

Jenkins User Handbook

jenkinsci-docs@googlegroups.com

Table of Contents

Getting Started with Jenkins	1
Installing Jenkins	2
Overview	3
Pre-install.....	4
System Requirements.....	4
Experimentation, Staging, or Production?	4
Stand-alone or Servlet?	4
Installation	5
Unix/Linux	5
OS X.....	5
Windows	6
Docker	6
Other	6
Post-install (Setup Wizard)	7
Create Admin User and Password for Jenkins	7
Initial Plugin Installation	7
Using Jenkins.....	8
Managing Jenkins.....	9
Managing Plugins.....	10
System Configuration	11
Managing Security.....	12
Managing Nodes.....	13
Managing Tools	14
Best Practices	15
Pipeline.....	16
What is Pipeline?.....	17
Why Pipeline?	18
Pipeline Terms	19
Getting Started	20
Prerequisites.....	21
Defining a Pipeline	22
Defining a Pipeline in the Web UI.....	22
Defining a Pipeline in SCM	25
Built-in Documentation	26
Snippet Generator.....	26
Global Variable Reference	27
Further Reading.....	28
Additional Resources	28

The Jenkinsfile	29
Creating a Jenkinsfile	30
Build	31
Test	31
Deploy	32
Advanced Syntax for Scripted Pipeline	34
String Interpolation	34
Working with the Environment	34
Build Parameters.....	35
Handling Failures	35
Using multiple nodes	36
Executing in parallel	37
Optional step arguments.....	38
Multibranch Pipelines.....	40
Creating a Multibranch Pipeline	41
Additional Environment Variables.....	44
Supporting Pull Requests	44
Using Organization Folders.....	45
Shared Libraries.....	46
Defining Shared Libraries	47
Directory structure.....	47
Global Shared Libraries.....	48
Folder-level Shared Libraries	48
Automatic Shared Libraries	48
Using libraries	49
Overriding versions	49
Writing libraries	50
Accessing steps	50
Defining global variables	51
Defining steps	52
Defining a more structured DSL	53
Using third-party libraries	54
Loading resources.....	54
Pretesting library changes	54
Jenkins Use-Cases.....	56
Jenkins with .NET.....	57
Jenkins with Java	58
Jenkins with Python	59
Test Reports	60
Jenkins with Ruby	61
Test Reports	62

Coverage Reports	63
Rails	64
Operating Jenkins	65
Backing-up/Restoring Jenkins	66
Monitoring Jenkins	67
Securing Jenkins	68
Access Control	69
Protect users of Jenkins from other threats	70
Disabling Security	71
Managing Jenkins with Chef	72
Managing Jenkins with Puppet	73
Scaling Jenkins	74
Appendix	75
Advanced Jenkins Installation	76
Glossary	77
General Terms	78

Getting Started with Jenkins

This chapter is intended for new users unfamiliar with Jenkins or those without experience with recent versions of Jenkins.

This chapter will lead you through installing an instance of Jenkins on a system for learning purposes and understanding basic Jenkins concepts. It will provide simple step-by-step tutorials on how to do a number common tasks. Each section is intended to be completed in order, with each building on knowledge from the previous section. When you are done you should have enough experience with the core of Jenkins to continue exploring on your own.

If you are already familiar with Jenkins basics and would like to delve deeper into how to use specific features, see [Using Jenkins](#).

If you are a Jenkins administrator and want to know more about managing Jenkins nodes and instances, see [Managing Jenkins](#).

If you are a system administrator and want learn how to back-up, restore, maintain as Jenkins servers and nodes, see [Operating Jenkins](#).

Installing Jenkins

NOTE | This is still very much a work in progress

IMPORTANT | This section is part of *Getting Started*. It provides instructions for **basic** Jenkins configuration on a number of platforms. It **DOES NOT** cover the full range of considerations or options for installing Jenkins. See [Advanced Jenkins Installation](#)

Overview

Pre-install

System Requirements

WARNING

These are **starting points**. For a full discussion of factors see [Discussion of hardware recommendations](#).

Minimum Recommended Configuration:

- Java 7
- 256MB free memory
- 1GB+ free disk space

Recommended Configuration for Small Team:

- Java 8
- 1GB+ free memory
- 50GB+ free disk space

Experimentation, Staging, or Production?

How you configure Jenkins will differ significantly depending on your intended use cases. This section is specifically targeted to initial use and experimentation. For other scenarios, see [Advanced Jenkins Installation](#).

Stand-alone or Servlet?

Jenkins can run stand-alone in it's own process using it's own web server. It can also run as one servlet in an existing framework, such as Tomcat. This section is specifically targeted to stand-alone install and execution. For other scenarios, see [Advanced Jenkins Installation](#)

Installation

WARNING

These are **clean install** instructions for **non-production** environments. If you have a **non-production** Jenkins server already running on a system and want to upgrade, see [Upgrading Jenkins](#). If you are installing or upgrading a production Jenkins server, see [Advanced Jenkins Installation](#).

Unix/Linux

Debian/Ubuntu

On Debian-based distributions, such as Ubuntu, you can install Jenkins through **apt**.

Recent versions are available in [an apt repository](#). Older but stable LTS versions are in [this apt repository](#).

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

This package installation will:

- Setup Jenkins as a daemon launched on start. See `/etc/init.d/jenkins` for more details.
- Create a 'jenkins' user to run this service.
- Direct console log output to the file `/var/log/jenkins/jenkins.log`. Check this file if you are troubleshooting Jenkins.
- Populate `/etc/default/jenkins` with configuration parameters for the launch, e.g `JENKINS_HOME`
- Set Jenkins to listen on port 8080. Access this port with your browser to start configuration.

NOTE

If your `/etc/init.d/jenkins` file fails to start Jenkins, edit the `/etc/default/jenkins` to replace the line `----HTTP_PORT=8080----` with `----HTTP_PORT=8081----` Here, "8081" was chosen but you can put another port available.

OS X

To install from the website, using a package:

- [Download the latest package](#)
- Open the package and follow the instructions

Jenkins can also be installed using **brew**:

- Install the latest release version

```
brew install jenkins
```

- Install the LTS version

```
brew install jenkins-lts
```

Windows

To install from the website, using the installer:

- [Download the latest package](#)
- Open the package and follow the instructions

Docker

You must have [Docker](#) properly installed on your machine. See the [Docker installation guide](#) for details.

First, pull the official [jenkins](#) image from Docker repository.

```
docker pull jenkins
```

Next, run a container using this image and map data directory from the container to the host; e.g in the example below `/var/jenkins_home` from the container is mapped to `jenkins/` directory from the current path on the host. Jenkins `8080` port is also exposed to the host as `49001`.

```
docker run -d -p 49001:8080 -v $PWD/jenkins:/var/jenkins_home -t jenkins
```

Other

See [Advanced Jenkins Installation](#)

Post-install (Setup Wizard)

Create Admin User and Password for Jenkins

Jenkins is initially configured to be secure on first launch. Jenkins can no longer be accessed without a username and password and open ports are limited. During the initial run of Jenkins a security token is generated and printed in the console log:

```
*****
```

```
Jenkins initial setup is required. A security token is required to proceed.  
Please use the following security token to proceed to installation:
```

```
41d2b60b0e4cb5bf2025d33b21cb
```

```
*****
```

The install instructions for each of the platforms above includes the default location for when you can find this log output. This token must be entered in the "Setup Wizard" the first time you open the Jenkins UI. This token will also serve as the default password for the user 'admin' if you skip the user-creation step in the Setup Wizard.

Initial Plugin Installation

The Setup Wizard will also install the initial plugins for this Jenkins server. The recommended set of plugins available are based on the most common use cases. You are free to add more during the Setup Wizard or install them later as needed.

Using Jenkins

This chapter will describe how to work with Jenkins as a non-administrator user. It will cover topics applicable to anyone using Jenkins on a day-to-day basis. This includes basic topics such as selecting, running, and monitoring existing jobs, and how to find and review jobs results. It will continue on to discussing a number of topics around designing and creating projects.

This chapter is intended to be used by Jenkins users of all skill levels. The sections are structured in a feature-centric way for easier searching and reference by experienced users. At the same time, to help beginners, we've attempted to order sections in the chapter from simpler to progressively more complex feature areas. Also, topics within each section will progress from basic to advanced, with expert-level considerations and corner-cases at the end or in a separate section later in the chapter.

If you are not yet familiar with basic Jenkins terminology and features, start with [Getting Started with Jenkins](#).

If you are a Jenkins administrator and want to know more about managing Jenkins nodes and instances, see [Managing Jenkins](#).

If you are a system administrator and want learn how to back-up, restore, maintain as Jenkins servers and nodes, see [Operating Jenkins](#).

WARNING

To Contributors: This chapter functions as a continuation of "[Getting Started with Jenkins](#)", but the format will be slightly different - see the description above. We need to balance between providing a feature reference for experienced users with providing a continuing on-ramp for beginners. Sections should be written and ordered to only assume knowledge from "Getting Started" or from previous sections in this chapter.

Managing Jenkins

This chapter cover how to manage and configure Jenkins masters and nodes.

This chapter is intended for Jenkins administrators. More experienced users may find this information useful, but only to the extent that they will understand what is and is not possible for administrators to do. Individual sections may assume knowledge of information from previous sections, but such assumptions will be explicitly called out and cross-referenced.

If you are not yet familiar with basic Jenkins terminology and features, start with [Getting Started with Jenkins](#).

If you are already familiar with Jenkins basics and would like to delve deeper into how to use specific features, see [Using Jenkins](#).

If you are a system administrator and want learn how to back-up, restore, maintain as Jenkins servers and nodes, see [Operating Jenkins](#).

Managing Plugins

NOTE | This is still very much a work in progress

System Configuration

NOTE | This is still very much a work in progress

Managing Security

NOTE | This is still very much a work in progress

Managing Nodes

NOTE | This is still very much a work in progress

Managing Tools

NOTE | This is still very much a work in progress

Best Practices

This chapter discusses best practices for various areas in Jenkins. Each section addresses a specific area, plugin, or feature, and should be understandable independently of other sections.

This chapter is intended for experienced users and administrators. Some sections and the content in parts of sections may not apply to all users. In those cases, the content will indicate the intended audience.

If you are not yet familiar with basic Jenkins terminology and features, start with [Getting Started with Jenkins](#).

Pipeline

This chapter will cover all aspects of Jenkins Pipeline, from running pipeline jobs to writing your own pipeline code, and even extending Pipeline.

This chapter is intended to be used by Jenkins users of all skill levels, but beginners may need to refer to some sections of "[Using Jenkins](#)" to understand some topics covered in this chapter.

If you are not yet familiar with basic Jenkins terminology and features, start with [Getting Started with Jenkins](#).

What is Pipeline?

Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline DSL. [1: [Domain-Specific Language](#)]

Typically, this "Pipeline as Code" would be written to a **Jenkinsfile** and checked into a project's source control repository, for example:

```
// Script //
node { ①
    stage('Build') { ②
        sh 'make' ③
    }

    stage('Test') {
        sh 'make check'
        junit 'reports/**/*.xml' ④
    }

    stage('Deploy') {
        sh 'make publish'
    }
}

// Declarative not yet implemented //
```

- ① **node** indicates that Jenkins should allocate an executor and workspace for this part of the Pipeline.
- ② **stage** describes a stage of this Pipeline.
- ③ **sh** executes the given shell command
- ④ **junit** is a Pipeline [step](#) provided by the [JUnit plugin](#) for aggregating test reports.

Why Pipeline?

Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive continuous delivery pipelines. By modeling a series of related tasks, users can take advantage of the many features of Pipeline:

- **Code:** Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
- **Durable:** Pipelines can survive both planned and unplanned restarts of the Jenkins master.
- **Pausable:** Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
- **Versatile:** Pipelines support complex real-world continuous delivery requirements, including the ability to fork/join, loop, and perform work in parallel.
- **Extensible:** The Pipeline plugin supports custom extensions to its DSL [1: [Domain-Specific Language](#)] and multiple options for integration with other plugins.

While Jenkins has always allowed rudimentary forms of chaining Freestyle Jobs together to perform sequential tasks, [2: Additional plugins have been used to implement complex behaviors utilizing Freestyle Jobs such as the Copy Artifact, Parameterized Trigger, and Promoted Builds plugins] Pipeline makes this concept a first-class citizen in Jenkins.

Building on the core Jenkins value of extensibility, Pipeline is also extensible both by users with [Pipeline Shared Libraries](#) and by plugin developers. [3: [GitHub Organization Folder plugin](#)]

The flowchart below is an example of one continuous delivery scenario easily modeled in Jenkins Pipeline:

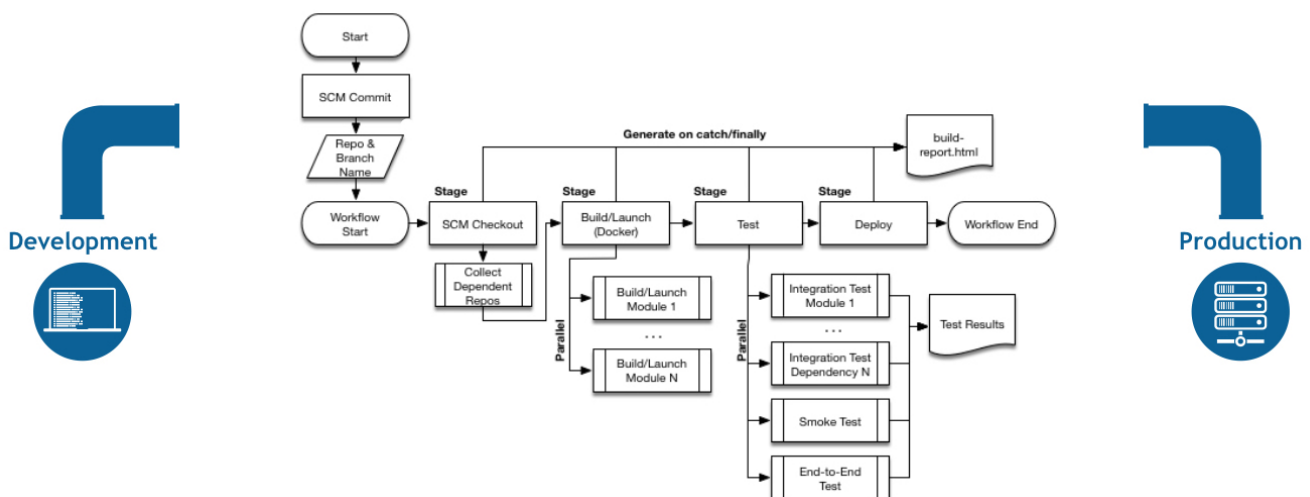


Figure 1. Pipeline Flow

Pipeline Terms

Step

A single task; fundamentally steps tell Jenkins *what* to do. For example, to execute the shell command `make` use the `sh` step: `sh 'make'`. When a plugin extends the Pipeline DSL, that typically means the plugin has implemented a new *step*.

Node

Most *work* a Pipeline performs is done in the context of one or more declared `node` steps. Confining the work inside of a node step does two things:

1. Schedules the steps contained within the block to run by adding an item to the Jenkins queue. As soon as an executor is free on a node, the steps will run.
2. Creates a workspace (a directory specific to that particular Pipeline) where work can be done on files checked out from source control.

CAUTION

Depending on your Jenkins configuration, some workspaces may not get automatically cleaned up after a period of inactivity. See tickets and discussion linked from [JENKINS-2111](#) for more information.

Stage

`stage` is a step for defining a conceptually distinct subset of the entire Pipeline, for example: "Build", "Test", and "Deploy", which is tused by many. plugins to visualize or present Jenkins Pipeline status/progress. [4: [Blue Ocean](#), [Pipeline Stage View plugin](#)]

Getting Started

Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline DSL. [5: [Domain-Specific Language](#)]

This section introduces some of the key concepts to Jenkins Pipeline and help introduce the basics of defining and working with Pipelines inside of a running Jenkins instance.

Prerequisites

To use Jenkins Pipeline, you will need:

- Jenkins 2.x or later (older versions back to 1.642.3 may work but are not recommended)
- Pipeline plugin [6: [Pipeline plugin](#)]

To learn how to install and manage plugins, consult [Managing Plugins](#).

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. [7: 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**.

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 your job is configured with an SCM polling trigger.

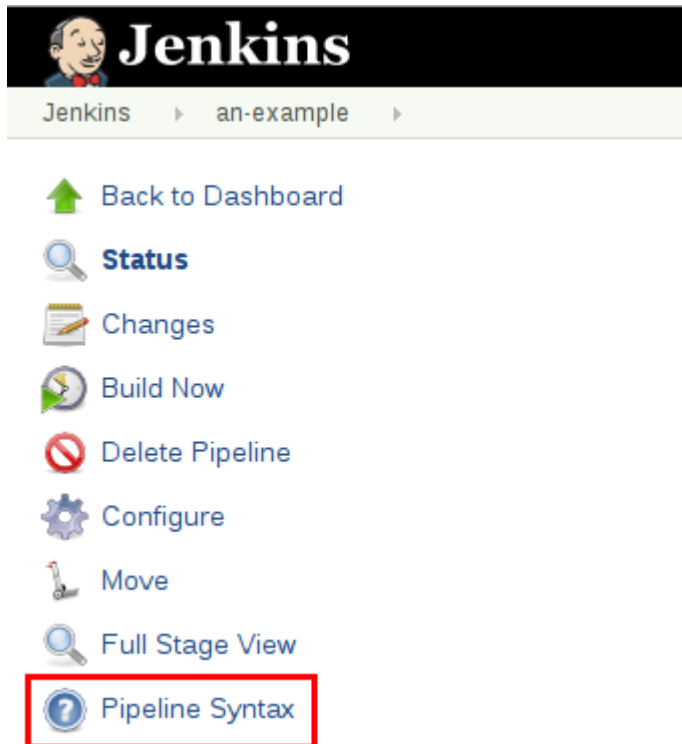
TIP

The first line of a **Jenkinsfile** should be **#!/groovy** [9: [en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))] 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/, 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/.
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

[?](#)

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.

The 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 [10: 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 [11: en.wikipedia.org/wiki/Single_Source_of_Truth] for the Pipeline, which can be viewed and edited by multiple members of the project.

While the syntax for defining a Pipeline, either in the web UI or with a **Jenkinsfile**, is the same, it's generally considered best practice to define the Pipeline in a **Jenkinsfile** and check that in to source control.

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.

```
// Script //
node { ❶
    stage('Build') { ❷
        /* .. snip .. */
    }
    stage('Test') {
        /* .. snip .. */
    }
    stage('Deploy') {
        /* .. snip .. */
    }
}
// Declarative not yet implemented //
```

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

❷ **stage** describes distinct parts of the Pipeline for better visualization of progress/status.

Not all Pipelines will have these same three stages, but this is a good continuous delivery 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.

In 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 ❶
    /* .. snip .. */
}
// Declarative not yet implemented //
```

❶ 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.

```
// Script //
node {
    /* .. snip .. */
    stage('Build') {
        sh 'make' ①
        archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true ②
    }
    /* .. snip .. */
}
// Declarative not yet implemented //
```

- 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.

CAUTION

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.

```
// 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 .. */
}
// Declarative not yet implemented //
```

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.

```
// Script //
node {
    /* .. snip .. */
    stage('Deploy') {
        if (currentBuild.result == 'SUCCESS') { ①
            sh 'make publish'
        }
    }
    /* .. snip .. */
}
// Declarative not yet implemented //
```

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 Scripted Pipeline

Scripted Pipeline is a domain-specific language [12: en.wikipedia.org/wiki/Domain-specific_language] based on Groovy, most [Groovy syntax](#) can be used in Scripted Pipeline without modification.

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 Groovy's string interpolation is vital for using some of Scripted 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:

```
// Script //
node {
    echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
}
// Declarative not yet implemented //
```

Setting environment variables

Setting an environment variable within a Jenkins Pipeline can be done with the `withEnv` step, which allows overriding specified environment variables for a given block of Scripted Pipeline, for example:

```
// Script //
node {
    /* .. snip .. */
    withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {
        sh 'mvn -B verify'
    }
}
// Declarative not yet implemented //
```

Build Parameters

If you configured your pipeline to accept parameters using the **Build with Parameters** option, those parameters are accessible as Groovy variables of the same name.

Assuming that a String parameter named "Greeting" has been configured for the Pipeline project in the web UI, a `Jenkinsfile` can access that parameter via `$Greeting`:

```
// Script //
node {
    echo "${Greeting} World!"
}
// Declarative not yet implemented //
```

Handling Failures

Scripted Pipeline 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:

```
// Script //
node {
    /* .. snip .. */
    stage('Test') {
        try {
            sh 'make check'
        }
        finally {
            junit '**/target/*.xml'
        }
    }
    /* .. snip .. */
}
// Declarative not yet implemented //
```

Using multiple nodes

In all previous uses of the `node` step, it has been used without any arguments. This means Jenkins will allocate an executor wherever one is available. The `node` step can take an optional "label" parameter, which is 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 node and the built results will be reused on two different nodes, labelled "linux" and "windows" respectively, during the "Test" stage.


```
// Script //
stage('Build') {
    node {
        checkout scm
        sh 'make'
        stash includes: '**/target/*.jar', name: 'app' ❶
    }
}

stage('Test') {
    node('linux') { ❷
        checkout scm
        try {
            unstash 'app' ❸
            sh 'make check'
        }
        finally {
            junit '**/target/*.xml'
        }
    }
    node('windows') {
        checkout scm
        try {
            unstash 'app'
            bat 'make check' ❹
        }
        finally {
            junit '**/target/*.xml'
        }
    }
}

// Declarative not yet implemented //
```

- ❶ 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.
- ❷ The optional parameter to **node** allows for any valid Jenkins label expression. Consult the inline help for **node** in the [Snippet Generator](#) for more details.
- ❸ **unstash** will retrieve the named "stash" from the Jenkins master into the Pipeline's current workspace.
- ❹ The **bat** script allows for executing batch scripts on Windows-based platforms.

Executing in parallel

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:

```
// Script //
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 .. */
        }
    }
}
// Declarative not yet implemented //
```

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.

Optional step arguments

Groovy allows parentheses around function arguments to be omitted.

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 */
```

Multibranch Pipelines

In the [previous section](#) a **Jenkinsfile** which could be checked into source control was implemented. This section covers the concept of **Multibranch** Pipelines which build on the **Jenkinsfile** foundation to provide more dynamic and automatic functionality in Jenkins.

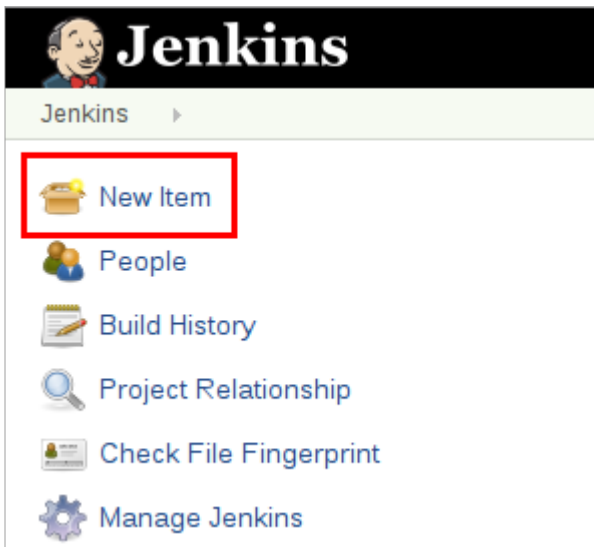
Creating a Multibranch Pipeline

The **Multibranch Pipeline** project type enables you to implement different Jenkinsfiles for different branches of the same project. In a Multibranch Pipeline project, Jenkins automatically discovers, manages and executes Pipelines for branches which contain a **Jenkinsfile** in source control.

This eliminates the need for manual Pipeline creation and management.

To create a Multibranch Pipeline:

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



- Enter a name for your Pipeline, select **Multibranch 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

- Add a **Branch Source** (for example, Git) and enter the location of the repository.

Name

Display Name 


Description

[Plain text] [Preview](#)

Branch Sources

Add source ▼

- Git**
- GitHub
- Single repository & branch
- Subversion


☐ Trigger builds remotely (e.g., from scripts) 

Git

Project Repository

Credentials


- none - ▼

 **Add** ▼

Ignore on push notifications ☐

Repository browser

(Auto) ▼



Additional Behaviours

Add ▼

Advanced...

Property strategy

All branches get the same properties ▼

Add property ▼

Delete source

- **Save** the Multibranch Pipeline project.

Upon **Save**, Jenkins automatically scans the designated repository and creates appropriate items for each branch in the repository which contains a **Jenkinsfile**.

By default, Jenkins will not automatically re-index the repository for branch additions or deletions (unless using an [Organization Folder](#)), so it is often useful to configure a Multibranch Pipeline to periodically re-index in the configuration:

Build Triggers

A screenshot of the Jenkins 'Build Triggers' configuration section. It contains three checkboxes: 'Trigger builds remotely (e.g., from scripts)', 'Build periodically', and 'Periodically if not otherwise run'. The third checkbox is checked and highlighted with a red rectangle. Below it, there is an 'Interval' field with a dropdown menu showing '30 minutes'. Each option has a help icon (question mark in a circle) to its right.

☐ Trigger builds remotely (e.g., from scripts) ?

☐ Build periodically ?

☒ Periodically if not otherwise run ?

Interval 30 minutes ▼ ?

Additional Environment Variables

Multibranch Pipelines expose additional information about the branch being built through the `env` global variable, such as:

`BRANCH_NAME`

Name of the branch for which this Pipeline is executing, for example `master`.

`CHANGE_ID`

An identifier corresponding to some kind of change request, such as a pull request number

Additional environment variables are listed in the [Global Variable Reference](#).

Supporting Pull Requests

With the "GitHub" or "Bitbucket" Branch Sources, Multibranch Pipelines can be used for validating pull/change requests. This functionality is provided, respectively, by the [GitHub Branch Source](#) and [Bitbucket Branch Source](#) plugins. Please consult their documentation for further information on how to use those plugins.

Using Organization Folders

Organization Folders enable Jenkins to monitor an entire GitHub Organization, or Bitbucket Team/Project and automatically create new Multibranch Pipelines for repositories which contain branches and pull requests containing a **Jenkinsfile**.

Currently, this functionality exists only for GitHub and Bitbucket, with functionality provided by the [GitHub Organization Folder](#) and [Bitbucket Branch Source](#) plugins.

Shared Libraries

As 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 and keep code "DRY" [13: en.wikipedia.org/wiki/Don't_repeat_yourself].

Pipeline has support for creating "Shared Libraries" which can be defined in external source control repositories and loaded into existing Pipelines.

Defining Shared Libraries

An 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.

The version could be anything understood by that SCM; for example, branches, tags, and commit hashes all work for Git. You may also declare whether scripts need to explicitly request that library (detailed below), or if it is present by default. Furthermore, if you specify a version in Jenkins configuration, you can block scripts from selecting a *different* version.

The best way to specify the SCM is using an SCM plugin which has been specifically updated to support a new API for checking out an arbitrary named version (*Modern SCM* option). As of this writing, the latest versions of the Git and Subversion plugins support this mode; others should follow.

If your SCM plugin has not been integrated, you may select *Legacy SCM* and pick anything offered. In this case, you need to include `${library.yourLibName.version}` somewhere in the configuration of the SCM, so that during checkout the plugin will expand this variable to select the desired version. For example, for Subversion, you can set the *Repository URL* to `svnserver/project/${library.yourLibName.version}` and then use versions such as `trunk` or `branches/dev` or `tags/1.0`.

Directory structure

The directory structure of a Shared Library repository is as follows:

```
(root)
+- src                      # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy  # for org.foo.Bar class
+- vars
|   +- foo.groovy          # for global 'foo' variable
|   +- foo.txt             # help for 'foo' variable
+- resources               # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json    # static helper data for org.foo.Bar
```

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 scripts that define global variables accessible from Pipeline. The basename of each `.groovy` file should be a Groovy (~ Java) identifier, conventionally camelCased. The matching `.txt`, if present, can contain documentation, processed through the system's configured markup formatter (so may really be HTML, Markdown, etc., though the `txt` extension is required).

The Groovy source files in these directories get the same “CPS transformation” as in Scripted

Pipeline.

A **resources** directory allows the **libraryResource** step to be used from an external library to load associated non-Groovy files. Currently this feature is not supported for internal libraries.

Other directories under the root are reserved for future enhancements.

Global Shared Libraries

There are several places where Shared Libraries can be defined, depending on the use-case. *Manage Jenkins » Configure System » Global Pipeline Libraries* as many libraries as necessary can be configured.

Since these libraries will be globally usable, any Pipeline in the system can utilize functionality implemented in these libraries.

These libraries are considered "trusted:" they can run any methods in Java, Groovy, Jenkins internal APIs, Jenkins plugins, or third-party libraries. This allows you to define libraries which encapsulate individually unsafe APIs in a higher-level wrapper safe for use from any job. Beware that **anyone able to push commits to this SCM repository could obtain unlimited access to Jenkins**. You need the *Overall/RunScripts* permission to configure these libraries (normally this will be granted to Jenkins administrators).

Folder-level Shared Libraries

Any Folder created can have Shared Libraries associated with it. This mechanism allows scoping of specific libraries to all the Pipelines inside of the folder or subfolder.

Folder-based libraries are not considered "trusted:" they run in the Groovy sandbox just like typical Pipelines.

Automatic Shared Libraries

Other plugins may add ways of defining libraries on the fly. For example, the [GitHub Organization Folder](#) plugin allows a script to use an untrusted library such as **github.com/someorg/somerepo** without any additional configuration. In this case, the specified GitHub repository would be loaded, from the **master** branch, using an anonymous checkout.

Using libraries

Pipelines can access shared libraries marked *Load implicitly*. They may immediately use classes or global variables defined by any such libraries (details below).

To access other shared libraries, a script needs to use the `@Library` annotation, specifying the library's name:

```
@Library('somelib')
/* Using a version specifier, such as branch, tag, etc */
@Library('somelib@1.0')
/* Accessing multiple libraries with one statement */
@Library(['somelib', 'otherlib@abc1234'])
```

The annotation can be anywhere in the script where an annotation is permitted by Groovy. When referring to class libraries (with `src/` directories), conventionally the annotation goes on an `import` statement:

```
@Library('somelib')
import com.mycorp.pipeline.somelib.UsefulClass
```

NOTE

It is legal, though unnecessary, to `import` a global variable (or method) defined in the `vars/` directory:

Note that libraries are resolved and loaded during *compilation* of the script, before it starts executing. This allows the Groovy compiler to understand the meaning of symbols used in static type checking, and permits them to be used in type declarations in the script, for example:

```
@Library('somelib')
import com.mycorp.pipeline.somelib.Helper

int useSomeLib(Helper helper) {
    helper.prepare()
    return helper.count()
}

echo useSomeLib(new Helper('some text'))
```

Global Variables however, are resolved at runtime.

Overriding versions

A `@Library` annotation may override a default version given in the library's definition, if the definition permits this. In particular, a shared library marked for implicit use can still be loaded in a different version using the annotation (unless the definition specifically forbids this).

Writing libraries

At the base level, any valid [Groovy code](#) is okay for use. Different data structures, utility methods, etc, such as:

```
// src/org/foo/Point.groovy
package org.foo;

// point in 3D space
class Point {
    float x,y,z;
}
```

Accessing steps

Library classes cannot directly call steps such as [sh](#) or [git](#). They can however implement methods, outside of the scope of an enclosing class, which in turn invoke Pipeline steps, for example:

```
// src/org/foo/Zot.groovy
package org.foo;

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

Which can then be called from a Scripted Pipeline:

```
def z = new org.foo.Zot()
z.checkoutFrom(repo)
```

This approach has limitations; for example, it prevents the declaration of a superclass.

Alternately, a set of [steps](#) can be passed explicitly to a library class, in a constructor, or just one method:

```
package org.foo
class Utilities implements Serializable {
    def steps
    Utilities(steps) {this.steps = steps}
    def mvn(args) {
        steps.sh "${steps.tool 'Maven'}/bin/mvn -o ${args}"
    }
}
```

When saving state on classes, such as above, the class **must** implement the [Serializable](#) interface.

This ensures that a Pipeline using the class, as seen in the example below, can properly suspend and resume in Jenkins.

```
@Library('utils') import org.foo.Utilities
def utils = new Utilities(steps)
node {
    utils.mvn 'clean package'
}
```

If the library needs to access global variables, such as `env`, those should be explicitly passed into the library classes, or methods, in a similar manner.

Instead of passing numerous variables from the Scripted Pipeline into a library,

```
package org.foo
class Utilities {
    static def mvn(script, args) {
        script.sh "${script.tool 'Maven'}/bin/mvn -s ${script.env.HOME}/jenkins.xml -o
        ${args}"
    }
}
```

The above example shows the script being passed in to one `static` method, invoked from a Scripted Pipeline as follows:

```
@Library('utils') import static org.foo.Utilities.*
node {
    mvn this, 'clean package'
}
```

Defining global variables

Internally, scripts in the `vars` directory are instantiated on-demand as singletons. This allows multiple methods or properties to be defined in a single `.groovy` file which interact with each other, for example:

```
// vars/acme.groovy
def setName(value) {
    this.name = value;
}
def getName() {
    return this.name;
}
def caution(message) {
    echo "Hello, ${this.name}! CAUTION: ${message}"
}
```

The Pipeline can then invoke these methods which will be defined on the `acme` object:

```
acme.name = 'Alice'
echo acme.name /* prints: 'Alice' */
acme.caution 'The queen is angry!' /* prints: 'Hello, Alice. CAUTION: The queen is
angry!' */
```

NOTE

A variable defined in a shared library will only show up in *Global Variables Reference* (under *Pipeline Syntax*) after Jenkins loads and uses that library as part of a successful Pipeline run.

Defining steps

Shared Libraries can also define global variables which behave similarly to built-in steps, such as `sh` or `git`. Global variables defined in Shared Libraries **must** be named with all lower-case or "camelCased" in order to be loaded properly by Pipeline. [14: gist.github.com/rtyler/e5e57f075af381fce4ed3ae57aa1f0c2]

For example, to define `sayHello`, the file `vars/sayHello.groovy` should be created and should implement a `call` method. The `call` method allows the global variable to be invoked in a manner similar to a step:

```
// vars/sayHello.groovy
def call(String name = 'human') {
    // Any valid steps can be called from this code, just like in other
    // Scripted Pipeline
    echo "Hello, ${name}."
}
```

The Pipeline would then be able to reference and invoke this variable:

```
sayHello 'Joe'
sayHello() /* invoke with default arguments */
```


If called with a block, the `call` method will receive a `Closure`. The type should be defined explicitly to clarify the intent of the step, for example:

```
// vars/windows.groovy
def call(Closure body) {
    node('windows') {
        body()
    }
}
```

The Pipeline can then use this variable like any built-in step which accepts a block:

```
windows {
    bat "cmd /?"
}
```

Defining a more structured DSL

If you have a lot of Pipeline jobs that are mostly similar, the global variable mechanism gives you a handy tool to build a higher-level DSL that captures the similarity. For example, all Jenkins plugins are built and tested in the same way, so we might write a step named `buildPlugin`:

```
// vars/buildPlugin.groovy
def call(body) {
    // evaluate the body block, and collect configuration into the object
    def config = [:]
    body.resolveStrategy = Closure.DELEGATE_FIRST
    body.delegate = config
    body()

    // now build, based on the configuration provided
    node {
        git url: "https://github.com/jenkinsci/${config.name}-plugin.git"
        sh "mvn install"
        mail to: "...", subject: "${config.name} plugin build", body: "..."
    }
}
```

Assuming the script has either been loaded as a [Global Shared Library](#) or as a [Folder-level Shared Library](#) the resulting `Jenkinsfile` will be dramatically simpler:

```
// Script //
buildPlugin {
    name = 'git'
}
// Declarative not yet implemented //
```

Using third-party libraries

It is possible to use third-party Java libraries, typically found in [Maven Central](#), from **trusted** library code using the `@Grab` annotation. Refer to the [Grape documentation](#) for details, but simply put:

```
@Grab('org.apache.commons:commons-math3:3.4.1')
import org.apache.commons.math3.primes.Primes
void parallelize(int count) {
    if (!Primes.isPrime(count)) {
        error "${count} was not prime"
    }
    // ...
}
```

Third-party libraries are cached by default in `~/.groovy/grapes/` on the Jenkins master.

Loading resources

External libraries may load adjunct files from a `resources/` directory using the `libraryResource` step. The argument is a relative pathname, akin to Java resource loading:

```
def request = libraryResource 'com/mycorp/pipeline/somelib/request.json'
```

The file is loaded as a string, suitable for passing to certain APIs or saving to a workspace using `writeFile`.

It is advisable to use an unique package structure so you do not accidentally conflict with another library.

Pretesting library changes

If you notice a mistake in a build using an untrusted library, simply click the *Replay* link to try editing one or more of its source files, and see if the resulting build behaves as expected. Once you are satisfied with the result, follow the diff link from the build's status page, and apply the diff to the library repository and commit.

(Even if the version requested for the library was a branch, rather than a fixed version like a tag, replayed builds will use the exact same revision as the original build: library sources will not be

checked out again.)

Replay is not currently supported for trusted libraries. Modifying resource files is also not currently supported during *Replay*.

Jenkins Use-Cases

An alternate title for this chapter would be "Jenkins for X". Each section of this chapter focuses on how to use Jenkins in a specific area.

The audience for each section in this chapter is any user interested in working with Jenkins in that area. Sections are completely independent, knowledge from any one section is not required to understand any other.

If you are not yet familiar with basic Jenkins terminology and features, start with [Getting Started with Jenkins](#).

If you are already familiar with Jenkins basics and would like to delve deeper into generally how to use various features, see [Using Jenkins](#).

If you are a system administrator and want learn how to back-up, restore, maintain as Jenkins servers and nodes, see [Operating Jenkins](#).

WARNING

To Contributors: For this chapter, topics which apply to multiple areas and thus would repeated in multiple sections, should instead be covered in other chapters and cross-referenced. For example, a general discussion of how to deploy artifacts would go in the chapter "[Using Jenkins](#)", while language-specific examples of how to deploy artifacts in Java, Python, and Ruby, would go in the appropriate sections in this chapter.

Jenkins with .NET

NOTE | This is still very much a work in progress

Jenkins with Java

NOTE | This is still very much a work in progress

Jenkins with Python

NOTE | This is still very much a work in progress

Test Reports

Jenkins with Ruby

NOTE | This is still very much a work in progress

Test Reports

Coverage Reports

Rails

Operating Jenkins

This chapter is for system administrators of Jenkins servers and nodes. It will cover system maintenance topics including security, monitoring, and backup/restore.

Users not involved with system-level tasks will find this chapter of limited use. Individual sections may assume knowledge of information from previous sections, but such assumptions will be explicitly called out and cross-referenced.

If you are not yet familiar with basic Jenkins terminology and features, start with [Getting Started with Jenkins](#).

If you are already familiar with Jenkins basics and would like to delve deeper into how to use specific features, see [Using Jenkins](#).

If you are a Jenkins administrator and want to know more about managing Jenkins nodes and instances, see [Managing Jenkins](#).

Backing-up/Restoring Jenkins

NOTE | This is still very much a work in progress

Monitoring Jenkins

NOTE | This is still very much a work in progress

Securing Jenkins

NOTE | This is still very much a work in progress

In the default configuration of Jenkins 1.x, Jenkins does not perform any security checks. This means the ability of Jenkins to launch processes and access local files are available to anyone who can access Jenkins web UI and some more.

Securing Jenkins has two aspects to it.

- Access control, which ensures users are authenticated when accessing Jenkins and their activities are authorized.
- Protecting Jenkins against external threats

Access Control

You should lock down the access to Jenkins UI so that users are authenticated and appropriate set of permissions are given to them. This setting is controlled mainly by two axes:

- **Security Realm**, which determines users and their passwords, as well as what groups the users belong to.
- **Authorization Strategy**, which determines who has access to what.

These two axes are orthogonal, and need to be individually configured. For example, you might choose to use external LDAP or Active Directory as the security realm, and you might choose "everyone full access once logged in" mode for authorization strategy. Or you might choose to let Jenkins run its own user database, and perform access control based on the permission/user matrix.

- [Quick and Simple Security](#) --- if you are running Jenkins like `java -jar jenkins.war` and only need a very simple set up
- [Standard Security Setup](#) --- discusses the most common set up of letting Jenkins run its own user database and do finer-grained access control
- [Apache frontend for security](#) --- run Jenkins behind Apache and perform access control in Apache instead of Jenkins
- [Authenticating scripted clients](#) --- if you need to programmatically access security-enabled Jenkins web UI, use BASIC auth
- [Matrix-based security](#) | [Matrix-based security](#) --- Granting and denying finer-grained permissions

Protect users of Jenkins from other threats

There are additional security subsystems in Jenkins that protect Jenkins and users of Jenkins from indirect attacks.

The following topics discuss features that are **off by default**. We recommend you read them first and act on them.

- [CSRF Protection](#) --- prevent a remote attack against Jenkins running inside your firewall
- [Security implication of building on master](#) --- protect Jenkins master from malicious builds
- [Slave To Master Access Control](#) --- protect Jenkins master from malicious build agents

The following topics discuss other security features that are on by default. You'll only need to look at them when they are causing problems.

- [Configuring Content Security Policy](#) --- protect users of Jenkins from malicious builds
- [Markup formatting](#) --- protect users of Jenkins from malicious users of Jenkins

Disabling Security

One may accidentally set up security realm / authorization in such a way that you may no longer be able to reconfigure Jenkins.

When this happens, you can fix this by the following steps:

1. Stop Jenkins (the easiest way to do this is to stop the servlet container.)
2. Go to `$JENKINS_HOME` in the file system and find `config.xml` file.
3. Open this file in the editor.
4. Look for the `<useSecurity>true</useSecurity>` element in this file.
5. Replace `true` with `false`
6. Remove the elements `authorizationStrategy` and `securityRealm`
7. Start Jenkins
8. When Jenkins comes back, it will be in an unsecured mode where everyone gets full access to the system.

If this is still not working, try renaming or deleting `config.xml`.

Managing Jenkins with Chef

NOTE | This is still very much a work in progress

Managing Jenkins with Puppet

NOTE | This is still very much a work in progress

Scaling Jenkins

This chapter will cover topics related to using and managing large scale Jenkins configurations: large numbers of users, nodes, agents, folders, projects, concurrent jobs, job results and logs, and even large numbers of masters.

The audience for this chapter is expert Jenkins users, administrators, and those planning large-scale installations.

If you are not yet familiar with basic Jenkins terminology and features, start with [Getting Started with Jenkins](#).

If you are already familiar with Jenkins basics and would like to delve deeper into generally how to use various features, see [Using Jenkins](#).

If you are a Jenkins administrator and want to know more about managing Jenkins nodes and instances, see [Managing Jenkins](#).

Appendix

These sections are generally intended for Jenkins administrators and system administrators. Each section covers a different topic independent of the other sections. They are advanced topics, reference material, and topics that do not fit into other chapters.

WARNING

To Contributors: Please consider adding material elsewhere before adding it here. In fact, topics that do not fit elsewhere may even be out of scope for this handbook. See [\[Contributing to Jenkins\]](#) for details of how to contact project contributors and discuss what you want to add.

Advanced Jenkins Installation

NOTE | This is still very much a work in progress

Glossary

General Terms

Agent

An agent is typically a machine, or container, which connects to a Jenkins master and executes tasks when directed by the master.

Artifact

An immutable file generated during a [Build](#) or [Pipeline](#) run which is **archived** onto the Jenkins [Master](#) for later retrieval by users.

Build

Result of a single execution of a [Project](#)

Cloud

A System Configuration which provides dynamic [Agent](#) provisioning and allocation, such as that provided by the [Azure VM Agents](#) or [Amazon EC2](#) plugins.

Core

The primary Jenkins application ([jenkins.war](#)) which provides the basic web UI, configuration, and foundation upon which [Plugins](#) can be built.

Downstream

A configured [Pipeline](#) or [Project](#) which is triggered as part of the execution of a separate Pipeline or Project.

Executor

A slot for execution of work defined by a [Pipeline](#) or [Project](#) on a [Node](#). A Node may have zero or more Executors configured which corresponds to how many concurrent Projects or Pipelines are able to execute on that Node.

Fingerprint

A hash considered globally unique to track the usage of an [Artifact](#) or other entity across multiple [Pipelines](#) or [Projects](#).

Folder

An organizational container for [Pipelines](#) and/or [Projects](#), similar to folders on a file system.

Item

An entity in the web UI corresponding to either a: [Folder](#), [Pipeline](#), or [Project](#).

Job

A deprecated term, synonymous with [Project](#).

Master

The central, coordinating process which stores configuration, loads plugins, and renders the various user interfaces for Jenkins.

Node

A machine which is part of the Jenkins environment and capable of executing [Pipelines](#) or [Projects](#). Both the [Master](#) and [Agents](#) are considered to be Nodes.

Project

A user-configured description of work which Jenkins should perform, such as building a piece of software, etc.

Pipeline

A user-defined model of a continuous delivery pipeline, for more read the [Pipeline chapter](#) in this handbook.

Plugin

An extension to Jenkins functionality provided separately from Jenkins [Core](#).

Publisher

Part of a [Build](#) after the completion of all configured [Steps](#) which publishes reports, sends notifications, etc.

Step

A single task; fundamentally steps tell Jenkins *what* to do inside of a [Pipeline](#) or [Project](#).

Trigger

A criteria for triggering a new [Pipeline](#) run or [Build](#).

Upstream

A configured [Pipeline](#) or [Project](#) which triggers a separate Pipeline or Project as part of its execution.

Workspace

A disposable directory on the file system of a [Node](#) where work can be done by a [Pipeline](#) or [Project](#). Workspaces are typically left in place after a [Build](#) or [Pipeline](#) run completes unless specific Workspace cleanup policies have been put in place on the Jenkins [Master](#).