

# Defining a Pipeline

Scripted Pipeline is written in [Groovy](#). The relevant bits of [Groovy syntax](#) will be introduced as necessary in this document, so while an understanding of Groovy is helpful, it is not required to work with Pipeline.

A basic Pipeline can be created in either of the following ways:

- By entering a script directly in the Jenkins web UI.
- By creating a [Jenkinsfile](#) which can be checked into a project's source control repository.

The syntax for defining a Pipeline with either approach is the same, but while Jenkins supports entering Pipeline directly into the web UI, it's generally considered best practice to define the Pipeline in a [Jenkinsfile](#) which Jenkins will then load directly from source control. [9: [en.wikipedia.org/wiki/Source\\_control\\_management](https://en.wikipedia.org/wiki/Source_control_management)]

## Defining a Pipeline in the Web UI

To create a basic Pipeline in the Jenkins web UI, follow these steps:

- Click **New Item** on Jenkins home page.



- Enter a name for your Pipeline, select **Pipeline** and click **OK**.


### CAUTION

Jenkins uses the name of the Pipeline to create directories on disk. Pipeline names which include spaces may uncover bugs in scripts which do not expect paths to contain spaces.


Enter an item name

an-example


» Required field

**Freestyle project**


This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**


Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**External Job**


This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.

**Multi-configuration project**


Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**GitHub Organization**

Scans a GitHub organization (or user account) for all repositories matching some defined markers.

**Multibranch Pipeline**

Creates a set of Pipeline projects according to detected branches in one SCM repository.

OK

- In the **Script** text area, enter a Pipeline and click **Save**.

23

## Pipeline

Definition Pipeline script ▼

Script

```
1 node {  
2   echo "Hello World"  
3 }
```


try sample Pipeline... ▼ ⓘ

☒ Use Groovy Sandbox ⓘ

[Pipeline Syntax](#)










Save Apply

- Click **Build Now** to run the Pipeline.



# Jenkins

Jenkins > an-example >

-  [Back to Dashboard](#)
-  [Status](#)
-  [Changes](#)
-  [Build Now](#)
-  [Delete Pipeline](#)
-  [Configure](#)
-  [Move](#)
-  [Full Stage View](#)
-  [Pipeline Syntax](#)

- Click **#1** under "Build History" and then click **Console Output** to see the full output from the Pipeline.

## Console Output

```
Started by user admin
[Pipeline] node
Running on master in /var/jenkins_home/workspace/an-example
[Pipeline] {
[Pipeline] echo
Hello World
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

The example above shows a successful run of a basic Pipeline created in the Jenkins web UI, using two steps.

```
// Script //
node { ❶
    echo 'Hello World' ❷
}
// Declarative not yet implemented //
```

❶ **node** allocates an executor and workspace in the Jenkins environment.

❷ **echo** writes simple string in the Console Output.

## Defining a Pipeline in SCM

Complex Pipelines are hard to write and maintain within the text area of the Pipeline configuration page. To make this easier, Pipeline can also be written in a text editor and checked into source control as a **Jenkinsfile** which Jenkins can load via the **Pipeline Script from SCM** option.

To do this, select **Pipeline script from SCM** when defining the Pipeline.

With the **Pipeline script from SCM** option selected, you do not enter any Groovy code in the Jenkins UI; you just indicate by specifying a path where in source code you want to retrieve the pipeline from. When you update the designated repository, a new build is triggered, as long as the Pipeline is configured with an SCM polling trigger.

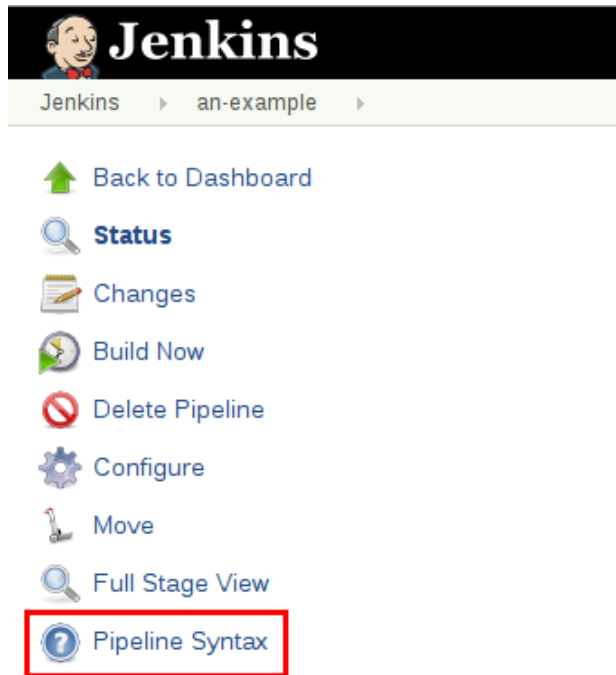
### TIP

The first line of a **Jenkinsfile** should be `#!/usr/bin/env groovy` [12: [en.wikipedia.org/wiki/Hashbang](https://en.wikipedia.org/wiki/Hashbang)] [13: [groovy-lang.org/syntax.html#\\_shebang\\_line](https://groovy-lang.org/syntax.html#_shebang_line)] which text editors, IDEs, GitHub, etc will use to syntax highlight the **Jenkinsfile** properly as Groovy code.

# Built-in Documentation

Pipeline ships with built-in documentation features to make it easier to create Pipelines of varying complexities. This built-in documentation is automatically generated and updated based on the plugins installed in the Jenkins instance.

The built-in documentation can be found globally at: [localhost:8080/pipeline-syntax/](http://localhost:8080/pipeline-syntax/), assuming you have a Jenkins instance running on localhost port 8080. The same documentation is also linked as **Pipeline Syntax** in the side-bar for any configured Pipeline project.



## Snippet Generator

The built-in "Snippet Generator" utility is helpful for creating bits of code for individual steps, discovering new steps provided by plugins, or experimenting with different parameters for a particular step.

The Snippet Generator is dynamically populated with a list of the steps available to the Jenkins instance. The number of steps available is dependent on the plugins installed which explicitly expose steps for use in Pipeline.

To generate a step snippet with the Snippet Generator:

1. Navigate to the **Pipeline Syntax** link (referenced above) from a configured Pipeline, or at [localhost:8080/pipeline-syntax/](http://localhost:8080/pipeline-syntax/).
2. Select the desired step in the **Sample Step** dropdown menu
3. Use the dynamically populated area below the **Sample Step** dropdown to configure the selected step.
4. Click **Generate Pipeline Script** to create a snippet of Pipeline which can be copied and pasted

into a Pipeline.

#### Steps

Sample Step stage: Stage ▼

Stage Name Deploy 

Generate Pipeline Script

```
stage('Deploy') {  
    // some block  
}
```

To access additional information and/or documentation about the step selected, click on the help icon (indicated by the red arrow in the image above).

## Global Variable Reference

In addition to the Snippet Generator, which only surfaces steps, Pipeline also provides a built-in "**Global Variable Reference**." Like the Snippet Generator, it is also dynamically populated by plugins. Unlike the Snippet Generator however, the Global Variable Reference only contains documentation for **variables** provided by Pipeline or plugins, which are available for Pipelines.

The variables provided by default in Pipeline are:

#### env

Environment variables accessible from Scripted Pipeline, for example: `env.PATH` or `env.BUILD_ID`. Consult the built-in [Global Variable Reference](#) for a complete, and up to date, list of environment variables available in Pipeline.

#### params

Exposes all parameters defined for the Pipeline as a read-only [Map](#), for example: `params.MY_PARAM_NAME`.

#### currentBuild

May be used to discover information about the currently executing Pipeline, with properties such as `currentBuild.result`, `currentBuild.displayName`, etc. Consult the built-in [Global Variable Reference](#) for a complete, and up to date, list of properties available on `currentBuild`.

# Further Reading

This section merely scratches the surface of what can be done with Jenkins Pipeline, but should provide enough of a foundation for you to start experimenting with a test Jenkins instance.

In the next section, [The Jenkinsfile](#), more Pipeline steps will be discussed along with patterns for implementing successful, real-world, Jenkins Pipelines.

## Additional Resources

- [Pipeline Steps Reference](#), encompassing all steps provided by plugins distributed in the Jenkins Update Center.
  - [Pipeline Examples](#), a community-curated collection of copyable Pipeline examples.
-

# Using a Jenkinsfile

This section builds on the information covered in [Getting Started](#), and introduces more useful steps, common patterns, and demonstrates some non-trivial **Jenkinsfile** examples.

Creating a **Jenkinsfile**, which is checked into source control [14: [en.wikipedia.org/wiki/Source\\_control\\_management](https://en.wikipedia.org/wiki/Source_control_management)], provides a number of immediate benefits:

- Code review/iteration on the Pipeline
- Audit trail for the Pipeline
- Single source of truth [15: [en.wikipedia.org/wiki/Single\\_Source\\_of\\_Truth](https://en.wikipedia.org/wiki/Single_Source_of_Truth)] 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 [Getting Started](#) section, 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.

```
// Declarative //
pipeline {
    agent any

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

// Script //
node {
    stage('Build') {
        echo 'Building....'
    }
    stage('Test') {
        echo 'Building....'
    }
    stage('Deploy') {
        echo 'Deploying....'
    }
}
```

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.

## NOTE

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

```
// Script //
node {
    checkout scm ①
    /* .. snip .. */
}
// Declarative not yet implemented //
```

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.

```
// Declarative //
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'make' ❶
                archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true ❷
            }
        }
    }
}

// Script //
node {
    stage('Build') {
        sh 'make' ❶
        archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true ❷
    }
}
```

- ❶ 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.
- ❷ `archiveArtifacts` captures the files built matching the include pattern (`*/target/*.jar`) and saves them to the Jenkins master for later retrieval.

#### TIP

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 `plugin:junit` [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.

```
// Declarative //
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
            }
        }
    }
}

// Script //
node {
    /* .. snip .. */
    stage('Test') {
        /* `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
    }
    /* .. snip .. */
}
```

① Using an inline shell conditional (`sh 'make || 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 [\[handling-failures\]](#) 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.

```
// Declarative //
pipeline {
    agent any

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

// Script //
node {
    /* .. snip .. */
    stage('Deploy') {
        if (currentBuild.result == null || currentBuild.result == 'SUCCESS') { ①
            sh 'make publish'
        }
    }
    /* .. snip .. */
}
```

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.

# Advanced Syntax for Pipeline

## 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.

## Working with the Environment

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 [localhost:8080/pipeline-syntax/globals#env](#), assuming a Jenkins master is running on `localhost:8080`, and includes:

### **BUILD\_ID**

The current build ID, identical to BUILD\_NUMBER for builds created in Jenkins versions 1.597+

### **JOB\_NAME**

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

### **JENKINS\_URL**

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

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