredentials( … ) { … }` step from the dropdown list.
6. Specify the credential **Bindings** details. Read more above these in the procedure under For other credential types (above).
7. Repeat from "Click **Add** …" (above) for each (set of) credential/s to add to the `withCredentials( … ) { … }` step.
8. Click **Generate Pipeline Script** to generate the final `withCredentials( … ) { … }` step snippet.

Handling parameters

Declarative Pipeline supports parameters out-of-the-box, allowing the Pipeline to accept user-specified parameters at runtime via the parameters directive. Configuring parameters with Scripted Pipeline is done with the `properties` step, which can be found in the Snippet Generator.

If you configured your pipeline to accept parameters using the **Build with Parameters** option, those parameters are accessible as members of the `params` variable.

Assuming that a String parameter named "Greeting" has been configuring in the `Jenkinsfile`, it can access that parameter via `${params.Greeting}`:

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    parameters {
        string(name: 'Greeting', defaultValue: 'Hello', description: 'How should I greet the world?')
    }
    stages {
        stage('Example') {
            steps {
                echo "${params.Greeting} World!"
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

Handling failure

Declarative Pipeline supports robust failure handling by default via its post section which allows declaring a number of different "post conditions" such as: `always`, `unstable`, `success`, `failure`, and `changed`. The Pipeline Syntax section provides more detail on how to use the various post conditions.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh 'make check'
            }
        }
    }
    post {
        always {
            junit '**/target/*.xml'
        }
        failure {
            mail to: team@example.com, subject: 'The Pipeline failed :('
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

Scripted Pipeline however relies on Groovy's built-in `try`/`catch`/`finally` semantics for handling failures during execution of the Pipeline.

In the Test example above, the `sh` step was modified to never return a non-zero exit code (`sh 'make check || true'`). This approach, while valid, means the following stages need to check `currentBuild.result` to know if there has been a test failure or not.

An alternative way of handling this, which preserves the early-exit behavior of failures in Pipeline, while still giving `junit` the chance to capture test reports, is to use a series of `try`/`finally` blocks:

Using multiple agents

In all the previous examples, only a single agent has been used. This means Jenkins will allocate an executor wherever one is available, regardless of how it is labeled or configured. Not only can this behavior be overridden, but Pipeline allows utilizing multiple agents in the Jenkins environment from within the *same* `Jenkinsfile`, which can helpful for more advanced use-cases such as executing builds/tests across multiple platforms.

In the example below, the "Build" stage will be performed on one agent and the built results will be reused on two subsequent agents, labelled "linux" and "windows" respectively, during the "Test" stage.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent any
            steps {
                checkout scm
                sh 'make'
                stash includes: '**/target/*.jar', name: 'app' (1)
            }
        }
        stage('Test on Linux') {
```

```
            agent { (2)
                label 'linux'
            }
            steps {
                unstash 'app' (3)
                sh 'make check'
            }
            post {
                always {
                    junit '**/target/*.xml'
                }
            }
        }
        stage('Test on Windows') {
            agent {
                label 'windows'
            }
            steps {
                unstash 'app'
                bat 'make check' (4)
            }
            post {
                always {
                    junit '**/target/*.xml'
                }
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

1. The `stash` step allows capturing files matching an inclusion pattern (`**/target/*.jar`) for reuse within the *same* Pipeline. Once the Pipeline has completed its execution, stashed files are deleted from the Jenkins master.
2. The parameter in `agent`/`node` allows for any valid Jenkins label expression. Consult the Pipeline Syntax section for more details.
3. `unstash` will retrieve the named "stash" from the Jenkins master into the Pipeline's current workspace.
4. The `bat` script allows for executing batch scripts on Windows-based platforms.

Optional step arguments

Pipeline follows the Groovy language convention of allowing parentheses to be omitted around method arguments.

Many Pipeline steps also use the named-parameter syntax as a shorthand for creating a Map in Groovy, which uses the syntax `[key1: value1, key2: value2]`. Making statements like the following functionally equivalent:

```
git url: 'git://example.com/amazing-project.git', branch: 'master'
git([url: 'git://example.com/amazing-project.git', branch: 'master'])
```

For convenience, when calling steps taking only one parameter (or only one mandatory parameter), the parameter name may be omitted, for example:

```
sh 'echo hello' /* short form  */
sh([script: 'echo hello'])  /* long form */
```

Advanced Scripted Pipeline

Scripted Pipeline is a domain-specific language [3] based on Groovy, most Groovy syntax can be used in Scripted Pipeline without modification.

Parallel execution

The example in the section above runs tests across two different platforms in a linear series. In practice, if the `make check` execution takes 30 minutes to complete, the "Test" stage would now take 60 minutes to complete!

Fortunately, Pipeline has built-in functionality for executing portions of Scripted Pipeline in parallel, implemented in the aptly named `parallel` step.

Refactoring the example above to use the `parallel` step:

Jenkinsfile (Scripted Pipeline)

```
stage('Build') {
    /* .. snip .. */
}

stage('Test') {
    parallel linux: {
        node('linux') {
            checkout scm
            try {
                unstash 'app'
                sh 'make check'
            }
            finally {
                junit '**/target/*.xml'
            }
        }
    },
    windows: {
        node('windows') {
            /* .. snip .. */
        }
    }
}
```

Instead of executing the tests on the "linux" and "windows" labelled nodes in series, they will now execute in parallel assuming the requisite capacity exists in the Jenkins environment.

---

1. en.wikipedia.org/wiki/Source_control_management
2. en.wikipedia.org/wiki/Single_Source_of_Truth
3. en.wikipedia.org/wiki/Domain-specific_language

⇐ Getting started with Pipeline
⇑ Pipeline
Index
Running Pipelines ⇒

Type the answer to 1 plus 6 before clicking "Submit" below.

Submit

See existing feedback here.

Improve this page | Page history

Resources

- Downloads
- Blog
- Documentation
- Plugins
- Security
- Contributing

Project

- Structure and governance
- Issue tracker
- Wiki
- GitHub
- Jenkins on Jenkins

Community

- Events
- Mailing lists
- Chats
- Special Interest Groups
- Twitter
- Reddit

Other

- Code of Conduct
- Press information
- Merchandise
- Artwork
- Awards