**Table of Contents**

[Getting Started with Jenkins](#_bookmark0) [1](#_bookmark0)

[Installing Jenkins](#_bookmark1) [2](#_bookmark1)

[Overview](#_bookmark2) [3](#_bookmark2)

[Pre-install](#_bookmark3) [4](#_bookmark3)

[System Requirements](#_bookmark4) [4](#_bookmark4)

[Experimentation, Staging, or Production?](#_bookmark5) [4](#_bookmark5)

[Stand-alone or Servlet?](#_bookmark6) [4](#_bookmark6)

[Installation](#_bookmark7) [5](#_bookmark7)

[Unix/Linux](#_bookmark8) [5](#_bookmark8)

[OS X](#_bookmark9) [5](#_bookmark9)

[Windows](#_bookmark10) [6](#_bookmark10)

[Docker](#_bookmark11) [6](#_bookmark11)

[Other](#_bookmark12) [6](#_bookmark12)

[Post-install (Setup Wizard)](#_bookmark13) [7](#_bookmark13)

[Create Admin User and Password for Jenkins](#_bookmark14) [7](#_bookmark14)

[Initial Plugin Installation](#_bookmark15) [7](#_bookmark15)

[Using Jenkins](#_bookmark16) [8](#_bookmark16)

[Fingerprints](#_bookmark17) [9](#_bookmark17)

[Remote API](#_bookmark18) [10](#_bookmark18)

[Security](#_bookmark19) [11](#_bookmark19)

[CSRF](#_bookmark20) [11](#_bookmark20)

[Managing Jenkins](#_bookmark21) [12](#_bookmark21)

[Configuring the System](#_bookmark22) [13](#_bookmark22)

[Managing Security](#_bookmark23) [14](#_bookmark23)

[Enabling Security](#_bookmark24) [15](#_bookmark24)

[JNLP TCP Port](#_bookmark25) [15](#_bookmark25)

[Access Control](#_bookmark26) [16](#_bookmark26)

[Markup Formatter](#_bookmark27) [18](#_bookmark27)

[Cross Site Request Forgery](#_bookmark28) [19](#_bookmark28)

[Caveats](#_bookmark29) [19](#_bookmark29)

[Agent/Master Access Control](#_bookmark30) [20](#_bookmark30)

[Customizing Access](#_bookmark31) [20](#_bookmark31)

[Disabling](#_bookmark32) [22](#_bookmark32)

[Managing Tools](#_bookmark33) [24](#_bookmark33)

[Built-in tool providers](#_bookmark34) [25](#_bookmark34)

[Ant](#_bookmark35) [25](#_bookmark35)

[Git](#_bookmark36) [25](#_bookmark36)

[JDK](#_bookmark37) [25](#_bookmark37)

[Maven](#_bookmark38) [25](#_bookmark38)

[Managing Plugins](#_bookmark39) [26](#_bookmark39)

[Installing a plugin](#_bookmark40) [27](#_bookmark40)

[From the web UI](#_bookmark41) [27](#_bookmark41)

[Using the Jenkins CLI](#_bookmark42) [28](#_bookmark42)

[Advanced installation](#_bookmark43) [28](#_bookmark43)

[Updating a plugin](#_bookmark44) [30](#_bookmark44)

[Removing a plugin](#_bookmark45) [31](#_bookmark45)

[Uninstalling a plugin](#_bookmark46) [31](#_bookmark46)

[Disabling a plugin](#_bookmark47) [32](#_bookmark47)

[Pinned plugins](#_bookmark48) [33](#_bookmark48)

[Jenkins CLI](#_bookmark49) [34](#_bookmark49)

[Using the CLI over SSH](#_bookmark50) [35](#_bookmark50)

[Authentication](#_bookmark51) [35](#_bookmark51)

[Common Commands](#_bookmark52) [36](#_bookmark52)

[Using the CLI client](#_bookmark53) [39](#_bookmark53)

[Downloading the client](#_bookmark54) [39](#_bookmark54)

[Using the client](#_bookmark55) [39](#_bookmark55)

[Client connection modes](#_bookmark56) [39](#_bookmark56)

[Script Console](#_bookmark57) [42](#_bookmark57)

[Managing Nodes](#_bookmark58) [43](#_bookmark58)

[Managing Users](#_bookmark59) [44](#_bookmark59)

[Best Practices](#_bookmark60) [45](#_bookmark60)

[Pipeline](#_bookmark61) [46](#_bookmark61)

[What is Pipeline?](#_bookmark135) [47](#_bookmark135)

[Why Pipeline?](#_bookmark62) [49](#_bookmark62)

[Pipeline Terms](#_bookmark63) [50](#_bookmark63)

[Getting Started with Pipeline](#_bookmark65) [51](#_bookmark65)

[Prerequisites](#_bookmark66) [52](#_bookmark66)

[Defining a Pipeline](#_bookmark67) [53](#_bookmark67)

[Defining a Pipeline in the Web UI](#_bookmark68) [53](#_bookmark68)

[Defining a Pipeline in SCM](#_bookmark69) [56](#_bookmark69)

[Built-in Documentation](#_bookmark70) [57](#_bookmark70)

[Snippet Generator](#_bookmark71) [57](#_bookmark71)

[Global Variable Reference](#_bookmark72) [58](#_bookmark72)

[Further Reading](#_bookmark73) [59](#_bookmark73)

[Additional Resources](#_bookmark74) [59](#_bookmark74)

[Using a Jenkinsfile](#_bookmark75) [60](#_bookmark75)

[Creating a Jenkinsfile](#_bookmark76) [61](#_bookmark76)

[Build](#_bookmark77) [62](#_bookmark77)

[Test](#_bookmark78) [63](#_bookmark78)

[Deploy](#_bookmark79) [64](#_bookmark79)

[Advanced Syntax for Pipeline](#_bookmark80) [66](#_bookmark80)

[String Interpolation](#_bookmark81) [66](#_bookmark81)

[Working with the Environment](#_bookmark82) [66](#_bookmark82)

[Parameters](#_bookmark83) [68](#_bookmark83)

[Handling Failures](#_bookmark84) [68](#_bookmark84)

[Using multiple agents](#_bookmark85) [69](#_bookmark85)

[Optional step arguments](#_bookmark86) [71](#_bookmark86)

[Advanced Scripted Pipeline](#_bookmark87) [72](#_bookmark87)

[Branches and Pull Requests](#_bookmark88) [74](#_bookmark88)

[Creating a Multibranch Pipeline](#_bookmark89) [75](#_bookmark89)

[Additional Environment Variables](#_bookmark90) [78](#_bookmark90)

[Supporting Pull Requests](#_bookmark91) [78](#_bookmark91)

[Using Organization Folders](#_bookmark92) [79](#_bookmark92)

[Extending with Shared Libraries](#_bookmark93) [80](#_bookmark93)

[Defining Shared Libraries](#_bookmark94) [81](#_bookmark94)

[Directory structure](#_bookmark95) [81](#_bookmark95)

[Global Shared Libraries](#_bookmark96) [82](#_bookmark96)

[Folder-level Shared Libraries](#_bookmark97) [82](#_bookmark97)

[Automatic Shared Libraries](#_bookmark98) [82](#_bookmark98)

[Using libraries](#_bookmark99) [83](#_bookmark99)

[Loading libraries dynamically](#_bookmark100) [84](#_bookmark100)

[Library versions](#_bookmark101) [85](#_bookmark101)

[Retrieval Method](#_bookmark102) [85](#_bookmark102)

[Writing libraries](#_bookmark103) [88](#_bookmark103)

[Accessing steps](#_bookmark104) [88](#_bookmark104)

[Defining global variables](#_bookmark105) [89](#_bookmark105)

[Defining steps](#_bookmark106) [90](#_bookmark106)

[Defining a more structured DSL](#_bookmark107) [91](#_bookmark107)

[Using third-party libraries](#_bookmark108) [92](#_bookmark108)

[Loading resources](#_bookmark109) [92](#_bookmark109)

[Pretesting library changes](#_bookmark110) [93](#_bookmark110)

[Pipeline Development Tools](#_bookmark111) [94](#_bookmark111)

[Blue Ocean Editor](#_bookmark112) [95](#_bookmark112)

[Command-line Pipeline Linter](#_bookmark113) [96](#_bookmark113)

[Examples](#_bookmark114) [96](#_bookmark114)

["Replay" Pipeline Runs with Modifications](#_bookmark115) [98](#_bookmark115)

[Usage](#_bookmark116) [98](#_bookmark116)

[Features](#_bookmark117) [99](#_bookmark117)

[Limitations](#_bookmark118) [99](#_bookmark118)

[Pipeline Unit Testing Framework](#_bookmark119) [100](#_bookmark119)

[Pipeline Syntax](#_bookmark120) [101](#_bookmark120)

[Declarative Pipeline](#_bookmark121) [102](#_bookmark121)

[Sections](#_bookmark122) [102](#_bookmark122)

[Directives](#_bookmark125) [108](#_bookmark125)

[Steps](#_bookmark128) [118](#_bookmark128)

[Scripted Pipeline](#_bookmark129) [120](#_bookmark129)

[Flow Control](#_bookmark130) [120](#_bookmark130)

[Steps](#_bookmark131) [121](#_bookmark131)

[Differences from plain Groovy](#_bookmark132) [121](#_bookmark132)

[Syntax Comparison](#_bookmark133) [122](#_bookmark133)

[Blue Ocean](#_bookmark134) [123](#_bookmark134)

[What is Blue Ocean?](#_bookmark135) [124](#_bookmark135)

[Join the community](#_bookmark136) [125](#_bookmark136)

[Frequently Asked Questions](#_bookmark137) [126](#_bookmark137)

[Why does Blue Ocean exist?](#_bookmark138) [126](#_bookmark138)

[Where is the name from?](#_bookmark139) [126](#_bookmark139)

[What does this mean for the classic Jenkins UI?](#_bookmark140) [127](#_bookmark140)

[What does this mean for my plugins?](#_bookmark141) [127](#_bookmark141)

[What technologies are currently in use?](#_bookmark142) [127](#_bookmark142)

[Where can I find the source code?](#_bookmark143) [127](#_bookmark143)

[Getting Started with Blue Ocean](#_bookmark144) [128](#_bookmark144)

[Installing](#_bookmark145) [129](#_bookmark145)

[With Docker](#_bookmark146) [130](#_bookmark146)

[Starting Blue Ocean](#_bookmark147) [131](#_bookmark147)

[Navigation bar](#_bookmark148) [132](#_bookmark148)

[Switching to the "Classic" UI](#_bookmark149) [133](#_bookmark149)

[Creating Pipelines](#_bookmark150) [134](#_bookmark150)

[Starting Pipeline Creation](#_bookmark151) [135](#_bookmark151)

[Creating a Pipeline for a Git Repository](#_bookmark152) [136](#_bookmark152)

[Creating Pipelines for GitHub Repositories](#_bookmark153) [137](#_bookmark153)

[Provide a GitHub Access Token](#_bookmark154) [137](#_bookmark154)

[Select a GitHub Account or Organization](#_bookmark155) [138](#_bookmark155)

[Dashboard](#_bookmark158) [141](#_bookmark158)

[Navigation Bar](#_bookmark159) [142](#_bookmark159)

[Favorites](#_bookmark160) [143](#_bookmark160)

[Pipelines](#_bookmark161) [144](#_bookmark161)

[Health Icons](#_bookmark162) [144](#_bookmark162)

[Pipeline Run Status](#_bookmark163) [145](#_bookmark163)

[Activity View](#_bookmark164) [146](#_bookmark164)

[Navigation Bar](#_bookmark165) [147](#_bookmark165)

[Activity](#_bookmark166) [148](#_bookmark166)

[Branches](#_bookmark167) [149](#_bookmark167)

[Pull Requests](#_bookmark168) [150](#_bookmark168)

[Pipeline Run Details View](#_bookmark169) [151](#_bookmark169)

[Pipeline Run Status](#_bookmark170) [152](#_bookmark170)

[Special cases](#_bookmark171) [153](#_bookmark171)

[Pipelines outside of Souce Control](#_bookmark172) [153](#_bookmark172)

[Freestyle Projects](#_bookmark173) [153](#_bookmark173)

[Matrix projects](#_bookmark174) [153](#_bookmark174)

[Tabs](#_bookmark175) [154](#_bookmark175)

[Pipeline](#_bookmark176) [154](#_bookmark176)

[Changes](#_bookmark177) [154](#_bookmark177)

[Tests](#_bookmark178) [154](#_bookmark178)

[Artifacts](#_bookmark179) [155](#_bookmark179)

[Pipeline Editor](#_bookmark180) [157](#_bookmark180)

[Starting the editor](#_bookmark181) [158](#_bookmark181)

[Limitations](#_bookmark182) [159](#_bookmark182)

[Navigation bar](#_bookmark183) [160](#_bookmark183)

[Pipeline settings](#_bookmark184) [161](#_bookmark184)

[Agent](#_bookmark185) [161](#_bookmark185)

[Environment](#_bookmark186) [161](#_bookmark186)

[Stage editor](#_bookmark187) [162](#_bookmark187)

[Stage configuration](#_bookmark188) [163](#_bookmark188)

[Step configuration](#_bookmark189) [164](#_bookmark189)

[Save Pipeline dialog](#_bookmark190) [165](#_bookmark190)

[Jenkins Use-Cases](#_bookmark191) [166](#_bookmark191)

[Jenkins with .NET](#_bookmark192) [167](#_bookmark192)

[Jenkins with Java](#_bookmark193) [168](#_bookmark193)

[Jenkins with Python](#_bookmark194) [169](#_bookmark194)

[Test Reports](#_bookmark195) [170](#_bookmark195)

[Jenkins with Ruby](#_bookmark196) [171](#_bookmark196)

[Test Reports](#_bookmark197) [172](#_bookmark197)

[Coverage Reports](#_bookmark198) [173](#_bookmark198)

[Rails](#_bookmark199) [174](#_bookmark199)

[Operating Jenkins](#_bookmark200) [175](#_bookmark200)

[Backing-up/Restoring Jenkins](#_bookmark201) [176](#_bookmark201)

[Monitoring Jenkins](#_bookmark202) [177](#_bookmark202)

[Securing Jenkins](#_bookmark203) [178](#_bookmark203)

[Access Control](#_bookmark204) [179](#_bookmark204)

[Protect users of Jenkins from other threats](#_bookmark205) [180](#_bookmark205)

[Disabling Security](#_bookmark206) [181](#_bookmark206)

[Managing Jenkins with Chef](#_bookmark207) [182](#_bookmark207)

[Managing Jenkins with Puppet](#_bookmark208) [183](#_bookmark208)

[Scaling Jenkins](#_bookmark209) [184](#_bookmark209)

[Appendix](#_bookmark210) [185](#_bookmark210)

[Advanced Jenkins Installation](#_bookmark211) [186](#_bookmark211)

[Glossary](#_bookmark212) [187](#_bookmark212)

[General Terms](#_bookmark213) [188](#_bookmark213)

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.

## **Pre-install**

### **System Requirements**

Minimum Recommended Configuration:

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

#### 

#### **Debian/Ubuntu**

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](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.

### **Docker**

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

## **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**

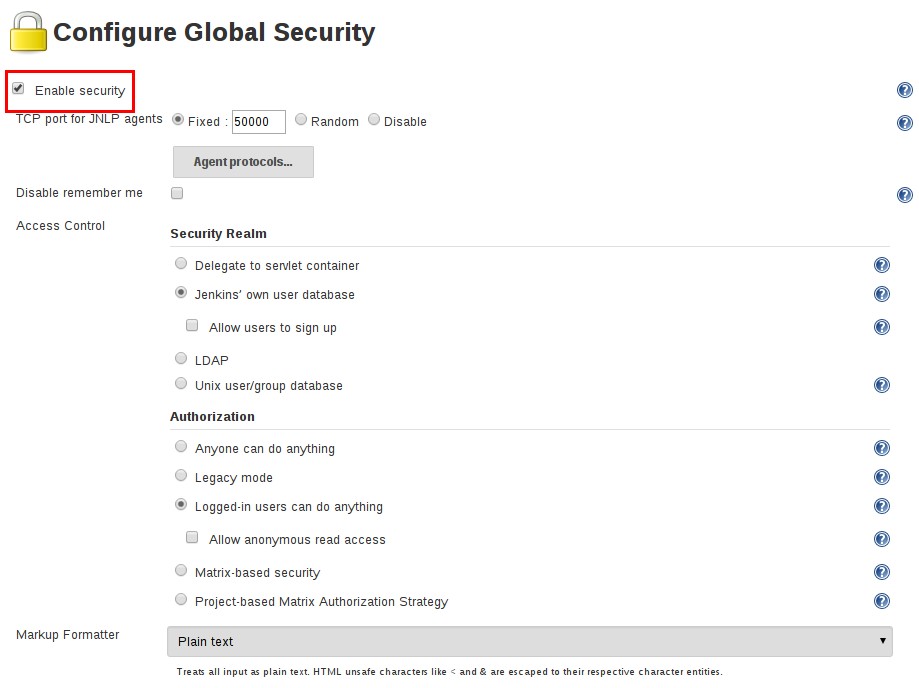
The Setup Wizard will also install the initial plugins for this Jenkins server. The recommended set of plugins available are based on the most common use cases. You are free to add more during the Setup Wizard or install them later as needed.

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.

## **Enabling Security**

When the **Enable Security** checkbox is checked, which has been the default since Jenkins 2.0, users can log in with a username and password in order to perform operations not available to anonymous users. Which operations require users to log in depends on the chosen authorization strategy and its configuration; by default anonymous users have no permissions, and logged in users have full control. This checkbox should **always** be enabled for any non-local (test) Jenkins environment.

The Enable Security section of the web UI allows a Jenkins administrator to enable, configure, or disable key security features which apply to the entire Jenkins environment.



### **JNLP TCP Port**

Jenkins uses a TCP port to communicate with agents launched via the JNLP protocol, such as Windows-based agents. As of Jenkins 2.0, by default this port is disabled.

For administrators wishing to use JNLP-based agents, the two port options are:

1. **Random**: The JNLP port is chosen random to avoid collisions on the Jenkins master. The downside to randomized JNLP ports is that they’re chosen during the boot of the Jenkins master, making it difficult to manage firewall rules allowing JNLP traffic.
2. **Fixed**: The JNLP port is chosen by the Jenkins administrator and is consistent across reboots of

the Jenkins master. This makes it easier to manage firewall rules allowing JNLP-based agents to connect to the master.

### **Access Control**

Access Control is the primary mechanism for securing a Jenkins environment against unauthorized usage. Two facets of configuration are necessary for configuring Access Control in Jenkins:

1. A **Security Realm** which informs the Jenkins environment how and where to pull user (or identity) information from. Also commonly known as "authentication."
2. **Authorization** configuration which informs the Jenkins environment as to which users and/or groups can access which aspects of Jenkins, and to what extent.

Using both the Security Realm and Authorization configurations it is possible to configure very relaxed or very rigid authentication and authorization schemes in Jenkins.

Additionally, some plugins such as the plugin:role-strategy[Role-based Authorization Strategy] plugin can extend the Access Control capabilities of Jenkins to support even more nuanced authentication and authorization schemes.

#### **Security Realm**

By default Jenkins includes support for a few different Security Realms:

**Delegate to servlet container**

For delegating authentication a servlet container running the Jenkins master, such as [Jetty](http://www.eclipse.org/jetty/). If using this option, please consult the servlet container’s authentication documentation.

**Jenkins’ own user database**

Use Jenkins’s own built-in user data store for authentication instead of delegating to an external system. This is enabled by default with new Jenkins 2.0 or later installations and is suitable for smaller environments.

**LDAP**

Delegate all authentication to a configured LDAP server, including both users and groups. This option is more common for larger installations in organizations which already have configured an external identity provider such as LDAP. This also supports Active Directory installations.

**NOTE**

This feature is provided by the plugin:ldap[LDAP plugin] that may not be installed on your instance.

**Unix user/group database**

Delegates the authentication to the underlying Unix OS-level user database on the Jenkins master. This mode will also allow re-use of Unix groups for authorization. For example, Jenkins can be configured such that "Everyone in the developers group has administrator access." To support this feature, Jenkins relies on [PAM](http://en.wikipedia.org/wiki/Pluggable_Authentication_Modules) which may need to be configured external to the Jenkins environment.

**CAUTION**

Unix allows an user and a group to have the same name. In order to disambiguate, use the @ prefix to force the name to be interpreted as a group. For example, @dev would mean the dev group and not the dev user.

Plugins can provide additional security realms which may be useful for incorporating Jenkins into existing identity systems, such as:

* plugin:active-directory[Active Directory]
* plugin:github-oauth[GitHub Authentication]
* plugin:crowd2[Atlassian Crowd 2]

#### **Authorization**

The Security Realm, or authentication, indicates *who* can access the Jenkins environment. The other piece of the puzzle is **Authorization**, which indicates *what* they can access in the Jenkins environment. By default Jenkins supports a few different Authorization options:

**Anyone can do anything**

Everyone gets full control of Jenkins, including anonymous users who haven’t logged in. **Do not use this setting** for anything other than local test Jenkins masters.

**Legacy mode**

Behaves exactly the same as Jenkins <1.164. Namely, if a user has the "admin" role, they will be granted full control over the system, and otherwise (including anonymous users) will only have the read access. **Do not use this setting** for anything other than local test Jenkins masters.

**Logged in users can do anything**

In this mode, every logged-in user gets full control of Jenkins. Depending on an advanced option, anonymous users get read access to Jenkins, or no access at all. This mode is useful to force users to log in before taking actions, so that there is an audit trail of users' actions.

**Matrix-based security**

This authorization scheme allows for granular control over which users and groups are able to perform which actions in the Jenkins environment (see the screenshot below).

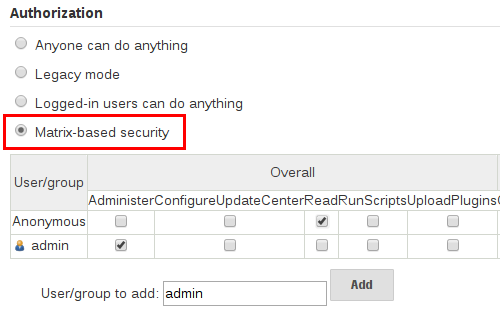
**Project-based Matrix Authorization Strategy**

This authorization scheme is an extension to Matrix-based security which allows additional access control lists (ACLs) to be defined for **each project** separately in the Project configuration screen. This allows granting specific users or groups access only to specified projects, instead of all projects in the Jenkins environment. The ACLs defined with Project-based Matrix Authorization are additive such that access grants defined in the Configure Global Security screen will be combined with project-specific ACLs.

**NOTE**

Matrix-based security and Project-based Matrix Authorization Strategy are provided by the plugin:matrix-auth[Matrix Authorization Strategy Plugin] and may not be installed on your Jenkins.

For most Jenkins environments, Matrix-based security provides the most security and flexibility so it is recommended as a starting point for "production" environments.



*Figure 1. Matrix-based security*

The table shown above can get quite wide as each column represents a permission provided by Jenkins core or a plugin. Hovering the mouse over a permission will display more information about the permission.

Each row in the table represents a user or group (also known as a "role"). This includes special entries named "anonymous" and "authenticated." The "anonymous" entry represents permissions granted to all unauthenticated users accessing the Jenkins environment. Whereas "authenticated' can be used to grant permissions to all authenticated users accessing the environment.

The permissions granted in the matrix are additive. For example, if a user "kohsuke" is in the groups "developers" and "administrators", then the permissions granted to "kohsuke" will be a union of all those permissions granted to "kohsuke", "developers", "administrators", "authenticated", and "anonymous."

### **Markup Formatter**

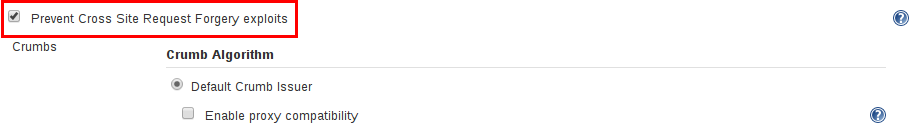
Jenkins allows user-input in a number of different configuration fields and text areas which can lead to users inadvertently, or maliciously, inserting unsafe HTML and/or JavaScript.

By default the **Markup Formatter** configuration is set to **Plain Text** which will escape unsafe characters such as < and & to their respective character entities.

Using the **Safe HTML** Markup Formatter allows for users and administrators to inject useful and information HTML snippets into Project Descriptions and elsewhere.

## **Cross Site Request Forgery**

A cross site request forgery (or CSRF/XSRF) [1: [www.owasp.org/index.php/Cross-](https://www.owasp.org/index.php/Cross-Site_Request_Forgery) [Site\_Request\_Forgery](https://www.owasp.org/index.php/Cross-Site_Request_Forgery)] is an exploit that enables an unauthorized third party to perform requests against a web application by impersonating another, authenticated, user. In the context of a Jenkins environment, a CSRF attack could allow an malicious actor to delete projects, alter builds, or modify Jenkins' system configuration. To guard against this class of vulnerabilities, CSRF protection has been enabled by default with all Jenkins versions since 2.0.



When the option is enabled, Jenkins will check for a CSRF token, or "crumb", on any request that may change data in the Jenkins environment. This includes any form submission and calls to the remote API, including those using "Basic" authentication.

It is **strongly recommended** that this option be left **enabled**, including on instances operating on private, fully trusted networks.

### **Caveats**

CSRF protection *may* result in challenges for more advanced usages of Jenkins, such as:

* Some Jenkins features, like the remote API, are more difficult to use when this option is enabled. Consult the Remote API documentation for more information.
* Accessing Jenkins through a poorly-configured reverse proxy may result in the CSRF HTTP header being stripped from requests, resulting in protected actions failing.
* Out-dated plugins, not tested with CSRF protection enabled, may not properly function. More information about CSRF exploits can be found [on the OWASP website](http://www.owasp.org/index.php/Cross-Site_Request_Forgery).

## **Agent/Master Access Control**

Conceptually, the Jenkins master and agents can be thought of as a cohesive system which happens to execute across multiple discrete processes and machines. This allows an agent to ask the master process for information available to it, for example, the contents of files, etc.

For larger or mature Jenkins environments where a Jenkins administrator might enable agents provided by other teams or organizations, a flat agent/master trust model is insufficient.

The Agent/Master Access Control system was introduced [2: Starting with 1.587, and 1.580.1, releases] to allow Jenkins administrators to add more granular access control definitions between the Jenkins master and the connected agents.



As of Jenkins 2.0, this subsystem has been turned on by default.

### **Customizing Access**

For advanced users who may wish to allow certain access patterns from the agents to the Jenkins master, Jenkins allows administrators to create specific exemptions from the built-in access control rules.



By following the link highlighted above, an administrator may edit **Commands** and **File Access**

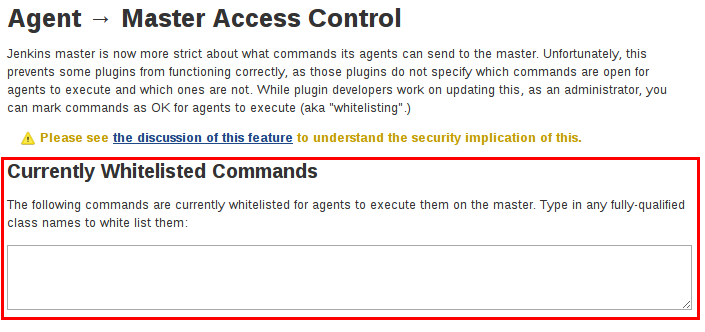
Agent/Master access control rules.

#### **Commands**

"Commands" in Jenkins and its plugins are identified by their fully-qualified class names. The majority of these commands are intended to be executed on agents by a request of a master, but some of them are intended to be executed on a master by a request of an agent.

Plugins not yet updated for this subsystem may not classify which category each command falls into, such that when an agent requests that the master execute a command which is not explicitly allowed, Jenkins will err on the side of caution and refuse to execute the command.

In such cases, Jenkins administrators may "whitelist" [3: [en.wikipedia.org/wiki/Whitelist](https://en.wikipedia.org/wiki/Whitelist)] certain commands as acceptable for execution on the master.



**Advanced**

Administrators may also whitelist classes by creating files with the .conf extension in the directory JENKINS\_HOME/secrets/whitelisted-callables.d/. The contents of these .conf files should list command names on separate lines.

The contents of all the .conf files in the directory will be read by Jenkins and combined to create a default.conf file in the directory which lists all known safe command. The default.conf file will be re-written each time Jenkins boots.

Jenkins also manages a file named gui.conf, in the whitelisted-callables.d directory, where commands added via the web UI are written. In order to disable the ability of administrators to change whitelisted commands from the web UI, place an empty gui.conf file in the directory and change its permissions such that is not writeable by the operating system user Jenkins run as.

#### **File Access Rules**

The File Access Rules are used to validate file access requests made from agents to the master. Each File Access Rule is a triplet which must contain each of the following elements:

1. allow / deny: if the following two parameters match the current request being considered, an allow entry would allow the request to be carried out and a deny entry would deny the request to be rejected, regardless of what later rules might say.
2. *operation*: Type of the operation requested. The following 6 values exist. The operations can also be combined by comma-separating the values. The value of all indicates all the listed operations are allowed or denied.
   * read: read file content or list directory entries
   * write: write file content
   * mkdirs: create a new directory
   * create: create a file in an existing directory
   * delete: delete a file or directory
   * stat: read metadata of a file/directory, such as timestamp, length, file access modes.
3. *file path*: regular expression that specifies file paths that matches this rule. In addition to the base regexp syntax, it supports the following tokens:
   * <JENKINS\_HOME> can be used as a prefix to match the master’s JENKINS\_HOME directory.
   * <BUILDDIR> can be used as a prefix to match the build record directory, such as

/var/lib/jenkins/job/foo/builds/2014-10-17\_12-34-56.

* + <BUILDID> matches the timestamp-formatted build IDs, like 2014-10-17\_12-34-56.

The rules are ordered, and applied in that order. The earliest match wins. For example, the following rules allow access to all files in JENKINS\_HOME except the secrets folders:

# To avoid hassle of escaping every '\' on Windows, you can use / even on Windows. deny all <JENKINS\_HOME>/secrets/.\*

allow all <JENKINS\_HOME>/.\*

Ordering is very important! The following rules are incorrectly written because the 2nd rule will never match, and allow all agents to access all files and folders under JENKINS\_HOME:

allow all <JENKINS\_HOME>/.\*

deny all <JENKINS\_HOME>/secrets/.\*

**Advanced**

Administrators may also add File Access Rules by creating files with the .conf. extension in the directory JENKINS\_HOME/secrets/filepath-filters.d/. Jenkins itself generates the 30-default.conf file on boot in this directory which contains defaults considered the best balance between compatibility and security by the Jenkins project. In order to disable these built-in defaults, replace 30- default.conf with an empty file which is not writable by the operating system user Jenkins run as.

On each boot, Jenkins will read all .conf files in the filepath-filters.d directory in alphabetical order, therefore it is good practice to name files in a manner which indicates their load order.

Jenkins also manages 50-gui.conf, in the filepath-filters/ directory, where File Access Rules added via the web UI are written. In order to disable the ability of administrators to change the File Access Rules from the web UI, place an empty 50-gui.conf file in the directory and change its permissions such that is not writeable by the operating system user Jenkins run as.

### **Disabling**

While it is not recommended, if all agents in a Jenkins environment can be considered "trusted" to the same degree that the master is trusted, the Agent/Master Access Control feature may be disabled.

Additionally, all the users in the Jenkins environment should have the same level of access to all configured projects.

An administrator can disable Agent/Master Access Control in the web UI by un-checking the box on the **Configure Global Security** page. Alternatively an administrator may create a file in JENKINS\_HOME/secrets named slave-to-master-security-kill-switch with the contents of true and restart Jenkins.

Most Jenkins environments grow over time requiring their trust models to evolve as the environment grows. Please consider scheduling regular "check- ups" to review whether any disabled security settings should be re-enabled.

# **Managing Plugins**

Plugins are the primary means of enhancing the functionality of a Jenkins environment to suit organization- or user-specific needs. There are [over a thousand different plugins](https://plugins.jenkins.io/) which can be installed on a Jenkins master and to integrate various build tools, cloud providers, analysis tools, and much more.

Plugins can be automatically downloaded, with their dependencies, from the Update Center. The Update Center is a service operated by the Jenkins project which provides an inventory of open source plugins which have been developed and maintained by various members of the Jenkins community.

This section will cover everything from the basics of managing plugins within the Jenkins web UI, to making changes on the master’s file system.

## **Installing a plugin**

Jenkins provides a couple of different methods for installing plugins on the master:

1. Using the "Plugin Manager" in the web UI.
2. Using the [Jenkins CLI](#_bookmark42) install-plugin command.

Each approach will result in the plugin being loaded by Jenkins but may require different levels of access and trade-offs in order to use.

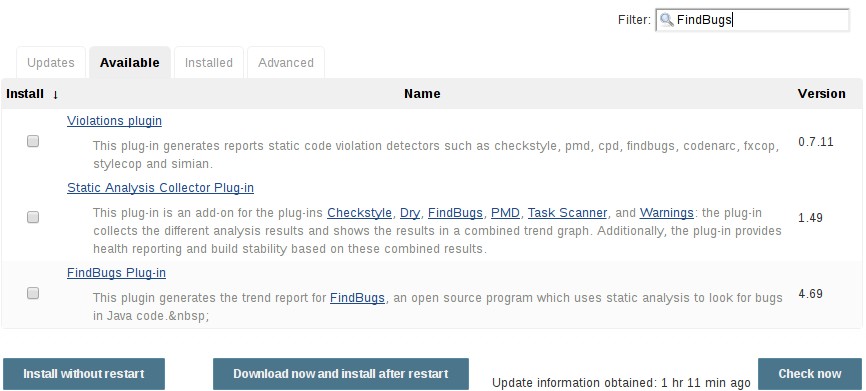
The two approaches require that the Jenkins master be able to download meta-data from an Update Center, whether the primary Update Center operated by the Jenkins project [4: [updates.jenkins.io](https://updates.jenkins.io/)], or a custom Update Center.

The plugins are packaged as self-contained .hpi files, which have all the necessary code, images, and other resources which the plugin needs to operate successfully.

### **From the web UI**

The simplest and most common way of installing plugins is through the **Manage Jenkins** > **Manage Plugins** view, available to administrators of a Jenkins environment.

Under the **Available** tab, plugins available for download from the configured Update Center can be searched and considered:



Most plugins can be installed and used immediately by checking the box adjacent to the plugin and clicking **Install without restart**.

**CAUTION**

If the list of available plugins is empty, the master might be incorrectly configured or has not yet downloaded plugin meta-data from the Update Center. Clicking the **Check now** button will force Jenkins to attempt to contact its configured Update Center.

### **Using the Jenkins CLI**

Administrators may also use the Jenkins CLI which provides a command to install plugins. Scripts to manage Jenkins environments, or configuration management code, may need to install plugins without direct user interaction in the web UI. The Jenkins CLI allows a command line user or automation tool to download a plugin and its dependencies.

java -jar jenkins-cli.jar -s http://localhost:8080/ install-plugin SOURCE ... [- deploy] [-name VAL] [-restart]

Installs a plugin either from a file, an URL, or from update center.

SOURCE : If this points to a local file, that file will be installed. If this is an URL, Jenkins downloads the URL and installs that as a plugin.Otherwise the name is assumed to be the short name of the plugin in the existing update center (like "findbugs"),and the plugin will be installed from the update center.

-deploy : Deploy plugins right away without postponing them until the reboot.

-name VAL : If specified, the plugin will be installed as this short name (whereas normally the name is inferred from the source name automatically).

-restart : Restart Jenkins upon successful installation.

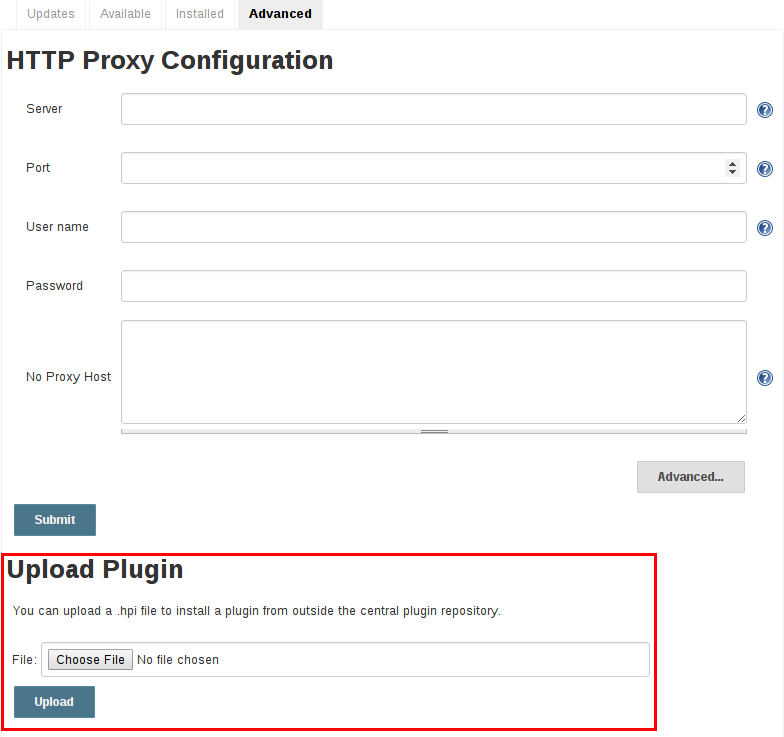
### **Advanced installation**

The Update Center only allows the installation of the most recently released version of a plugin. In cases where an older release of the plugin is desired, a Jenkins administrator can download an older .hpi archive and manually install that on the Jenkins master.

#### **From the web UI**

Assuming a .hpi file has been downloaded, a logged-in Jenkins administrator may upload the file from within the web UI:

1. Navigate to the **Manage Jenkins** > **Manage Plugins** page in the web UI.
2. Click on the **Advanced** tab.
3. Choose the .hpi file under the **Upload Plugin** section.
4. **Upload** the plugin file.



Once a plugin file has been uploaded, the Jenkins master must be manually restarted in order for the changes to take effect.

#### **On the master**

Assuming a .hpi file has been explicitly downloaded by a systems administrator, the administrator can manually place the .hpi file in a specific location on the file system.

Copy the downloaded .hpi` file into the JENKINS\_HOME/plugins directory on the Jenkins master (for example, on Debian systems JENKINS\_HOME is generally /var/lib/jenkins).

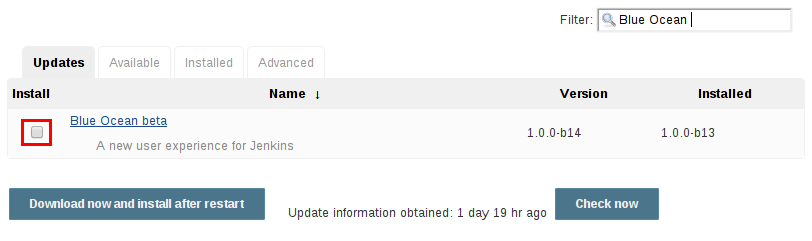
The master will need to be restarted before the plugin is loaded and made available in the Jenkins environment.

**NOTE**

The names of the plugin directories in the Update Site [4: [updates.jenkins.io](https://updates.jenkins.io/)] are not always the same as the plugin’s display name. Searching [plugins.jenkins.io](https://plugins.jenkins.io/) for the desired plugin will provide the appropriate link to the .hpi files.

## **Updating a plugin**

Updates are listed in the **Updates** tab of the **Manage Plugins** page and can be installed by checking the checkboxes of the desired plugin updates and clicking the **Download now and install after restart** button.



By default, the Jenkins master will check for updates from the Update Center once every 24 hours. To manually trigger a check for updates, simply click on the **Check now** button in the **Updates** tab.

## **Removing a plugin**

When a plugin is no longer used in a Jenkins environment, it is prudent to remove the plugin from the Jenkins master. This provides a number of benefits such as reducing memory overhead at boot or runtime, reducing configuration options in the web UI, and removing the potential for future conflicts with new plugin updates.

### **Uninstalling a plugin**

The simplest way to uninstall a plugin is to navigate to the **Installed** tab on the **Manage Plugins** page. From there, Jenkins will automatically determine which plugins are safe to uninstall, those which are not dependencies of other plugins, and present a button for doing so.



A plugin may also be uninstalled by removing the corresponding .hpi file from the JENKINS\_HOME/plugins directory on the master. The plugin will continue to function until the master has been restarted.

**CAUTION**

If a plugin .hpi file is removed but required by other plugins, the Jenkins master may fail to boot correctly.

Uninstalling a plugin does **not** remove the configuration that the plugin may have created. If there are existing jobs/nodes/views/builds/etc configurations that reference data created by the plugin, during boot Jenkins will warn that some configurations could not be fully loaded and ignore the unrecognized data.

Since the configuration(s) will be preserved until they are overwritten, re-installing the plugin will result in those configuration values reappearing.

#### **Removing old data**

Jenkins provides a facility for purging configuration left behind by uninstalled plugins. Navigate to

**Manage Jenkins** and then click on **Manage Old Data** to review and remove old data.

### **Disabling a plugin**

Disabling a plugin is a softer way to retire a plugin. Jenkins will continue to recognize that the plugin is installed, but it will not start the plugin, and no extensions contributed from this plugin will be visible.

A Jenkins administrator may disable a plugin by unchecking the box on the **Installed** tab of the

**Manage Plugins** page (see below).



A systems administrator may also disable a plugin by creating a file on the Jenkins master, such as:

JENKINS\_HOME/plugins/PLUGIN\_NAME.hpi.disabled.

The configuration(s) created by the disabled plugin behave as if the plugin were uninstalled, insofar that they result in warnings on boot but are otherwise ignored.

## **Pinned plugins**

**CAUTION**

Pinned plugins feature was removed in Jenkins 2.0. Versions later than Jenkins

2.0 do not bundle plugins, instead providing a wizard to install the most useful plugins.

The notion of **pinned plugins** applies to plugins that are bundled with Jenkins 1.x, such as the plugin:matrix-auth[**Matrix Authorization plugin**].

By default, whenever Jenkins is upgraded, its bundled plugins overwrite the versions of the plugins that are currently installed in JENKINS\_HOME.

However, when a bundled plugin has been manually updated, Jenkins will mark that plugin as pinned to the particular version. On the file system, Jenkins creates an empty file called JENKINS\_HOME/plugins/PLUGIN\_NAME.hpi.pinned to indicate the pinning.

Pinned plugins will never be overwritten by bundled plugins during Jenkins startup. (Newer versions of Jenkins do warn you if a pinned plugin is *older* than what is currently bundled.)

It is safe to update a bundled plugin to a version offered by the Update Center. This is often necessary to pick up the newest features and fixes. The bundled version is occasionally updated, but not consistently.

The Plugin Manager allows plugins to be explicitly unpinned. The JENKINS\_HOME/plugins/PLUGIN\_NAME.hpi.pinned file can also be manually created/deleted to control the pinning behavior. If the pinned file is present, Jenkins will use whatever plugin version the user has specified. If the file is absent, Jenkins will restore the plugin to the default version on startup.

# **Jenkins CLI**

Jenkins has a built-in command line interface that allows users and administrators to access Jenkins from a script or shell environment. This can be convenient for scripting of routine tasks, bulk updates, troubleshooting, and more.

The command line interface can be accessed over SSH or with the Jenkins CLI client, a .jar file distributed with Jenkins.

**WARNING**

Use of the CLI client distributed with Jenkins 2.53 and older and Jenkins LTS

* + 1. and older is **not recommended** for security reasons: while there are no currently known vulnerabilities, several have been reported and patched in the past, and the Jenkins Remoting protocol it uses is inherently vulnerable to remote code execution bugs, even “preauthentication” exploits (by anonymous users able to physically access the Jenkins network).

The client distributed with Jenkins 2.54 and newer and Jenkins LTS 2.46.2 and newer is considered secure in its default (-http) or -ssh modes, as is using the standard ssh command.

## **Using the CLI over SSH**

In a new Jenkins installation, the SSH service is disabled by default. Administrators may choose to set a specific port or ask Jenkins to pick a random port in the Configure Global Security page. In order to determine the randomly assigned SSH port, inspect the headers returned on a Jenkins URL, for example:

% curl -Lv https://JENKINS\_URL/login 2>&1 | grep 'X-SSH-Endpoint'

< X-SSH-Endpoint: localhost:53801

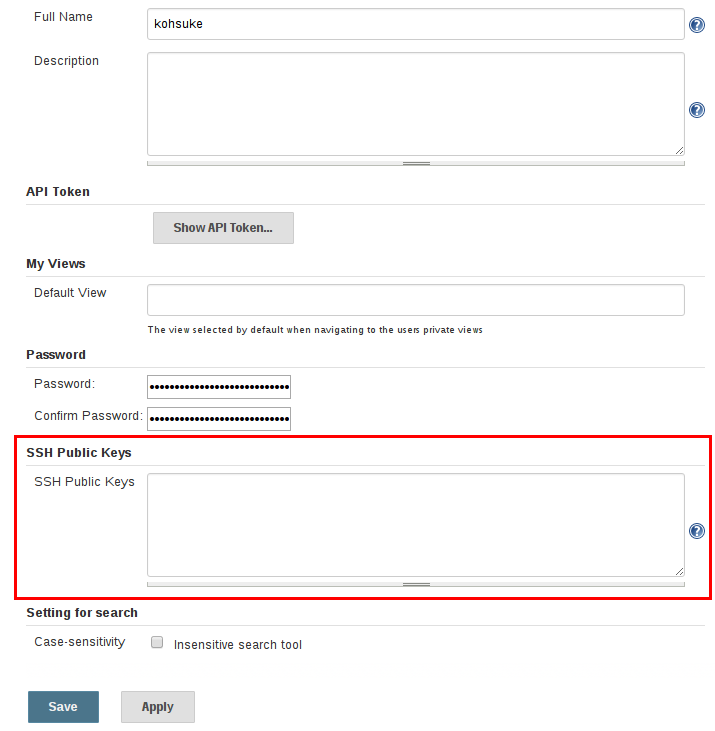
%

With the random SSH port (53801 in this example), and [Authentication](#_bookmark51) configured, any modern SSH client may securely execute CLI commands.

### **Authentication**

Whichever user used for authentication with the Jenkins master must have the Overall/Read permission in order to *access* the CLI. The user may require additional permissions depending on the commands executed.

Authentication relies on SSH-based public/private key authentication. In order to add an SSH public key for the appropriate user, navigate to JENKINS\_URL/user/USERNAME/configure and paste an SSH public key into the appropriate text area.



### **Common Commands**

Jenkins has a number of built-in CLI commands which can be found in every Jenkins environment, such as build or list-jobs. Plugins may also provide CLI commands; in order to determine the full list of commands available in a given Jenkins environment, execute the CLI help command:

% ssh -l kohsuke -p 53801 localhost help

The following list of commands is not comprehensive, but it is a useful starting point for Jenkins CLI usage.

#### **build**

One of the most common and useful CLI commands is build, which allows the user to trigger any job or Pipeline for which they have permission.

The most basic invocation will simply trigger the job or Pipeline and exit, but with the additional options a user may also pass parameters, poll SCM, or even follow the console output of the triggered build or Pipeline run.

##### % ssh -l kohsuke -p 53801 localhost help build

##### java -jar jenkins-cli.jar build JOB [-c] [-f] [-p] [-r N] [-s] [-v] [-w] Starts a build, and optionally waits for a completion. Aside from general scripting use, this command can be used to invoke another job from within a

##### build of one job. With the -s option, this command changes the exit code based on the outcome of the build (exit code 0 indicates a success) and interrupting the command will interrupt the job. With the -f option, this command changes the exit code based on the outcome of the build (exit code 0 indicates a success) however, unlike -s, interrupting the command will not interrupt the job (exit code 125 indicates the command was interrupted). With the -c option, a build will only run if there has been an SCM change.

##### JOB : Name of the job to build

##### -c : Check for SCM changes before starting the build, and if there's no change, exit without doing a build

##### -f : Follow the build progress. Like -s only interrupts are not passed through to the build.

##### -p : Specify the build parameters in the key=value format.

##### -s : Wait until the completion/abortion of the command. Interrupts are passed through to the build.

##### -v : Prints out the console output of the build. Use with -s

##### -w : Wait until the start of the command

##### % ssh -l kohsuke -p 53801 localhost build build-all-software -f -v Started build-all-software #1

##### Started from command line by admin

##### Building in workspace /tmp/jenkins/workspace/build-all-software [build-all-software] $ /bin/sh -xe /tmp/hudson1100603797526301795.sh

##### + echo hello world hello world Finished: SUCCESS

##### Completed build-all-software #1 : SUCCESS

##### %

#### **console**

Similarly useful is the console command, which retrieves the console output for the specified build or Pipeline run. When no build number is provided, the console command will output the last completed build’s console output.

% ssh -l kohsuke -p 53801 localhost help console

java -jar jenkins-cli.jar console JOB [BUILD] [-f] [-n N]

Produces the console output of a specific build to stdout, as if you are doing 'cat build.log'

JOB : Name of the job

BUILD : Build number or permalink to point to the build. Defaults to the last build

-f : If the build is in progress, stay around and append console output as it comes, like 'tail -f'

-n N : Display the last N lines

% ssh -l kohsuke -p 53801 localhost console build-all-software Started from command line by kohsuke

Building in workspace /tmp/jenkins/workspace/build-all-software [build-all-software] $ /bin/sh -xe /tmp/hudson1100603797526301795.sh

+ echo hello world yes

Finished: SUCCESS

%

#### **who-am-i**

The who-am-i command is helpful for listing the current user’s credentials and permissions available to the user. This can be useful when debugging the absence of CLI commands due to the lack of certain permissions.

% ssh -l kohsuke -p 53801 localhost help who-am-i

java -jar jenkins-cli.jar who-am-i Reports your credential and permissions.

% ssh -l kohsuke -p 53801 localhost who-am-i Authenticated as: kohsuke

Authorities: authenticated

%

## **Using the CLI client**

While the SSH-based CLI is fast and covers most needs, there may be situations where the CLI client distributed with Jenkins is a better fit. For example, the default transport for the CLI client is HTTP which means no additional ports need to be opened in a firewall for its use.

### **Downloading the client**

The CLI client can be downloaded directly from a Jenkins master at the URL /jnlpJars/jenkins- cli.jar, in effect JENKINS\_URL/jnlpJars/jenkins-cli.jar

While a CLI .jar can be used against different versions of Jenkins, should any compatibility issues arise during use, please re-download the latest .jar file from the Jenkins master.

### **Using the client**

The general syntax for invoking the client is as follows:

java -jar jenkins-cli.jar [-s JENKINS\_URL] [global options...] command [command options...] [arguments...]

The JENKINS\_URL can be specified via the environment variable $JENKINS\_URL. Summaries of other general options can be displayed by running the client with no arguments at all.

### **Client connection modes**

There are three basic modes in which the 2.54+ / 2.46.2+ client may be used, selectable by global option: -http; -ssh; and -remoting.

#### **HTTP connection mode**

This is the default mode as of 2.54 and 2.46.2, though you may pass the -http option explicitly for clarity.

Authentication is preferably with an -auth option, which takes a username:apitoken argument. Get your API token from /me/configure:

java -jar jenkins-cli.jar [-s JENKINS\_URL] -auth kohsuke:abc1234ffe4a command ...

(Actual passwords are also accepted, but this is discouraged.)

You can also precede the argument with @ to load the same content from a file:

java -jar jenkins-cli.jar [-s JENKINS\_URL] -auth @/home/kohsuke/.jenkins-cli command

...

Generally no special system configuration need be done to enable HTTP-based CLI connections. If you are running Jenkins behind an HTTP(S) reverse proxy, ensure it does not buffer request or response bodies.

#### **SSH connection mode**

Authentication is via SSH keypair. You must select the Jenkins user ID as well:

java -jar jenkins-cli.jar [-s JENKINS\_URL] -ssh -user kohsuke command ...

In this mode, the client acts essentially like a native ssh command.

By default the client will try to connect to an SSH port on the same host as is used in the JENKINS\_URL. If Jenkins is behind an HTTP reverse proxy, this will not generally work, so run Jenkins with the system property -Dorg.jenkinsci.main.modules.sshd.SSHD.hostName=ACTUALHOST to define a hostname or IP address for the SSH endpoint.

#### **Remoting connection mode**

This was the only mode supported by clients downloaded from a pre-2.54 / pre-2.46.2 Jenkins server (prior to the introduction of the -remoting option). Its use is deprecated for security and performance reasons. That said, certain commands or command modes can *only* run in Remoting mode, typically because the command functionality involves running server-supplied code on the client machine.

This mode is disabled on the server side for new installations of 2.54+ and 2.46.2. If you must use it, and accept the risks, it may be enabled in Configure Global Security.

Authentication is preferably via SSH keypair. A login command and --username / --password command (note: **not global**) options are also available; these are discouraged since they cannot work with a non-password-based security realm, certain command arguments will not be properly parsed if anonymous users lack overall or job read access, and saving human-chosen passwords for use in scripts is considered insecure.

Note that there are two transports available for this mode: over HTTP, or over a dedicated TCP socket. If the TCP port is enabled and seems to work, the client will use this transport. If the TCP port is disabled, or such a port is advertised but does not accept connections (for example because you are using an HTTP reverse proxy with a firewall), the client will automatically fall back to the less efficient HTTP transport.

**Common Problems with the Remoting-based client**

There are a number of common problems that may be experienced when running the CLI client.

**Operation timed out**

Check that the HTTP or TCP port is opened if you are using a firewall on your server. You can configure its value in Jenkins configuration. By default it is set to use a random port.

% java -jar jenkins-cli.jar -s JENKINS\_URL help

Exception in thread "main" java.net.ConnectException: Operation timed out at java.net.PlainSocketImpl.socketConnect(Native Method)

at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:351)

at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:213) at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:200)

at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:432) at java.net.Socket.connect(Socket.java:529)

at java.net.Socket.connect(Socket.java:478) at java.net.Socket.<init>(Socket.java:375) at java.net.Socket.<init>(Socket.java:189) at hudson.cli.CLI.<init>(CLI.java:97)

at hudson.cli.CLI.<init>(CLI.java:82) at hudson.cli.CLI.\_main(CLI.java:250) at hudson.cli.CLI.main(CLI.java:199)

**No X-Jenkins-CLI2-Port**

Go to **Manage Jenkins** > **Configure Global Security** and choose "Fixed" or "Random" under **TCP port for JNLP agents**.

##### java.io.IOException: No X-Jenkins-CLI2-Port among [X-Jenkins, null, Server, X-Content- Type-Options, Connection,

##### X-You-Are-In-Group, X-Hudson, X-Permission-Implied-By, Date, X-Jenkins- Session, X-You-Are-Authenticated-As,

##### X-Required-Permission, Set-Cookie, Expires, Content-Length, Content-Type] at hudson.cli.CLI.getCliTcpPort(CLI.java:284)

##### at hudson.cli.CLI.<init>(CLI.java:128)

##### at hudson.cli.CLIConnectionFactory.connect(CLIConnectionFactory.java:72) at hudson.cli.CLI.\_main(CLI.java:473)

##### at hudson.cli.CLI.main(CLI.java:384)

##### Suppressed: java.io.IOException: Server returned HTTP response code: 403 for URL:<http://citest.gce.px/cli>

##### at sun.net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.java:184 0)

##### at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1441

##### )

##### at hudson.cli.FullDuplexHttpStream.<init>(FullDuplexHttpStream.java:78) at hudson.cli.CLI.connectViaHttp(CLI.java:152)

##### at hudson.cli.CLI.<init>(CLI.java:132)

##### ... 3 more

# **Pipeline**

Running Pipelines , writing Pipeline code, and extending Pipeline itself.

## **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. [5: [Domain-Specific Language](https://en.wikipedia.org/wiki/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:

// Declarative // pipeline {

agent any ①

stages {

stage('Build') { ②

steps { ③

sh 'make' ④

}

}

stage('Test'){ steps {

sh 'make check'

junit 'reports/\*\*/\*.xml' ⑤

}

}

stage('Deploy') { steps {

sh 'make publish'

}

}

}

}

// Script // node {

stage('Build') { sh 'make'

}

stage('Test') {

sh 'make check'

junit 'reports/\*\*/\*.xml'

}

stage('Deploy') {

sh 'make publish'

}

}

① agent indicates that Jenkins should allocate an executor and workspace for this part of the

Pipeline.

② stage describes a stage of this Pipeline.

③ steps describes the steps to be run in this stage

④ sh executes the given shell command

⑤ junit is a Pipeline step provided by the plugin:junit[JUnit plugin] for aggregating test reports.

## **Why Pipeline?**

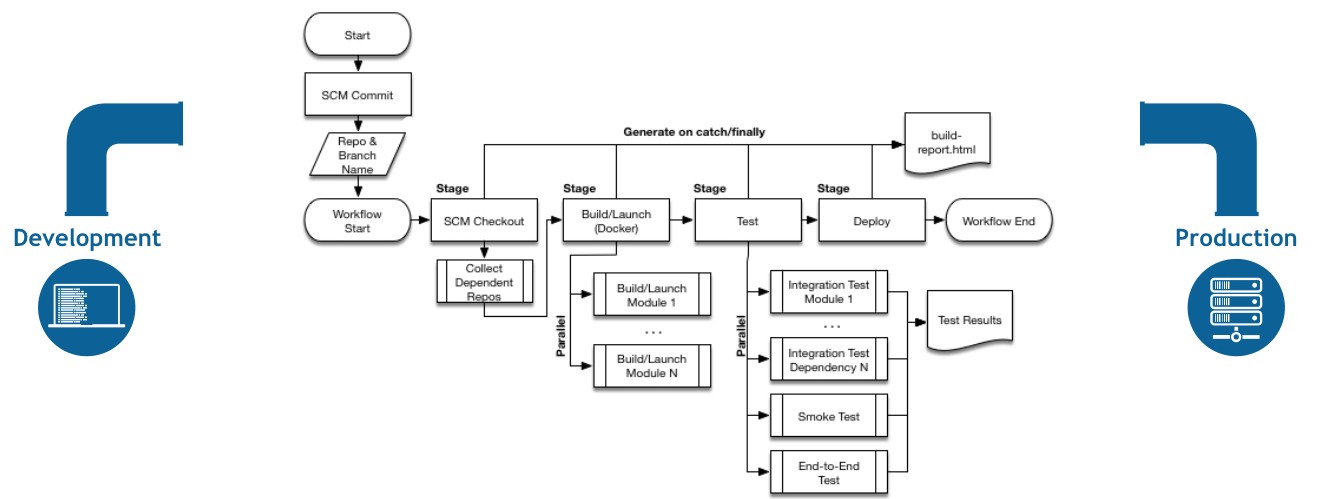
Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive continuous delivery pipelines. By modeling a series of related 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 [5: [Domain-Specific](https://en.wikipedia.org/wiki/Domain-specific_language) [Language](https://en.wikipedia.org/wiki/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, [6: 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. [7: plugin:github-organization-folder[GitHub Organization Folder plugin]]

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



*Figure 2. 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](https://issues.jenkins-ci.org/browse/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 used by many plugins to visualize or present Jenkins Pipeline status/progress. [8: Blue Ocean, [Pipeline Stage View plugin](https://wiki.jenkins-ci.org/display/JENKINS/Pipeline%2BStage%2BView%2BPlugin)]

# **Getting Started with 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. [9: [Domain-Specific Language](https://en.wikipedia.org/wiki/Domain-specific_language)]

## **Defining a Pipeline**

Scripted Pipeline is written in [Groovy](http://groovy-lang.org/). The relevant bits of [Groovy syntax](http://groovy-lang.org/semantics.html) 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:

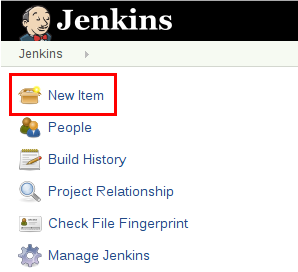
* By entering a script directly in the Jenkins web UI.
* By creating a Jenkinsfile which can be checked into a project’s source control repository.

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. [11: [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:

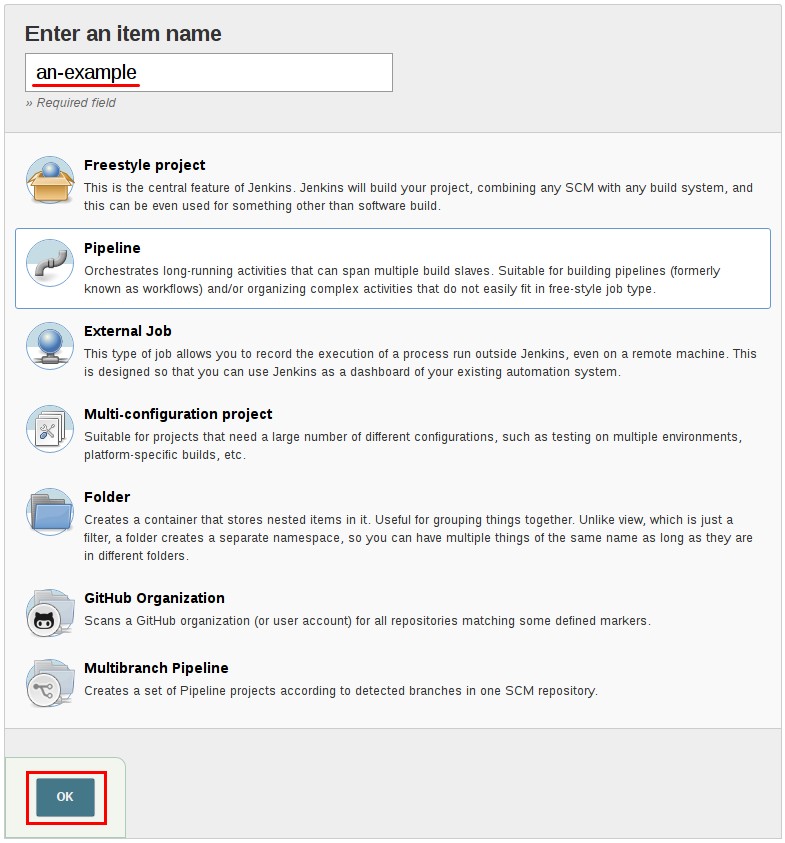
* Click **New Item** on Jenkins home page.



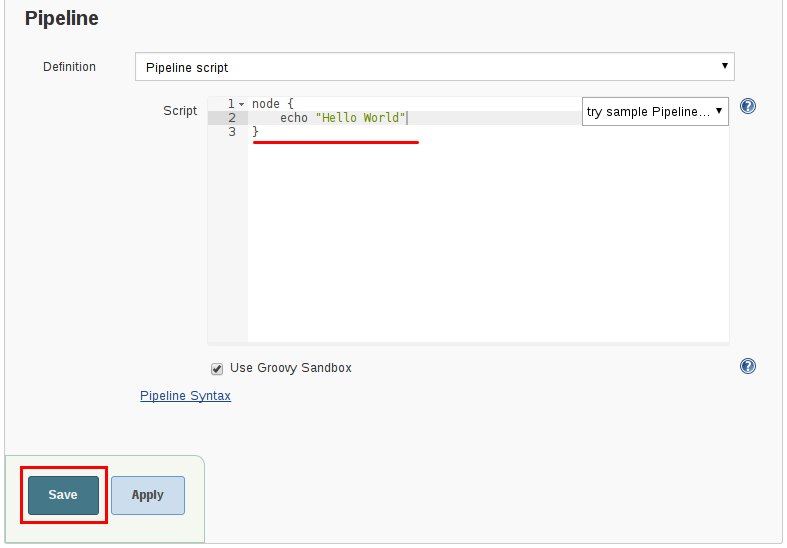
* Enter a name for your Pipeline, select **Pipeline** and click **OK**.

**CAUTION**

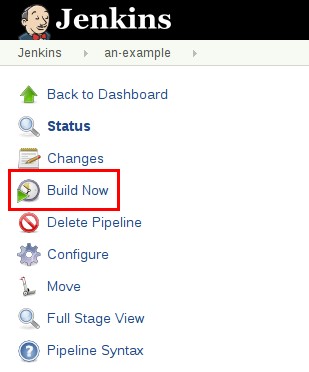
Jenkins uses the name of the Pipeline to create directories on disk. Pipeline names which include spaces may uncover bugs in scripts which do not expect paths to contain spaces.



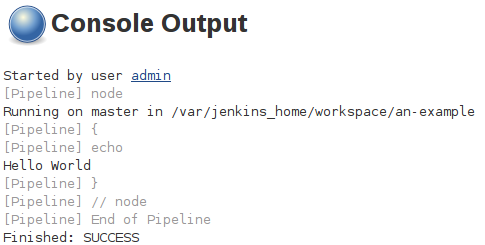
* In the **Script** text area, enter a Pipeline and click **Save**.



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



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



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

// Script // node { ①

echo 'Hello World' ②

}

// Declarative not yet implemented //

① node allocates an executor and workspace in the Jenkins environment.

② 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.

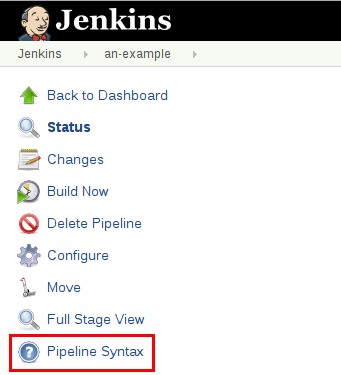
**TIP**

The first line of a Jenkinsfile should be #!/usr/bin/env groovy [14: [en.wikipedia.org/](https://en.wikipedia.org/wiki/Shebang_(Unix)) [wiki/Shebang\_(Unix)](https://en.wikipedia.org/wiki/Shebang_(Unix))] [15: [groovy-lang.org/syntax.html#\_shebang\_line](http://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/, 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**

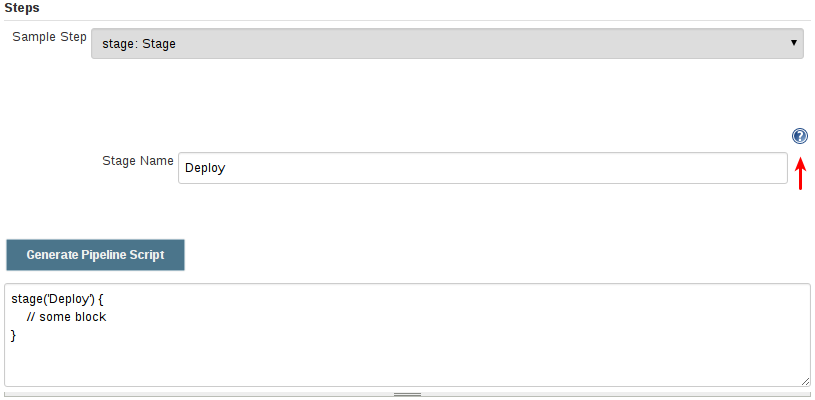
The built-in "Snippet Generator" utility is helpful for creating bits of code for individual steps, discovering new steps provided by plugins, or experimenting with different parameters for a particular step.

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.



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](http://groovy-lang.org/syntax.html#_maps), 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 [16: [en.wikipedia.org/wiki/](https://en.wikipedia.org/wiki/Source_control_management) [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 [17: [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](http://groovy-lang.org/) 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.

// Script // node {

checkout scm ①

/\* .. snip .. \*/

}

// Declarative not yet implemented //

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

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' ①

archiveArtifacts artifacts: '\*\*/target/\*.jar', fingerprint: true ②

}

}

}

}

// Script // node {

stage('Build') { sh 'make' ①

archiveArtifacts artifacts: '\*\*/target/\*.jar', fingerprint: true ②

}

}

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.

**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](https://plugins.jenkins.io/?labels=report). 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 essense, 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](http://groovy-lang.org/) 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](http://groovy-lang.org/syntax.html#_maps), for example:

// Declarative // pipeline {

agent any stages {

stage('Example') { steps {

echo "Running ${env.BUILD\_ID} on ${env.JENKINS\_URL}"

}

}

}

}

// Script // node {

echo "Running ${env.BUILD\_ID} on ${env.JENKINS\_URL}"

}

#### **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.

// Declarative // pipeline {

agent any environment { ①

CC = 'clang'

}

stages {

stage('Example') { environment { ②

DEBUG\_FLAGS = '-g'

}

steps {

sh 'printenv'

}

}

}

}

// Script // node {

/\* .. snip .. \*/ withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {

sh 'mvn -B verify'

}

}

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.

### **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}:

// Declarative // pipeline {

agent any parameters {

string(name: 'Greeting', defaultValue: 'Hello', description: 'How should I greet the world?')

}

stages {

stage('Example') { steps {

echo "${params.Greeting} World!"

}

}

}

}

// Script //

properties([parameters([string(defaultValue: 'Hello', description: 'How should I greet the world?', name: 'Greeting')])])

node {

echo "${params.Greeting} World!"

}

### **Handling Failures**

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.

// Declarative // pipeline {

agent any stages {

stage('Test') { steps {

sh 'make check'

}

}

}

post {

always {

junit '\*\*/target/\*.xml'

}

failure {

[mail to: team@example.com,](mailto:team@example.com) subject: 'The Pipeline failed :('

}

}

}

// Script // node {

/\* .. snip .. \*/

stage('Test') { try {

sh 'make check'

}

finally {

junit '\*\*/target/\*.xml'

}

}

/\* .. snip .. \*/

}

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.

##### // Declarative // 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'

##### }

##### }

##### }

##### }

}

// Script // stage('Build') {

node {

checkout scm sh 'make'

stash includes: '\*\*/target/\*.jar', name: 'app' ①

}

}

stage('Test') {

node('linux') { ② checkout scm try {

unstash 'app' ③

sh 'make check'

}

finally {

junit '\*\*/target/\*.xml'

}

}

node('windows') { checkout scm try {

unstash 'app'

bat 'make check' ④

}

finally {

junit '\*\*/target/\*.xml'

}

}

}

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 [18: [en.wikipedia.org/wiki/Domain-](https://en.wikipedia.org/wiki/Domain-specific_language) [specific\_language](https://en.wikipedia.org/wiki/Domain-specific_language)] based on Groovy, most [Groovy syntax](http://groovy-lang.org/semantics.html) can be used in Scripted Pipeline without modification.

#### **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.

# **Branches and Pull Requests**

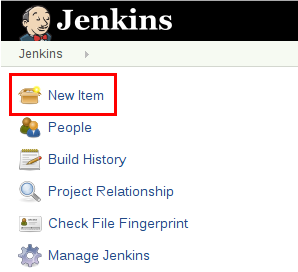
In the previous section a Jenkinsfile which could be checked into source control was implemented. This section covers the concept of **Multibranch** Pipelines which build on the Jenkinsfile foundation to provide more dynamic and automatic functionality in Jenkins.

## **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:

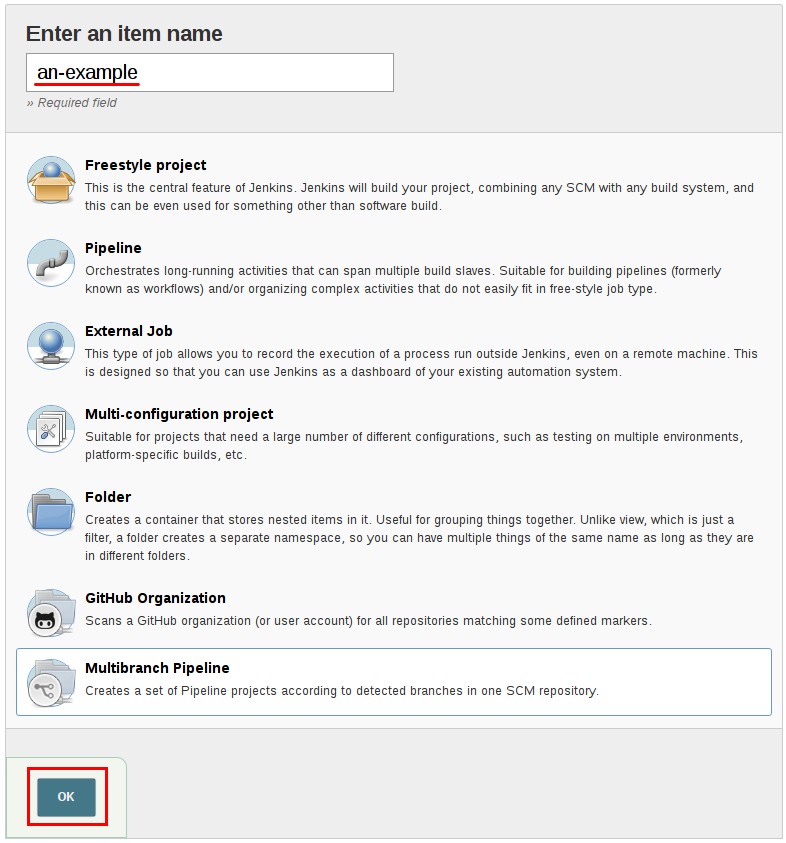
* Click **New Item** on Jenkins home page.



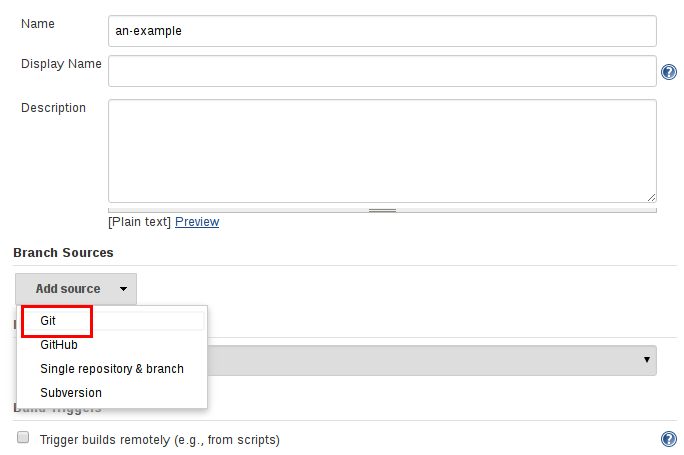
* Enter a name for your Pipeline, select **Multibranch Pipeline** and click **OK**.

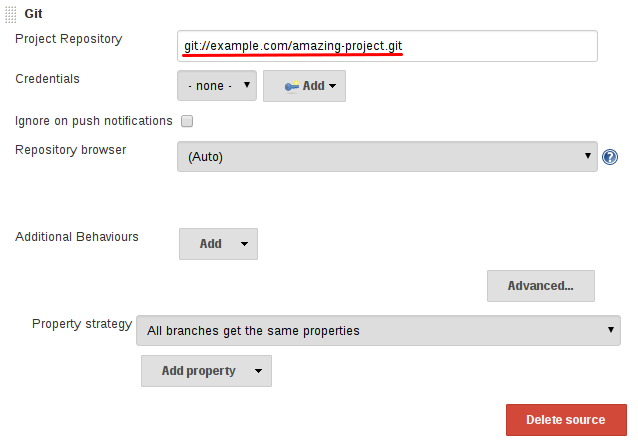
**CAUTION**

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.



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

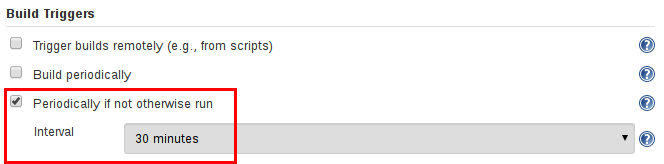




* **Save** the Multibranch Pipeline project.

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](#_bookmark92)), so it is often useful to configure a Multibranch Pipeline to periodically re-index in the configuration:



### **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 plugin:github-branch- source[GitHub Branch Source] and plugin:cloudbees-bitbucket-branch-source[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 plugin:github-organization-folder[GitHub Organization Folder] and plugin:cloudbees-bitbucket- branch-source[Bitbucket Branch Source] plugins.

# **Extending with 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" [19: [en.wikipedia.org/wiki/Don't\_repeat\_yourself](http://en.wikipedia.org/wiki/Don%27t_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**

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

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/](https://svnserver/project/%24%7Blibrary.yourLibName.version%7D)

[${library.yourLibName.version}](https://svnserver/project/%24%7Blibrary.yourLibName.version%7D) 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

| +- foo.txt

+- resources

# for global 'foo' variable

# help for 'foo' variable

# 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).

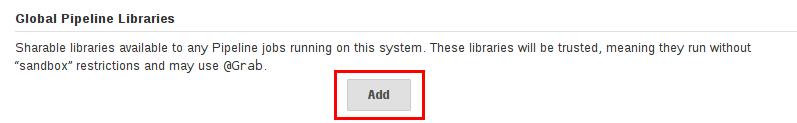
The Groovy source files in these directories get the same “CPS transformation” as in Scripted Pipeline.

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 Pipeline. 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.

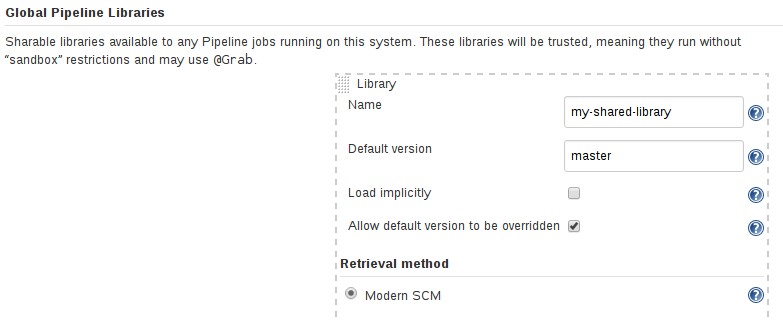
Folder-based libraries are not considered "trusted:" they run in the Groovy sandbox just like typical Pipelines.

### **Automatic Shared Libraries**

Other plugins may add ways of defining libraries on the fly. For example, the plugin:github- organization-folder[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**

Shared Libraries marked *Load implicitly* allows Pipelines to immediately use classes or global variables defined by any such libraries. To access other shared libraries, the Jenkinsfile needs to use the @Library annotation, specifying the library’s name:



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

/\* Using a version specifier, such as branch, tag, etc \*/

@Library('my-shared-library@1.0') \_

/\* Accessing multiple libraries with one statement \*/

@Library(['my-shared-library', '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

**TIP**

For Shared Libraries which only define Global Variables (<code>vars/</code>), or a

<code>Jenkinsfile</code> which only needs a Global Variable, the <a [href="http://groovy-lang.org/objectorientation.html#<em>annotation](http://groovy-lang.org/objectorientation.html)">annotation</a> pattern <code>@Library('my-shared-library') \_</code> may be useful for keeping code concise. In essence, instead of annotating an unnecessary <code>import</code> statement, the symbol <code></em></code> is annotated.

It is not recommended to import a global variable/function, since this will force the compiler to interpret fields and methods as static even if they were intended to be instance. The Groovy compiler in this case can produce confusing error messages.

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.

### **Loading libraries dynamically**

As of version 2.7 of the *Pipeline: Shared Groovy Libraries* plugin, there is a new option for loading (non-implicit) libraries in a script: a library step that loads a library *dynamically*, at any time during the build.

If you are only interested in using global variables/functions (from the vars/ directory), the syntax is quite simple:

library 'my-shared-library'

Thereafter, any global variables from that library will be accessible to the script.

Using classes from the src/ directory is also possible, but trickier. Whereas the @Library annotation prepares the “classpath” of the script prior to compilation, by the time a library step is encountered the script has already been compiled. Therefore you cannot import or otherwise “statically” refer to types from the library.

However you may use library classes dynamically (without type checking), accessing them by fully- qualified name from the return value of the library step. static methods can be invoked using a Java-like syntax:

library('my-shared-library').com.mycorp.pipeline.Utils.someStaticMethod()

You can also access static fields, and call constructors as if they were static methods named new:

def useSomeLib(helper) { // dynamic: cannot declare as Helper helper.prepare()

return helper.count()

}

def lib = library('my-shared-library').com.mycorp.pipeline // preselect the package echo useSomeLib(lib.Helper.new(lib.Constants.SOME\_TEXT))

### **Library versions**

The "Default version" for a configured Shared Library is used when "Load implicitly" is checked, or if a Pipeline references the library only by name, for example @Library('my-shared-library') \_. If a "Default version" is **not** defined, the Pipeline must specify a version, for example @Library('my- shared-library@master') \_.

If "Allow default version to be overridden" is enabled in the Shared Library’s configuration, a

@Library annotation may also override a default version defined for the library. This also allows a library with "Load implicitly" to be loaded from a different version if necessary.

When using the library step you may also specify a version:

library 'my-shared-library@master'

Since this is a regular step, that version could be *computed* rather than a constant as with the annotation; for example:

library "my-shared-library@$BRANCH\_NAME"

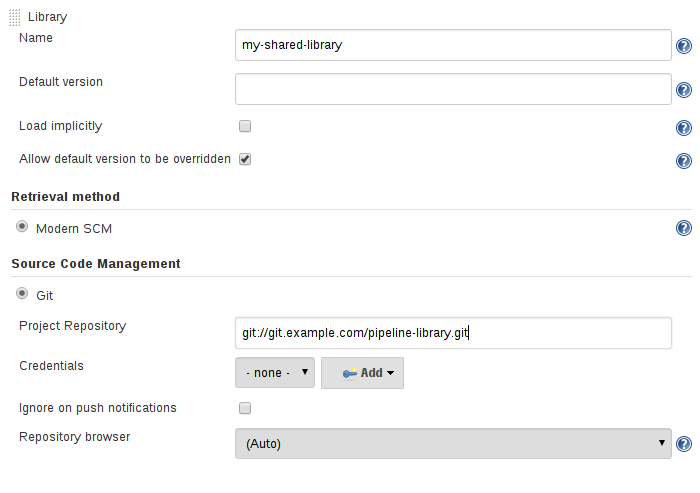
would load a library using the same SCM branch as the multibranch Jenkinsfile. As another example, you could pick a library by parameter:

properties([parameters([string(name: 'LIB\_VERSION', defaultValue: 'master')])]) library "my-shared-library@${params.LIB\_VERSION}"

Note that the library step may not be used to override the version of an implicitly loaded library. It is already loaded by the time the script starts, and a library of a given name may not be loaded twice.

### **Retrieval Method**

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.

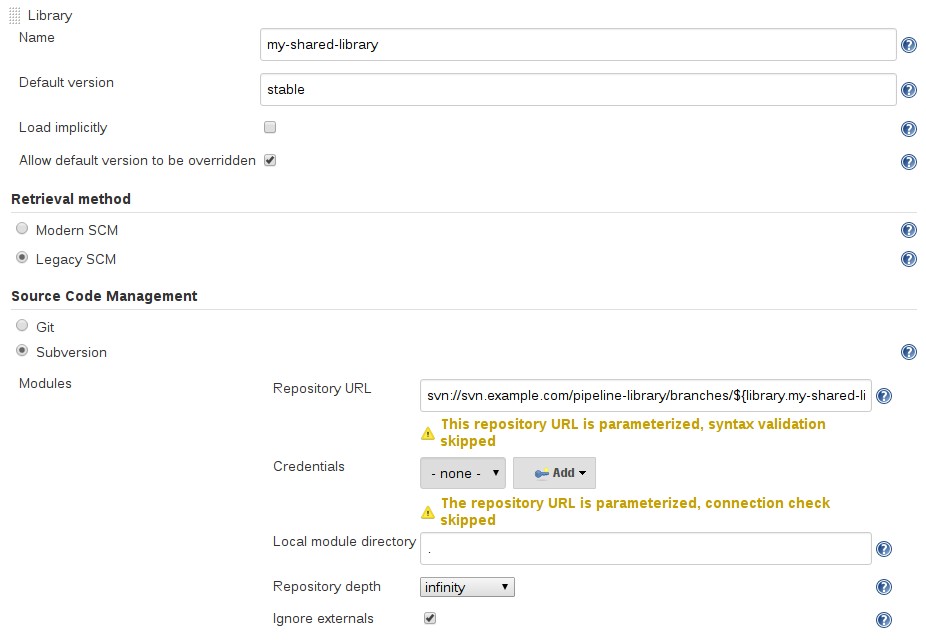


#### **Legacy SCM**

SCM plugins which have not yet been updated to support the newer features required by Shared Libraries, may still be used via the **Legacy SCM** option. In this case, include

${library.yourlibrarynamehere.version} wherever a branch/tag/ref may be configured for that

particular SCM plugin. This ensures that during checkout of the library’s source code, the SCM plugin will expand this variable to checkout the appropriate version of the library.



#### **Dynamic retrieval**

If you only specify a library name (optionally with version after @) in the library step, Jenkins will look for a preconfigured library of that name. (Or in the case of a github.com/owner/repo automatic library it will load that.)

But you may also specify the retrieval method dynamically, in which case there is no need for the library to have been predefined in Jenkins. Here is an example:

library identifier: 'custom-lib@master', retriever: modernSCM( [$class: 'GitSCMSource',

remote: 'git@git.mycorp.com:my-jenkins-utils.git', credentialsId: 'my-private-key'])

It is best to refer to **Pipeline Syntax** for the precise syntax for your SCM. Note that the library version *must* be specified in these cases.

## **Writing libraries**

At the base level, any valid [Groovy code](http://groovy-lang.org/syntax.html) 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) { name = value

}

def getName() { name

}

def caution(message) {

echo "Hello, ${name}! CAUTION: ${message}"

}

In the above, name is not referring to a field (even if you write it as this.name!), but to an entry created on demand in a Script.binding. To be clear about what data you intend to store and of what type, you can instead provide an explicit class declaration (the class name should match the file basename):

// vars/acme.groovy

class acme implements Serializable { private String name

def setName(value) { name = value

}

def getName() { name

}

def caution(message) {

echo "Hello, ${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. [20: [gist.github.com/rtyler/](https://gist.github.com/rtyler/e5e57f075af381fce4ed3ae57aa1f0c2)

[e5e57f075af381fce4ed3ae57aa1f0c2](https://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](http://groovy-lang.org/closures.html). 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 Pipelines that are mostly similar, the global variable mechanism provides 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](http://search.maven.org/), from **trusted** library code using the @Grab annotation. Refer to the [Grape documentation](http://docs.groovy-lang.org/latest/html/documentation/grape.html#_quick_start) 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*.

# **Pipeline Development Tools**

Jenkins Pipeline includes built-in documentation and the Snippet Generator which are key resources when developing Pipelines. They provide detailed help and information that is customized to the currently installed version of Jenkins and related plugins. In this section, we’ll discuss other tools and resources that may help with development of Jenkins Pipelines.

## **Blue Ocean Editor**

The Blue Ocean Pipeline Editor provides a [WYSIWYG](https://en.wikipedia.org/wiki/WYSIWYG) way to create Declarative Pipelines. The editor offers a structural view of all the stages, parallel branches, and steps in a Pipeline. The editor validates Pipeline changes as they are made, eliminating many errors before they are even committed. Behind the scenes it still generates Declarative Pipeline code.

## **Command-line Pipeline Linter**

Jenkins can validate, or "[lint](https://en.wikipedia.org/wiki/Lint_(software))", a Declarative Pipeline from the command line before actually running it. This can be done using a Jenkins CLI command or by making an HTTP POST request with appropriate parameters. We recommended using the SSH interface to run the linter. See the Jenkins CLI documentation for details on how to properly configure Jenkins for secure command- line access.

*Linting via the CLI with SSH*

# ssh (Jenkins CLI)

# JENKINS\_SSHD\_PORT=[sshd port on master]

# JENKINS\_HOSTNAME=[Jenkins master hostname]

ssh -p $JENKINS\_SSHD\_PORT $JENKINS\_HOSTNAME declarative-linter < Jenkinsfile

*Linting via HTTP POST using* curl

# curl (REST API)

# Assuming "anonymous read access" has been enabled on your Jenkins instance.

# JENKINS\_URL=[root URL of Jenkins master]

# JENKINS\_CRUMB is needed if your Jenkins master has CRSF protection enabled as it should

JENKINS\_CRUMB=`curl "$JENKINS\_URL/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,\":\",//crumb)"` curl -X POST -H $JENKINS\_CRUMB -F "jenkinsfile=<Jenkinsfile" $JENKINS\_URL/pipeline- model-converter/validate

### **Examples**

Below are two examples of the Pipeline Linter in action. This first example shows the output of the linter when it is passed an invalid Jenkinsfile, one that is missing part of the agent declaration.

*Jenkinsfile*

pipeline { agent stages {

stage ('Initialize') { steps {

echo 'Placeholder.'

}

}

}

}

*Linter output for invalid Jenkinsfile*

# pass a Jenkinsfile that does not contain an "agent" section ssh -p 8675 localhost declarative-linter < ./Jenkinsfile Errors encountered validating Jenkinsfile:

WorkflowScript: 2: Not a valid section definition: "agent". Some extra configuration is required. @ line 2, column 3.

agent

^

WorkflowScript: 1: Missing required section "agent" @ line 1, column 1. pipeline &#125;

^

In this second example, the Jenkinsfile has been updated to include the missing any on agent. The linter now reports that the Pipeline is valid.

*Jenkinsfile*

pipeline { agent any stages {

stage ('Initialize') { steps {

echo 'Placeholder.'

}

}

}

}

*Linter output for valid Jenkinsfile*

ssh -p 8675 localhost declarative-linter < ./Jenkinsfile Jenkinsfile successfully validated.

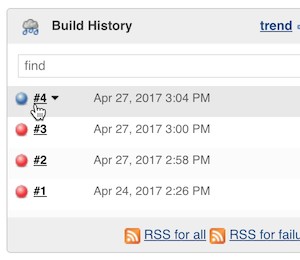
## **"Replay" Pipeline Runs with Modifications**

Typically a Pipeline will be defined inside of the classic Jenkins web UI, or by committing to a Jenkinsfile in source control. Unfortunately, neither approach is ideal for rapid iteration, or prototyping, of a Pipeline. The "Replay" feature allows for quick modifications and execution of an existing Pipeline without changing the Pipeline configuration or creating a new commit.

### **Usage**

To use the "Replay" feature:

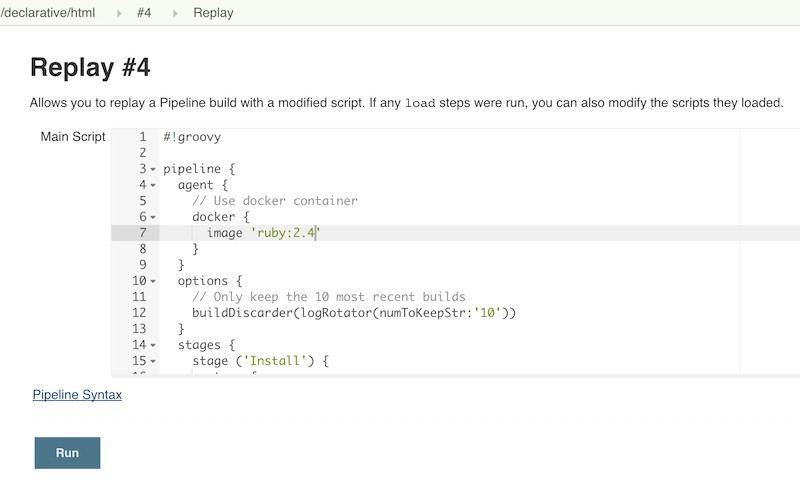
1. Select a previously completed run in the build history.



1. Click "Replay" in the left menu



1. Make modifications and click "Run". In this example, we changed "ruby-2.3" to "ruby-2.4".



1. Check the results of changes

Once you are satisfied with the changes, you can use Replay to view them again, copy them back to your Pipeline job or Jenkinsfile, and then commit them using your usual engineering processes.

### **Features**

* **Can be called multiple times on the same run** - allows for easy parallel testing of different changes.
* **Can also be called on Pipeline runs that are still in-progress** - As long as a Pipeline contained syntactically correct Groovy and was able to start, it can be Replayed.
* **Referenced Shared Library code is also modifiable** - If a Pipeline run references a Shared Library, the code from the shared library will also be shown and modifiable as part of the Replay page.

### **Limitations**

* **Pipeline runs with syntax errors cannot be replayed** - meaning their code cannot be viewed and any changes made in them cannot be retrieved. When using Replay for more significant modifications, save your changes to a file or editor outside of Jenkins before running them. See [JENKINS-37589](https://issues.jenkins-ci.org/browse/JENKINS-37589)
* **Replayed Pipeline behavior may differ from runs started by other methods** - For Pipelines that are not part of a Multi-branch Pipeline, the commit information may differ for the original run and the Replayed run. See [JENKINS-36453](https://issues.jenkins-ci.org/browse/JENKINS-36453)

## **Pipeline Unit Testing Framework**

**NOTE**

The Pipeline Unit Testing Framework is a third-party tool that is not supported by the Jenkins Project.

The [Pipeline Unit Testing Framework](https://github.com/lesfurets/JenkinsPipelineUnit) allows you to [unit test](https://en.wikipedia.org/wiki/Unit_testing) Pipelines and Shared Libraries before running them in full. It provides a mock execution environment where real Pipeline steps are replaced with mock objects that you can use to check for expected behavior. New and rough around the edges, but promising. The [README](https://github.com/lesfurets/JenkinsPipelineUnit/blob/master/README.md) for that project contains examples and usage instructions.

# **Pipeline Syntax**

This section builds on the information introduced in Getting Started, and should be treated solely as a reference. For more information on how to use Pipeline syntax in practical examples, refer to The Jenkinsfile section of this chapter. As of version 2.5 of the Pipeline plugin, Pipeline supports two discrete syntaxes which are detailed below. For the pros and cons of each, see the [Syntax](#_bookmark133) [Comparison](#_bookmark133).

As discussed in Getting Started, the most fundamental part of a Pipeline is the "step." Basically, steps tell Jenkins *what* to do, and serve as the basic building block for both Declarative and Scripted Pipeline syntax.

For an overview of available steps, please refer to the Pipeline Steps reference which contains a comprehensive list of steps built into Pipeline as well as steps provided by plugins.

## **Declarative Pipeline**

Declarative Pipeline is a relatively recent addition to Jenkins Pipeline [21: Version 2.5 of the "Pipeline plugin" introduces support for Declarative Pipeline syntax] which presents a more simplified and opinionated syntax on top of the Pipeline sub-systems.

All valid Declarative Pipelines must be enclosed within a pipeline block, for example:

pipeline {

/\* insert Declarative Pipeline here \*/

}

The basic statements and expressions which are valid in Declarative Pipeline follow the same rules as [Groovy’s syntax](http://groovy-lang.org/syntax.html) with the following exceptions:

* The top-level of the Pipeline must be a *block*, specifically: pipeline { }
* No semicolons as statement separators. Each statement has to be on its own line
* Blocks must only consist of [Sections](#_bookmark122), [Directives](#_bookmark125), [Steps](#_bookmark128), or assignment statements.
* A property reference statement is treated as no-argument method invocation. So for example, input is treated as input()

### **Sections**

Sections in Declarative Pipeline typically contain one or more [Directives](#_bookmark125) or [Steps](#_bookmark128).

#### **agent**

The agent section specifies where the entire Pipeline, or a specific stage, will execute in the Jenkins environment depending on where the agent section is placed. The section must be defined at the top-level inside the pipeline block, but stage-level usage is optional.

|  |  |
| --- | --- |
| **Require d** | Yes |
| **Parame ters** | [Described below](#_bookmark123) |
| **Allowed** | In the top-level pipeline block and each stage block. |

**Parameters**

In order to support the wide variety of use-cases Pipeline authors may have, the agent section supports a few different types of parameters. These parameters can be applied at the top-level of the pipeline block, or within each stage directive.

**any**

Execute the Pipeline, or stage, on any available agent. For example: agent any

**none**

When applied at the top-level of the pipeline block no global agent will be allocated for the entire Pipeline run and each stage section will need to contain its own agent section. For example: agent none

**label**

Execute the Pipeline, or stage, on an agent available in the Jenkins environment with the provided label. For example: agent { label 'my-defined-label' }

**node**

agent { node { label 'labelName' } } behaves the same as agent { label 'labelName' }, but

node allows for additional options (such as customWorkspace).

**docker**

Execute the Pipeline, or stage, with the given container which will be dynamically provisioned on a node pre-configured to accept Docker-based Pipelines, or on a node matching the optionally defined label parameter. docker also optionally accepts an args parameter which may contain arguments to pass directly to a docker run invocation. For example: agent { docker 'maven:3- alpine' } or

agent {

docker {

image 'maven:3-alpine' label 'my-defined-label' args '-v /tmp:/tmp'

}

}

**dockerfile**

Execute the Pipeline, or stage, with a container built from a Dockerfile contained in the source repository. Conventionally this is the Dockerfile in the root of the source repository: agent { dockerfile true }. If building a Dockerfile in another directory, use the dir option: agent { dockerfile { dir 'someSubDir' } }. You can pass additional arguments to the docker build … command with the additionalBuildArgs option, like agent { dockerfile { additionalBuildArgs '--build-arg foo=bar' } }.

**Common Options**

These are a few options that can be applied two or more agent implementations. They are not required unless explicitly stated.

**label**

A string. The label on which to run the Pipeline or individual stage.

This option is valid for node, docker and dockerfile, and is required for node.

**customWorkspace**

A string. Run the Pipeline or individual stage this agent is applied to within this custom

workspace, rather than the default. It can be either a relative path, in which case the custom workspace will be under the workspace root on the node, or an absolute path. For example:

agent {

node {

label 'my-defined-label' customWorkspace '/some/other/path'

}

}

This option is valid for node, docker and dockerfile.

**reuseNode**

A boolean, false by default. If true, run the container on the node specified at the top-level of the Pipeline, in the same workspace, rather than on a new node entirely.

This option is valid for docker and dockerfile, and only has an effect when used on an agent for an individual stage.

**Example**

// Declarative // pipeline {

agent { docker 'maven:3-alpine' } ①

stages {

stage('Example Build') { steps {

sh 'mvn -B clean verify'

}

}

}

}

// Script //

① Execute all the steps defined in this Pipeline within a newly created container of the given name and tag (maven:3-alpine).

**Stage-level** agent **section**

// Declarative // pipeline {

agent none ①

stages {

stage('Example Build') {

agent { docker 'maven:3-alpine' } ②

steps {

echo 'Hello, Maven' sh 'mvn --version'

}

}

stage('Example Test') {

agent { docker 'openjdk:8-jre' } ③

steps {

echo 'Hello, JDK' sh 'java -version'

}

}

}

}

// Script //

① Defining agent none at the top-level of the Pipeline ensures that an Executor will not be assigned unnecessarily. Using agent none also forces each stage section contain its own agent section.

② Execute the steps in this stage in a newly created container using this image.

③ Execute the steps in this stage in a newly created container using a different image from the previous stage.

#### **post**

The post section defines actions which will be run at the end of the Pipeline run or stage. A number of [post-condition](#_bookmark124) blocks are supported within the post section: always, changed, failure, success, and unstable. These blocks allow for the execution of steps at the end of the Pipeline run or stage, depending on the status of the Pipeline.

|  |  |
| --- | --- |
| **Require d** | No |
| **Parame ters** | *None* |
| **Allowed** | In the top-level pipeline block and each stage block. |

**Conditions**

always

Run regardless of the completion status of the Pipeline run.

changed

Only run if the current Pipeline run has a different status from the previously completed Pipeline.

failure

Only run if the current Pipeline has a "failed" status, typically denoted in the web UI with a red indication.

success

Only run if the current Pipeline has a "success" status, typically denoted in the web UI with a blue or green indication.

unstable

Only run if the current Pipeline has an "unstable" status, usually caused by test failures, code violations, etc. Typically denoted in the web UI with a yellow indication.

**Example**

// Declarative // pipeline {

agent any stages {

stage('Example') { steps {

echo 'Hello World'

}

}

}

post { ①

always { ②

echo 'I will always say Hello again!'

}

}

}

// Script //

① Conventionally, the post section should be placed at the end of the Pipeline.

② [Post-condition](#_bookmark124) blocks contain [steps](#_bookmark128) the same as the [steps] section.

#### **stages**

Containing a sequence of one or more [[stage]](#_bookmark64) directives, the stages section is where the bulk of the "work" described by a Pipeline will be located. At a minimum it is recommended that stages contain at least one [[stage]](#_bookmark64) directive for each discrete part of the continuous delivery process, such as Build, Test, and Deploy.

**Require d**

Yes

|  |  |
| --- | --- |
| **Parame ters** | *None* |
| **Allowed** | Only once, inside the pipeline block. |

**Example**

// Declarative // pipeline {

agent any stages { ①

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

// Script //

① The stages section will typically follow the directives such as agent, options, etc.

#### **steps**

The steps section defines a series of one or more [steps](#_bookmark128) to be executed in a given stage directive.

|  |  |
| --- | --- |
| **Require d** | Yes |
| **Parame ters** | *None* |
| **Allowed** | Inside each stage block. |

**Example**

// Declarative // pipeline {

agent any stages {

stage('Example') { steps { ①

echo 'Hello World'

}

}

}

}

// Script //

① The steps section must contain one or more steps.

### **Directives**

#### **environment**

The environment directive specifies a sequence of key-value pairs which will be defined as environment variables for the all steps, or stage-specific steps, depending on where the environment directive is located within the Pipeline.

This directive supports a special helper method credentials() which can be used to access pre- defined Credentials by their identifier in the Jenkins environment. For Credentials which are of type "Secret Text", the credentials() method will ensure that the environment variable specified contains the Secret Text contents. For Credentials which are of type "Standard username and password", the environment variable specified will be set to username:password and two additional environment variables will be automatically be defined: MYVARNAME\_USR and MYVARNAME\_PSW respective.

|  |  |
| --- | --- |
| **Require d** | No |
| **Parame ters** | *None* |
| **Allowed** | Inside the pipeline block, or within stage directives. |

**Example**

// Declarative // pipeline {

agent any environment { ①

CC = 'clang'

}

stages {

stage('Example') { environment { ②

AN\_ACCESS\_KEY = credentials('my-prefined-secret-text') ③

}

steps {

sh 'printenv'

}

}

}

}

// Script //

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.

③ The environment block has a helper method credentials() defined which can be used to access pre-defined Credentials by their identifier in the Jenkins environment.

#### **options**

The options directive allows configuring Pipeline-specific options from within the Pipeline itself. Pipeline provides a number of these options, such as buildDiscarder, but they may also be provided by plugins, such as timestamps.

|  |  |
| --- | --- |
| **Require d** | No |
| **Parame ters** | *None* |
| **Allowed** | Only once, inside the pipeline block. |

**Available Options**

**buildDiscarder**

Persist artifacts and console output for the specific number of recent Pipeline runs. For example:

options { buildDiscarder(logRotator(numToKeepStr: '1')) }

**disableConcurrentBuilds**

Disallow concurrent executions of the Pipeline. Can be useful for preventing simultaneous accesses to shared resources, etc. For example: options { disableConcurrentBuilds() }

**skipDefaultCheckout**

Skip checking out code from source control by default in the agent directive. For example:

options { skipDefaultCheckout() }

**skipStagesAfterUnstable**

Skip stages once the build status has gone to UNSTABLE. For example: options { skipStagesAfterUnstable() }

**timeout**

Set a timeout period for the Pipeline run, after which Jenkins should abort the Pipeline. For example: options { timeout(time: 1, unit: 'HOURS') }

**retry**

On failure, retry the entire Pipeline the specified number of times. For example: options { retry(3) }

**timestamps**

Prepend all console output generated by the Pipeline run with the time at which the line was emitted. For example: options { timestamps() }

**Example**

// Declarative // pipeline {

agent any options {

timeout(time: 1, unit: 'HOURS') ①

}

stages {

stage('Example') { steps {

echo 'Hello World'

}

}

}

}

// Script //

① Specifying a global execution timeout of one hour, after which Jenkins will abort the Pipeline run.

**NOTE** A comprehensive list of available options is pending the completion of [INFRA-1503](https://issues.jenkins-ci.org/browse/INFRA-1053).

#### **parameters**

The parameters directive provides a list of parameters which a user should provide when triggering the Pipeline. The values for these user-specified parameters are made available to Pipeline steps via the params object, see the [Example](#_bookmark126) for its specific usage.

|  |  |
| --- | --- |
| **Require d** | No |
| **Parame ters** | *None* |
| **Allowed** | Only once, inside the pipeline block. |

**Available Parameters**

**string**

A parameter of a string type, for example: parameters { string(name: 'DEPLOY\_ENV', defaultValue: 'staging', description: '') }

**booleanParam**

A boolean parameter, for example: parameters { booleanParam(name: 'DEBUG\_BUILD', defaultValue: true, description: '') }

**Example**

// Declarative // pipeline {

agent any parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

}

stages {

stage('Example') { steps {

echo "Hello ${params.PERSON}"

}

}

}

}

// Script //

**NOTE**

A comprehensive list of available parameters is pending the completion of [INFRA-](https://issues.jenkins-ci.org/browse/INFRA-1053) [1503](https://issues.jenkins-ci.org/browse/INFRA-1053).

#### **triggers**

The triggers directive defines the automated ways in which the Pipeline should be re-triggered. For Pipelines which are integrated with a source such as GitHub or BitBucket, triggers may not be necessary as webhooks-based integration will likely already be present. Currently the only two available triggers are cron and pollSCM.

|  |  |
| --- | --- |
| **Require d** | No |
| **Parame ters** | *None* |
| **Allowed** | Only once, inside the pipeline block. |

**cron**

Accepts a cron-style string to define a regular interval at which the Pipeline should be re- triggered, for example: triggers { cron('H 4/\* 0 0 1-5') }

**pollSCM**

Accepts a cron-style string to define a regular interval at which Jenkins should check for new source changes. If new changes exist, the Pipeline will be re-triggered. For example: triggers { pollSCM('H 4/\* 0 0 1-5') }

**NOTE** The pollSCM trigger is only available in Jenkins 2.22 or later.

**Example**

// Declarative // pipeline {

agent any triggers {

cron('H 4/\* 0 0 1-5')

}

stages {

stage('Example') { steps {

echo 'Hello World'

}

}

}

}

// Script //

#### **stage**

The stage directive goes in the stages section and should contain a [steps] section, an optional agent section, or other stage-specific directives. Practically speaking, all of the real work done by a Pipeline will be wrapped in one or more stage directives.

|  |  |
| --- | --- |
| **Require d** | At least one |
| **Parame ters** | One mandatory parameter, a string for the name of the stage. |
| **Allowed** | Inside the stages section. |

**Example**

// Declarative // pipeline {

agent any stages {

stage('Example') { steps {

echo 'Hello World'

}

}

}

}

// Script //

#### **tools**

A section defining tools to auto-install and put on the PATH. This is ignored if agent none is specified.

|  |  |
| --- | --- |
| **Require d** | No |
| **Parame ters** | *None* |
| **Allowed** | Inside the pipeline block or a stage block. |

**Supported Tools**

**maven jdk gradle**

**Example**

// Declarative // pipeline {

agent any tools {

maven 'apache-maven-3.0.1' ①

}

stages {

stage('Example') { steps {

sh 'mvn --version'

}

}

}

}

// Script //

① The tool name must be pre-configured in Jenkins under **Manage Jenkins** → **Global Tool Configuration**.

#### **when**

The when directive allows the Pipeline to determine whether the stage should be executed depending on the given condition. The when directive must contain at least one condition. If the when directive contains more than one condition, all the child conditions must return true for the stage to execute. This is the same as if the child conditions were nested in an allOf condition (see the [examples](#_bookmark127) below).

More complex conditional structures can be built using the nesting conditions: not, allOf, or anyOf. Nesting conditions may be nested to any arbitrary depth.

|  |  |
| --- | --- |
| **Require d** | No |
| **Parame ters** | *None* |
| **Allowed** | Inside a stage directive |

**Built-in Conditions**

**branch**

Execute the stage when the branch being built matches the branch pattern given, for example:

when { branch 'master' }. Note that this only works on a multibranch Pipeline.

**environment**

Execute the stage when the specified environment variable is set to the given value, for example:

when { environment name: 'DEPLOY\_TO', value: 'production' }

**expression**

Execute the stage when the specified Groovy expression evaluates to true, for example: when { expression { return params.DEBUG\_BUILD } }

**not**

Execute the stage when the nested condition is false. Must contain one condition. For example:

when { not { branch 'master' } }

**allOf**

Execute the stage when all of the nested conditions are true. Must contain at least one condition. For example: when { allOf { branch 'master'; environment name: 'DEPLOY\_TO', value: 'production' } }

**anyOf**

Execute the stage when at least one of the nested conditions is true. Must contain at least one condition. For example: when { anyOf { branch 'master'; branch 'staging' } }

**Examples**

*Single condition*

// Declarative // pipeline {

agent any stages {

stage('Example Build') { steps {

echo 'Hello World'

}

}

stage('Example Deploy') { when {

branch 'production'

}

steps {

echo 'Deploying'

}

}

}

}

// Script //

*Multiple condition*

// Declarative // pipeline {

agent any stages {

stage('Example Build') { steps {

echo 'Hello World'

}

}

stage('Example Deploy') { when {

branch 'production'

environment name: 'DEPLOY\_TO', value: 'production'

}

steps {

echo 'Deploying'

}

}

}

}

// Script //

*Nested condition (same behavior as previous example)*

// Declarative // pipeline {

agent any stages {

stage('Example Build') { steps {

echo 'Hello World'

}

}

stage('Example Deploy') { when {

allOf {

branch 'production'

environment name: 'DEPLOY\_TO', value: 'production'

}

}

steps {

echo 'Deploying'

}

}

}

}

// Script //

*Multiple condition and nested condition*

// Declarative // pipeline {

agent any stages {

stage('Example Build') { steps {

echo 'Hello World'

}

}

stage('Example Deploy') { when {

branch 'production' anyOf {

environment name: 'DEPLOY\_TO', value: 'production' environment name: 'DEPLOY\_TO', value: 'staging'

}

}

steps {

echo 'Deploying'

}

}

}

}

// Script //

*Expression condition and nested condition*

// Declarative // pipeline {

agent any stages {

stage('Example Build') { steps {

echo 'Hello World'

}

}

stage('Example Deploy') { when {

expression { BRANCH\_NAME ==~ /(production|staging)/ } anyOf {

environment name: 'DEPLOY\_TO', value: 'production' environment name: 'DEPLOY\_TO', value: 'staging'

}

}

steps {

echo 'Deploying'

}

}

}

}

// Script //

### **Steps**

Declarative Pipelines may use all the available steps documented in the Pipeline Steps reference, which contains a comprehensive list of steps, with the addition of the steps listed below which are **only supported** in Declarative Pipeline.

#### **script**

The script step takes a block of [scripted-pipeline] and executes that in the Declarative Pipeline. For most use-cases, the script step should be unnecessary in Declarative Pipelines, but it can provide a useful "escape hatch." script blocks of non-trivial size and/or complexity should be moved into Shared Libraries instead.

**Example**

// Declarative // pipeline {

agent any stages {

stage('Example') { steps {

echo 'Hello World'

script {

def browsers = ['chrome', 'firefox']

for (int i = 0; i < browsers.size(); ++i) { echo "Testing the ${browsers[i]} browser"

}

}

}

}

}

}

// Script //

## **Scripted Pipeline**

Scripted Pipeline, like [declarative-pipeline], is built on top of the underlying Pipeline sub-system. Unlike Declarative, Scripted Pipeline is effectively a general purpose DSL [22: Domain-specific Language] built with [Groovy](http://groovy-lang.org/syntax.html). Most functionality provided by the Groovy language is made available to users of Scripted Pipeline, which means it can be a very expressive and flexible tool with which one can author continuous delivery pipelines.

### **Flow Control**

Scripted Pipeline is serially executed from the top of a Jenkinsfile downwards, like most traditional scripts in Groovy or other languages. Providing flow control therefore rests on Groovy expressions, such as the if/else conditionals, for example:

// Scripted // node {

stage('Example') {

if (env.BRANCH\_NAME == 'master') {

echo 'I only execute on the master branch'

} else {

echo 'I execute elsewhere'

}

}

}

// Declarative //

Another way Scripted Pipeline flow control can be managed is with Groovy’s exception handling support. When [Steps](#_bookmark131) fail for whatever reason they throw an exception. Handling behaviors on- error must make use of the try/catch/finally blocks in Groovy, for example:

// Scripted // node {

stage('Example') { try {

sh 'exit 1'

}

catch (exc) {

echo 'Something failed, I should sound the klaxons!' throw

}

}

}

// Declarative //

# **Backing-up/Restoring Jenkins**

# **Monitoring Jenkins**

# **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](https://wiki.jenkins-ci.org/display/JENKINS/Quick%2Band%2BSimple%2BSecurity) --- if you are running Jenkins like java -jar jenkins.war and only need a very simple set up
* [Standard Security Setup](https://wiki.jenkins-ci.org/display/JENKINS/Standard%2BSecurity%2BSetup) --- discusses the most common set up of letting Jenkins run its own user database and do finer-grained access control
* [Apache frontend for security](https://wiki.jenkins-ci.org/display/JENKINS/Apache%2Bfrontend%2Bfor%2Bsecurity) --- run Jenkins behind Apache and perform access control in Apache instead of Jenkins
* [Authenticating scripted clients](https://wiki.jenkins-ci.org/display/JENKINS/Authenticating%2Bscripted%2Bclients) --- if you need to programatically access security-enabled Jenkins web UI, use BASIC auth
* [Matrix-based security|Matrix-based security](https://wiki.jenkins-ci.org/display/JENKINS/Matrix-based%2Bsecurity) --- 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](https://wiki.jenkins-ci.org/display/JENKINS/CSRF%2BProtection) --- prevent a remote attack against Jenkins running inside your firewall
* [Security implication of building on master](https://wiki.jenkins-ci.org/display/JENKINS/Security%2Bimplication%2Bof%2Bbuilding%2Bon%2Bmaster) --- protect Jenkins master from malicious builds
* [Slave To Master Access Control](https://wiki.jenkins-ci.org/display/JENKINS/Slave%2BTo%2BMaster%2BAccess%2BControl) --- 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](https://wiki.jenkins-ci.org/display/JENKINS/Configuring%2BContent%2BSecurity%2BPolicy) --- protect users of Jenkins from malicious builds
* [Markup formatting](https://wiki.jenkins-ci.org/display/JENKINS/Markup%2Bformatting) --- 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 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 stopthe 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, trying renaming or deleting config.xml.

# **Managing Jenkins with Chef**

# **Managing Jenkins with Puppet**

# **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.

## **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](#_bookmark216) or [Pipeline](#_bookmark222) run which is **archived** onto the Jenkins [Master](#_bookmark219) for later retrieval by users.

**Build**

Result of a single execution of a [Project](#_bookmark221)

**Cloud**

A System Configuration which provides dynamic [Agent](#_bookmark214) provisioning and allocation, such as that provided by the plugin:azure-vm-agents[Azure VM Agents] or plugin:ec2[Amazon EC2] plugins.

**Core**

The primary Jenkins application (jenkins.war) which provides the basic web UI, configuration, and foundation upon which [Plugins](#_bookmark223) can be built.

**Downstream**

A configured [Pipeline](#_bookmark222) or [Project](#_bookmark221) 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](#_bookmark222) or [Project](#_bookmark221) on a [Node](#_bookmark220). 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](#_bookmark215) or other entity across multiple [Pipelines](#_bookmark222) or [Projects](#_bookmark221).

**Folder**

An organizational container for [Pipelines](#_bookmark222) and/or [Projects](#_bookmark221), similar to folders on a file system.

**Item**

An entity in the web UI corresponding to either a: [Folder](#_bookmark218), [Pipeline](#_bookmark222), or [Project](#_bookmark221).

**Job**

A deprecated term, synonymous with [Project](#_bookmark221).

**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](#_bookmark222) or [Projects](#_bookmark221). Both the [Master](#_bookmark219) and [Agents](#_bookmark214) 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](#_bookmark217).

**Publisher**

Part of a [Build](#_bookmark216) after the completion of all configured [Steps](#_bookmark224) which publishes reports, sends notifications, etc.

**Step**

A single task; fundamentally steps tell Jenkins *what* to do inside of a [Pipeline](#_bookmark222) or [Project](#_bookmark221).

**Trigger**

A criteria for triggering a new [Pipeline](#_bookmark222) run or [Build](#_bookmark216).

**Update Center**

Hosted inventory of plugins and plugin metadata to enable plugin installation from within Jenkins.

**Upstream**

A configured [Pipeline](#_bookmark222) or [Project](#_bookmark221) which triggers a separate Pipeline or Project as part of its execution.

**Workspace**

A disposable directory on the file system of a [Node](#_bookmark220) where work can be done by a [Pipeline](#_bookmark222) or [Project](#_bookmark221). Workspaces are typically left in place after a [Build](#_bookmark216) or [Pipeline](#_bookmark222) run completes unless specific Workspace cleanup policies have been put in place on the Jenkins [Master](#_bookmark219).