

Jenkins User Handbook

jenkinsci-docs@googlegroups.com

Table of Contents

Getting Started with Jenkins	i1
Installing Jenkins	i2
Overview	i3
Pre-install	i4
System Requirements	i4
Experimentation, Staging, or Production?	i4
Stand-alone or Servlet?	i4
Installation	i5
Unix/Linux	i5
OS X	i5
Windows	i6
Docker	i6
Other	i6
Post-install (Setup Wizard)	i7
Create Admin User and Password for Jenkins	i7
Initial Plugin Installation	i7
Using Jenkins	i8
Fingerprints	i9
Remote API	i10
Security	i11
CSRF	i11
Managing Jenkins	i12
Configuring the System	i13
Managing Security	i14
Enabling Security	i15
JNLP TCP Port	i15
Access Control	i16
Markup Formatter	i18
Cross Site Request Forgery	i19
Caveats	i19
Agent/Master Access Control	i20
Customizing Access	i20
Disabling	i22
Managing Tools	i24
Built-in tool providers	i25
Ant	i25
Git	i25
JDK	i25

Maven	25
Managing Plugins.....	26
Installing a plugin	27
From the web UI	27
Using the Jenkins CLI	28
Advanced installation	28
Updating a plugin	30
Removing a plugin	31
Uninstalling a plugin	31
Disabling a plugin	32
Pinned plugins	33
Jenkins CLI.....	34
Using the CLI.....	35
Authentication.....	35
Common Commands	36
Using the CLI client	39
Downloading the client	39
Using the client	39
Common Problems	39
Script Console	41
Managing Nodes.....	42
Managing Users	43
Best Practices	44
Pipeline.....	45
What is Pipeline?.....	46
Why Pipeline?	48
Scripted Pipeline Syntax and Declarative Pipeline Syntax	49
Pipeline Terms	50
Getting Started	51
Prerequisites.....	52
Defining a Pipeline	53
Defining a Pipeline in the Web UI.....	53
Defining a Pipeline in SCM	56
Built-in Documentation	57
Snippet Generator.....	57
Global Variable Reference	58
Further Reading.....	59
Additional Resources	59
Multibranch Pipelines.....	60
Creating a Multibranch Pipeline	61
Additional Environment Variables.....	64

Supporting Pull Requests	64
Using Organization Folders.....	65
The Jenkinsfile	66
Creating a Jenkinsfile	67
Build	68
Test	68
Deploy	69
Advanced Syntax for Scripted Pipeline	71
String Interpolation	71
Working with the Environment	71
Build Parameters.....	72
Handling Failures	72
Using multiple nodes	73
Executing in parallel	74
Optional step arguments.....	75
Docker Pipeline Plugin.....	77
Introduction	77
Running build steps inside containers.....	77
Customizing agent allocation.....	79
Building and publishing images	79
Running and testing containers	81
Specifying a custom registry and server.....	81
Advanced usage.....	82
Demonstrations	82
Shared Libraries.....	83
Defining Shared Libraries.....	84
Directory structure	84
Global Shared Libraries.....	85
Folder-level Shared Libraries	85
Automatic Shared Libraries	85
Using libraries	86
Overriding versions	86
Writing libraries	87
Accessing steps	87
Defining global variables	88
Defining steps	89
Defining a more structured DSL	90
Using third-party libraries	91
Loading resources.....	91
Pretesting library changes	91
Plugin Developer Guide	93

Extension points accessible via metastep	94
General guidelines	94
SCMs	99
Build steps	100
Build wrappers	102
Triggers	103
Clouds	104
Custom steps	105
Historical background	106
Jenkins Use-Cases	107
Jenkins with .NET	108
Jenkins with Java	109
Jenkins with Python	110
Test Reports	111
Jenkins with Ruby	112
Test Reports	113
Coverage Reports	114
Rails	115
Operating Jenkins	116
Backing-up/Restoring Jenkins	117
Monitoring Jenkins	118
Securing Jenkins	119
Access Control	120
Protect users of Jenkins from other threats	121
Disabling Security	122
Managing Jenkins with Chef	123
Managing Jenkins with Puppet	124
Scaling Jenkins	125
Appendix	126
Advanced Jenkins Installation	127
Glossary	128
General Terms	129

Getting Started with Jenkins

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

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

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

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

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

Installing Jenkins

NOTE | This is still very much a work in progress

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

Overview

Pre-install

System Requirements

WARNING

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

Minimum Recommended Configuration:

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

Recommended Configuration for Small Team:

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

Experimentation, Staging, or Production?

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

Stand-alone or Servlet?

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

Installation

WARNING

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

Unix/Linux

Debian/Ubuntu

On Debian-based distributions, such as Ubuntu, you can install Jenkins through [apt](#).

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

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

This package installation will:

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

NOTE

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

OS X

To install from the website, using a package:

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

Jenkins can also be installed using [brew](#):

¥ Install the latest release version

```
brew install jenkins
```

¥ Install the LTS version

```
brew install jenkins-lts
```

Windows

To install from the website, using the installer:

¥ [Download the latest package](#)

¥ Open the package and follow the instructions

Docker

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

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

```
docker pull jenkins
```

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

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

Other

See [Advanced Jenkins Installation](#)

Post-install (Setup Wizard)

Create Admin User and Password for Jenkins

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

```
*****
```

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

```
41d2b60b0e4cb5bf2025d33b21cb
```

```
*****
```

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

Initial Plugin Installation

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

Using Jenkins

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

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

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

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

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

WARNING

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

Fingerprints

Remote API

NOTE | This is still very much a work in progress

Security

CSRF

Managing Jenkins

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

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

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

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

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

Configuring the System

NOTE | This is still very much a work in progress

Managing Security

Jenkins is used everywhere from workstations on corporate intranets, to high-powered servers connected to the public internet. To safely support this wide spread of security and threat profiles, Jenkins offers many configuration options for enabling, editing, or disabling various security features.

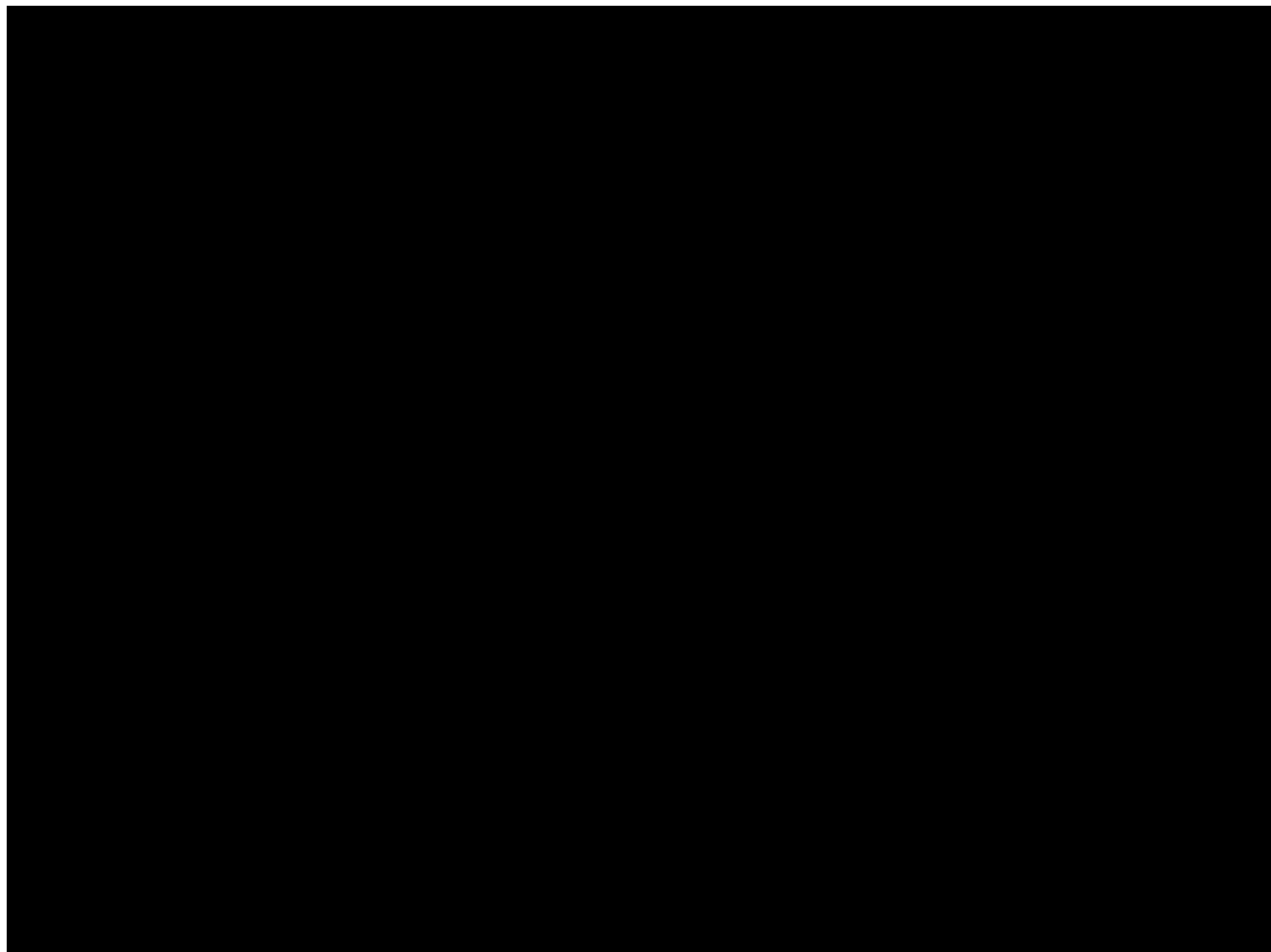
As of Jenkins 2.0, many of the security options were enabled by default to ensure that Jenkins environments remained secure unless an administrator explicitly disabled certain protections.

This section will introduce the various security options available to a Jenkins administrator, explaining the protections offered, and trade-offs to disabling some of them.

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 [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](#). If using this option, please consult the servlet container's authentication documentation.

Jenkins's 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 [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](#) 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:

¥ [Active Directory](#)

¥ [GitHub Authentication](#)

¥ [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 [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.

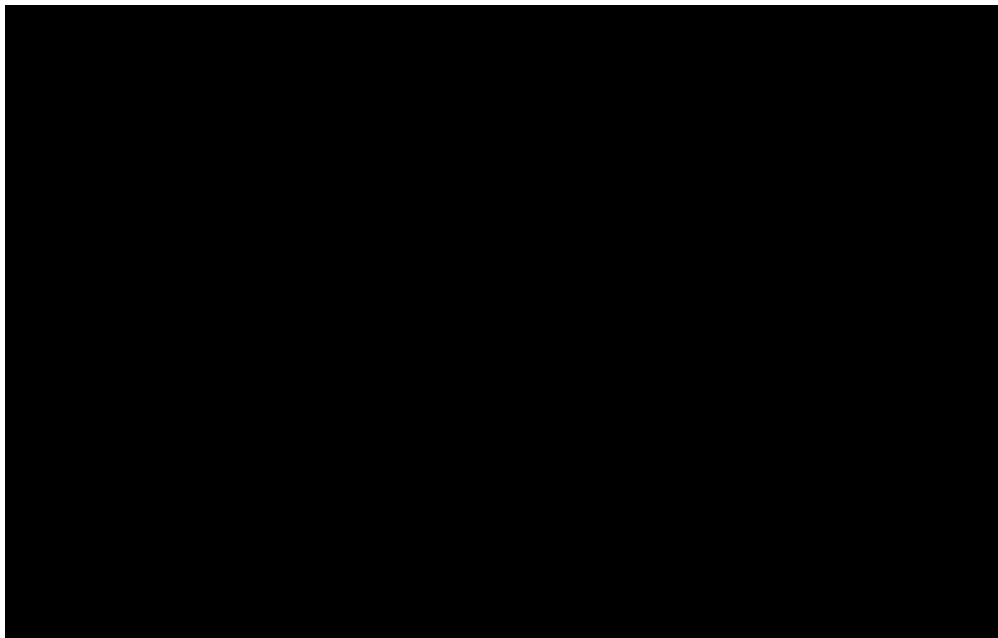


Figure 1. Matrix-based security

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

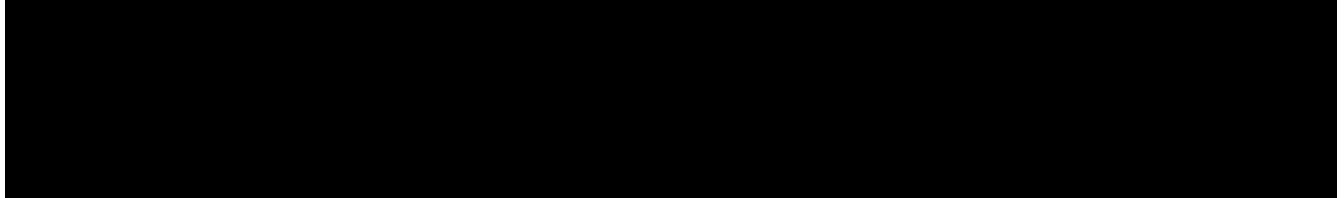
Jenkins allows user-input in a number of different configuration fields and text areas which can lead to users inadvertently, or maliciously, inserting unsafe HTML and/or JavaScript.

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-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.

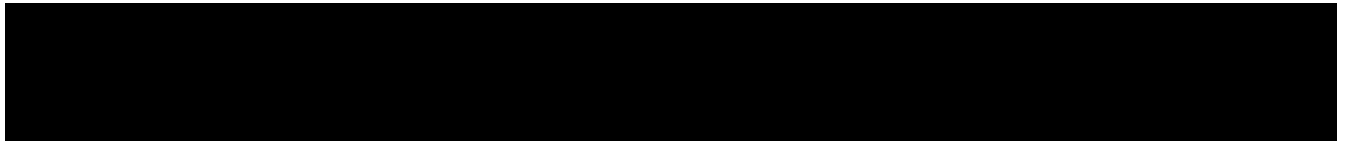
More information about CSRF exploits can be found [on the OWASP website](#).

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.

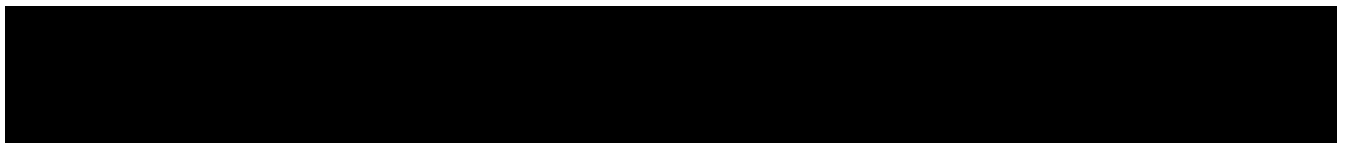
The Agent/Master Access Control system was introduced [2: Starting with 1.587, and 1.580.1, releases] to allow Jenkins administrators to add more granular access control definitions between the Jenkins master and the connected agents.



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] 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 it is not writeable by the operating system user Jenkins runs 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.

CAUTION

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 Tools

NOTE | This is still very much a work in progress

Built-in tool providers

Ant

Ant build step

Git

JDK

Maven

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](#) 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](#) 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 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.

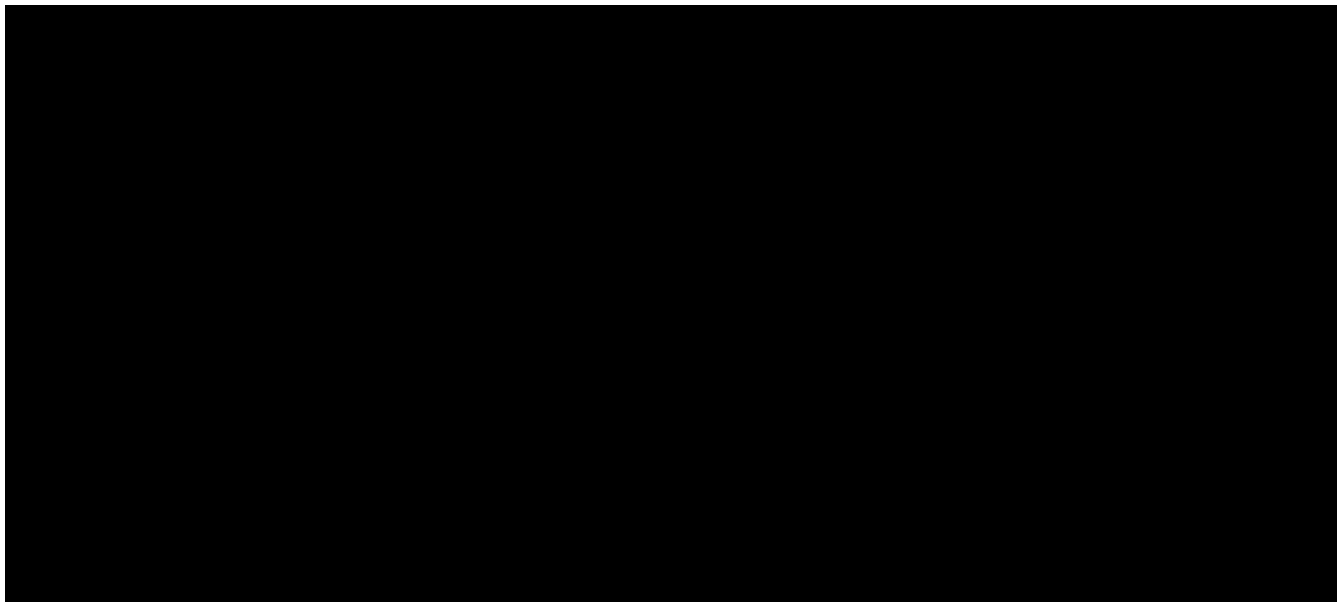
The two approaches require that the Jenkins master be able to download meta-data from an Update Center, whether the primary Update Center operated by the Jenkins project [4: updates.jenkins.io], or a custom Update Center.

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 Configure System > 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.

ÊSOURCE : If this points to a local file, that file will be installed. If
Ê this is an URL, Jenkins downloads the URL and installs that as a
Ê plugin. Otherwise the name is assumed to be the short name of the
Ê plugin in the existing update center (like "findbugs"), and the
Ê plugin will be installed from the update center.
Ê-deploy : Deploy plugins right away without postponing them until the reboot.
Ê-name VAL : If specified, the plugin will be installed as this short name
Ê (whereas normally the name is inferred from the source name
Ê automatically).
Ê-restart : Restart Jenkins upon successful installation.

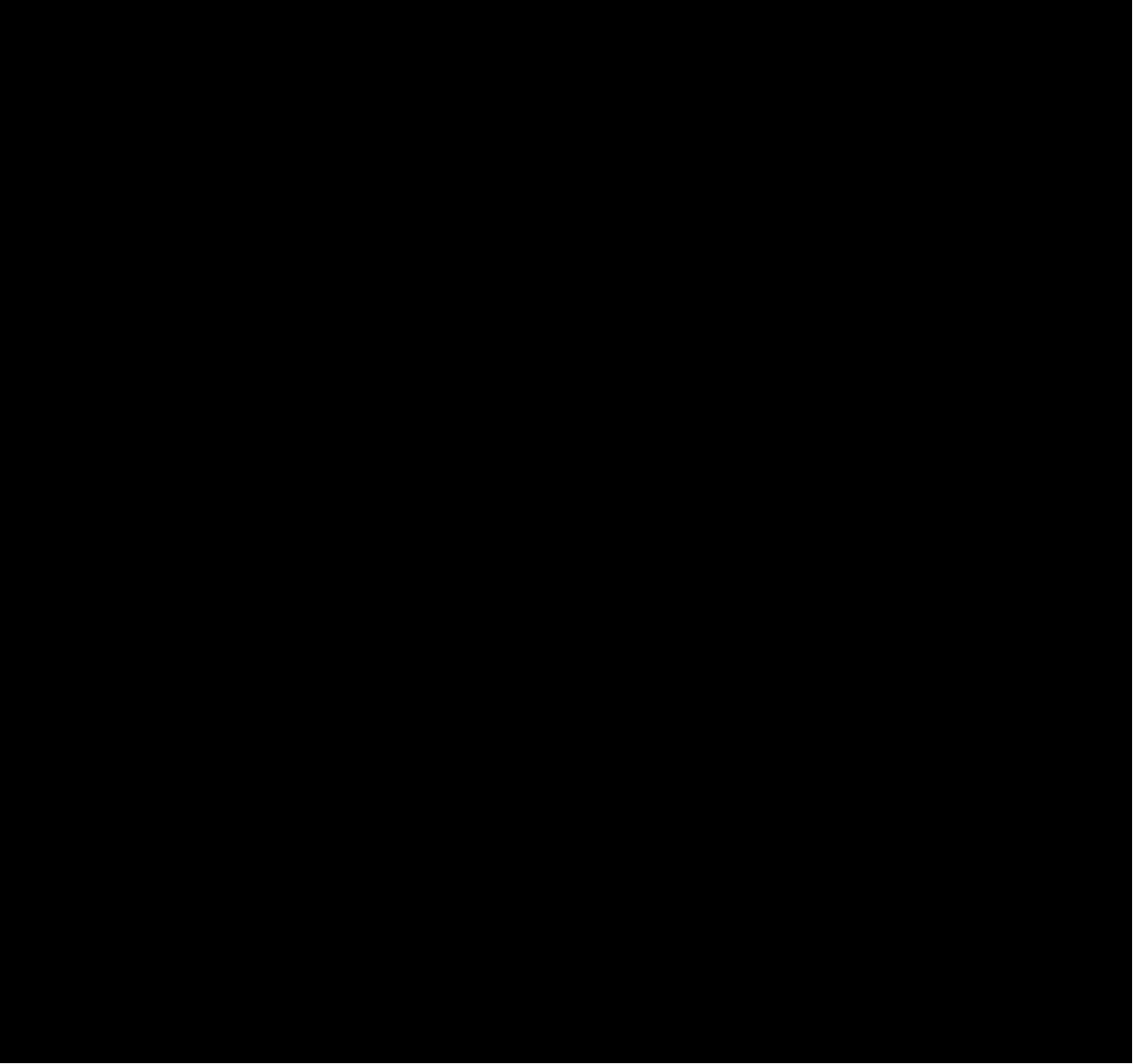
Advanced installation

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] are not always the same as the plugin's display name. Searching 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.



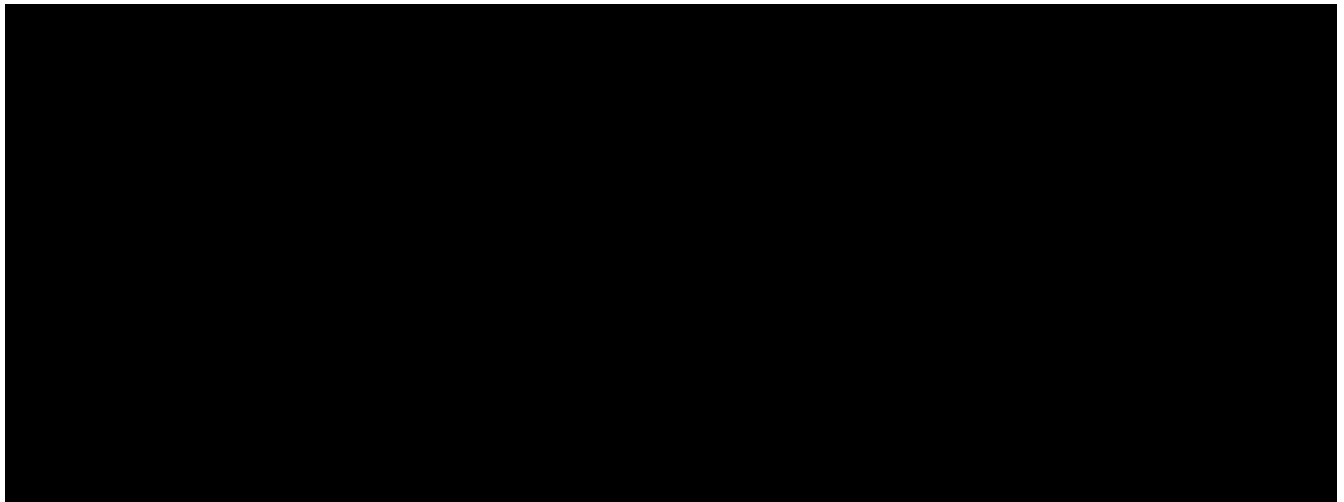
By default, the Jenkins master will check for updates from the Update Center once every 24 hours. To manually trigger a check for updates, simply click on the Check now button in the Updates tab.

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.

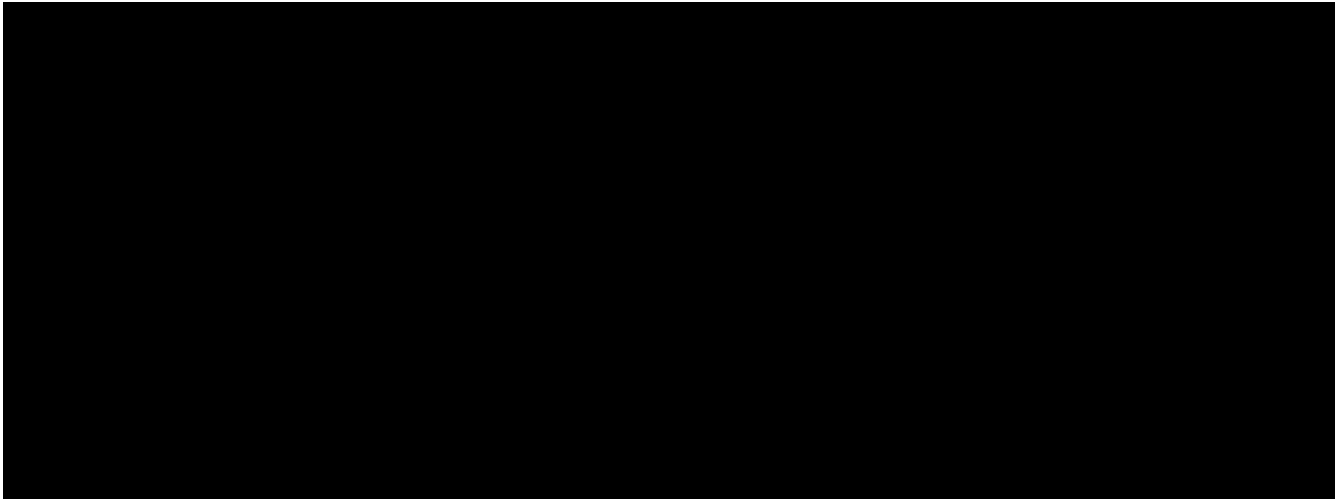
Removing old data

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 [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. The SSH approach is preferred over the CLI client as it is considered more secure.

Using the CLI

By default Jenkins will boot with a randomly assigned SSH port, which administrators may choose to override in the [Configure System](#) 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](#) 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.

Whether using the CLI via SSH, or with the CLI client, both rely primarily 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.
```

```
ÊJOB : Name of the job to build
```

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

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

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

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

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

```
Ê-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'
ÊJOB    : Name of the job
ÊBUILD  : Build number or permalink to point to the build. Defaults to the last
Ê        build
Ê-f     : If the build is in progress, stay around and append console output as
Ê        it comes, like 'tail -f'
Ê-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:
Ê authenticated
%
```

Using the CLI client

While the SSH-based CLI is preferred, there may be situations where the CLI client 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] command [options...] [arguments...]
```

The `JENKINS_URL` can be specified via the environment variable `$JENKINS_URL`.

TIP

The `JENKINS_URL` environment variable is automatically set when Jenkins forks a process during builds or Pipelines, allowing the use of the Jenkins CLI from inside a project without explicit configuration of the Jenkins URL.

Common Problems

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

Operation timed out

Check that the HTTP or JNLP 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)
    at Suppressed: java.io.IOException: Server returned HTTP response code: 403 for URL: http://ci test.gce.px/cli
    at
    sun.net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.java:1840)
    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)
    at ... 3 more
```

Script Console

NOTE | This is still very much a work in progress

Managing Nodes

NOTE | This is still very much a work in progress

Managing Users

NOTE | This is still very much a work in progress

Best Practices

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

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

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

Pipeline

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

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

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

What is Pipeline?

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

    stages{
        stage('Build') { "
            steps { #
                sh 'make' $
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml' %
            }
        }
        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'
    }
}
```

- ! **agent** indicates that Jenkins should allocate an executor and workspace for this part of the Pipeline.
- " **stage** describes a stage of this Pipeline.
- # **steps** describes the steps to be run in this **stage**
- \$ **sh** executes the given shell command
- % **junit** is a Pipeline **step** provided by the **JUnit plugin** for aggregating test reports.

Why Pipeline?

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

- ¥ Code: Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
- ¥ Durable: Pipelines can survive both planned and unplanned restarts of the Jenkins master.
- ¥ Pausable: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
- ¥ Versatile: Pipelines support complex real-world continuous delivery requirements, including the ability to fork/join, loop, and perform work in parallel.
- ¥ Extensible: The Pipeline plugin supports custom extensions to its DSL [5: [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: [GitHub Organization Folder plugin](#)]

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

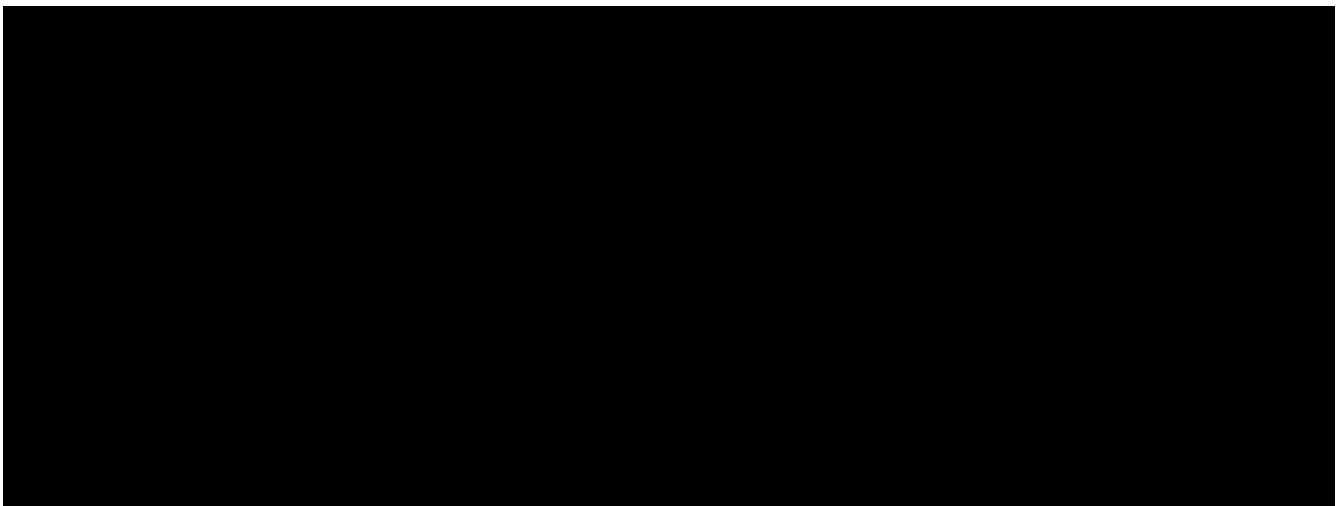


Figure 2. Pipeline Flow

Scripted Pipeline Syntax and Declarative Pipeline Syntax

When Pipeline was first conceived it was natural to begin with Groovy as the foundation. Jenkins already had an embedded Groovy scripting engine with a console to interact with all parts of Jenkins and Groovy is a great language for creating a custom Domain-Specific Language (DSL). The [Job-DSL](#), plugin for Jenkins was also written in Groovy. This plugin lets you automate the creation and editing of any job in Jenkins, including Pipelines.

Pipeline initially introduced key pipeline-specific concepts such as `node`, `stage`, `parallel`, and extension points to allow plugins to add other steps to the DSL but, otherwise didn't restrict the use of Groovy. This original syntax for creating Pipelines is now referred to as "Scripted Pipeline" and includes full programmatic control to allow scripting continuous delivery pipelines in Groovy. This gives Pipeline creators tremendous flexibility in defining a pipeline and allows it to be extended via [Shared Libraries](#) or plugins.

CAUTION

Due to the need to serialize all variables for durability some Groovy idioms are not fully supported yet. See [JENKINS-27421](#) and [JENKINS-26481](#) for more information.

Writing Pipelines with Scripted Pipeline syntax, however, does require at least some proficiency with Groovy. Requiring all team members that touch the application's Pipeline to understand Groovy limits the ability for full-participation in code-review, audits, and editing of the Pipeline as part of the application code. Enter "Declarative Pipeline" syntax.

Declarative Pipeline syntax was created to extend Pipeline to users of all experience levels and complement Scripted Pipeline syntax. As the name implies, it is intended to enable declarative programming [8: [Declarative Programming](#)] for defining Pipelines as opposed to the imperative programming [9: [Imperative Programming](#)] provided by Scripted Pipeline. While it is still a DSL written on top of Groovy, Declarative Pipeline is limited to a pre-defined structure that is much more specific to continuous delivery. This allows all stakeholders to help create, edit, review, and audit the application's Pipeline.

Scripted Pipeline and Declarative Pipeline both use the same underlying Pipeline execution engine and both are fully-supported. You can use whichever you prefer in any of your Pipelines and even combine them when needed. All examples in this handbook will show both a Declarative Pipeline version and Scripted Pipeline version for your reference.

Pipeline Terms

Step

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

Node

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

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

CAUTION

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

Stage

`stage` is a step for defining a conceptually distinct subset of the entire Pipeline, for example: "Build", "Test", and "Deploy", which is used by many plugins to visualize or present Jenkins Pipeline status/progress. [10: [Blue Ocean, Pipeline Stage View plugin](#)]

Getting Started

Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline DSL. [11: [Domain-Specific Language](#)]

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

Prerequisites

To use Jenkins Pipeline, you will need:

¥ Jenkins 2.x or later (older versions back to 1.642.3 may work but are not recommended)

¥ Pipeline plugin [12: [Pipeline plugin](#)]

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

Defining a Pipeline

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

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

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

The syntax for defining a Pipeline with either approach is the same, but while Jenkins supports entering Pipeline directly into the web UI, it's generally considered best practice to define the Pipeline in a [Jenkinsfile](#) which Jenkins will then load directly from source control. [13: en.wikipedia.org/wiki/Source_control_management]

Defining a Pipeline in the Web UI

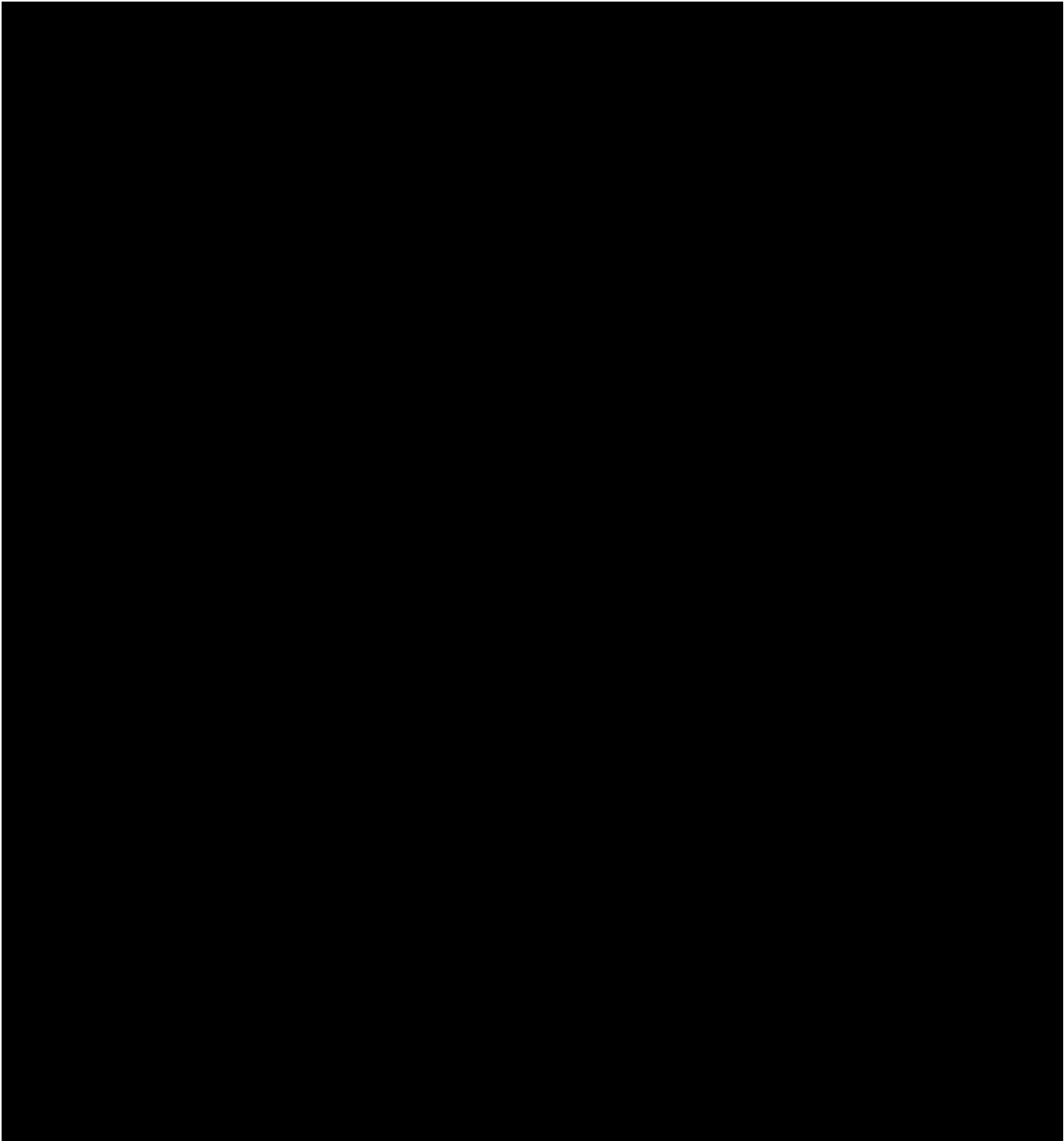
To create a basic Pipeline in the Jenkins web UI, follow these steps:

- ¥ Click New Item on Jenkins home page.

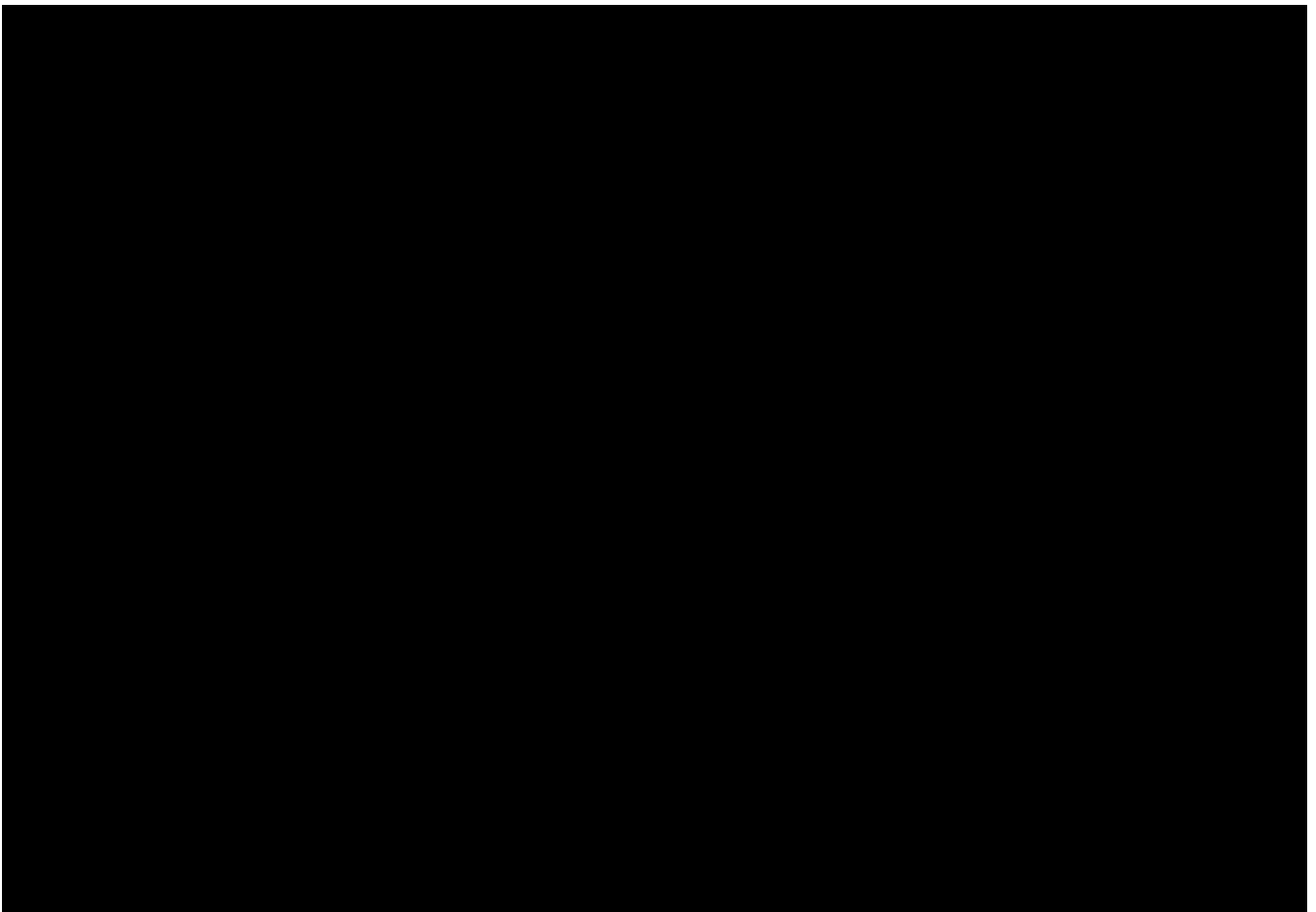
- ¥ Enter a name for your Pipeline, select Pipeline and click OK.

CAUTION

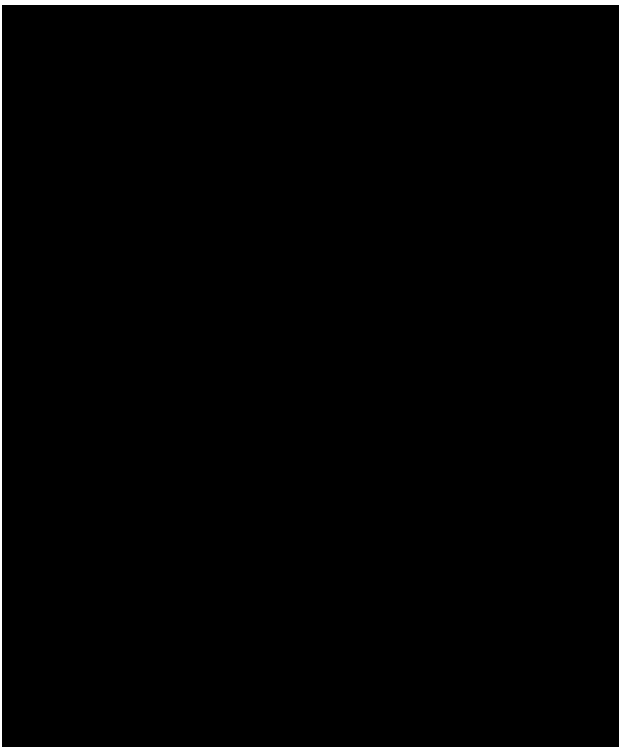
Jenkins uses the name of the Pipeline to create directories on disk. Pipeline names which include spaces may uncover bugs in scripts which do not expect paths to contain spaces.



¥ 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 {  
    echo 'Hello World'  
}  
// Declarative not yet implemented //
```

! `node` allocates an executor and workspace in the Jenkins environment.

" `echo` writes simple string in the Console Output.

Defining a Pipeline in SCM

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

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

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

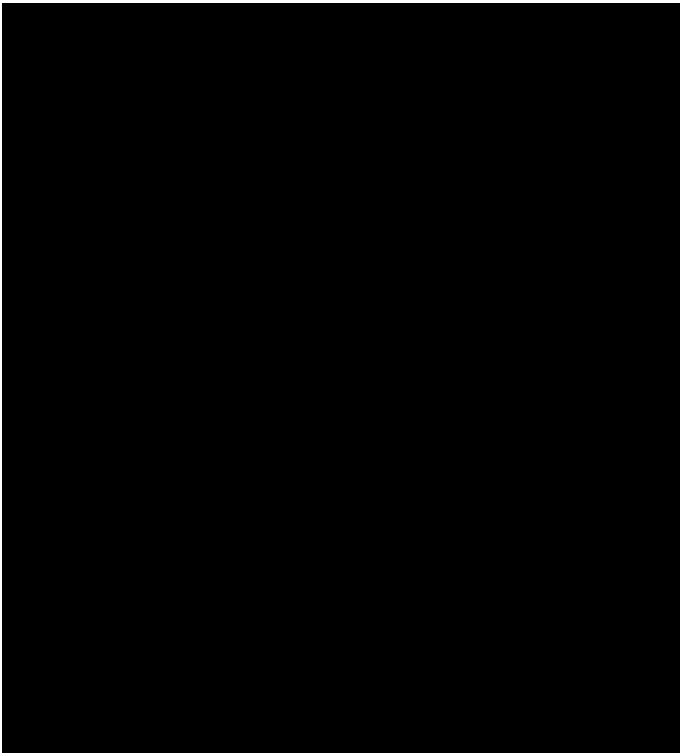
TIP

The first line of a `Jenkinsfile` should be `#!/groovy` [15: [en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))] which text editors, IDEs, GitHub, etc will use to syntax highlight the `Jenkinsfile` properly as Groovy code.

Built-in Documentation

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

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



Snippet Generator

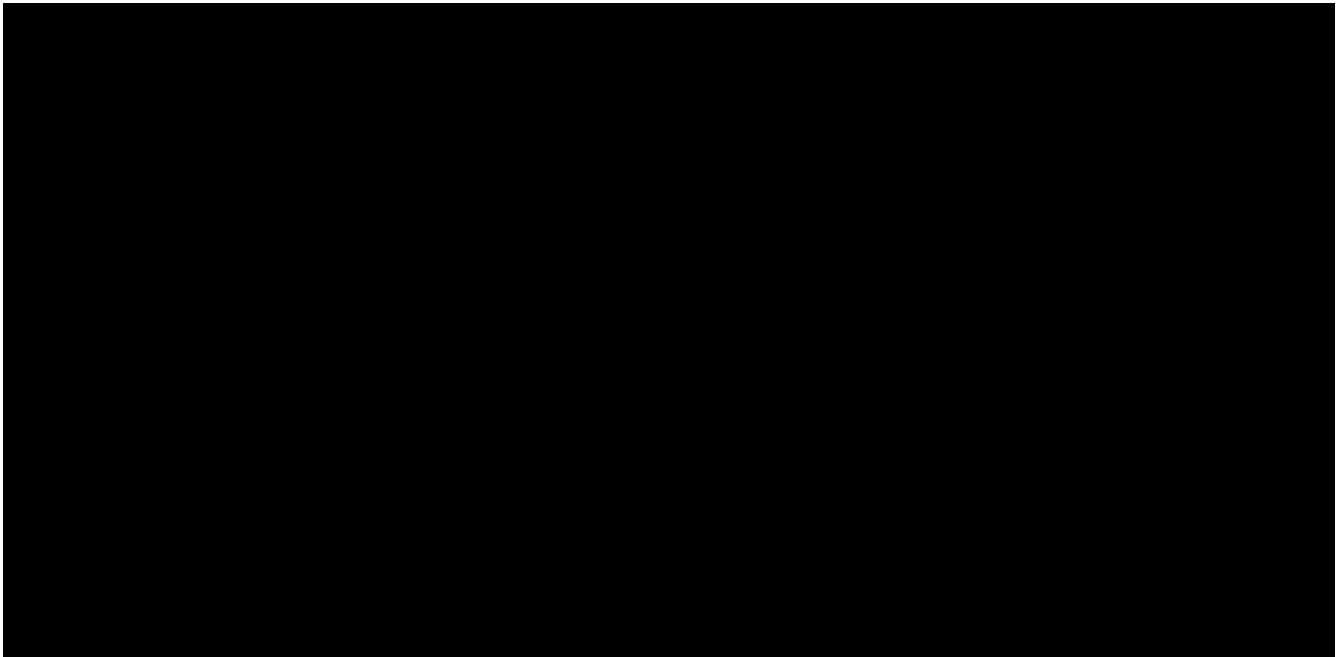
The built-in "Snippet Generator" utility is helpful for creating bits of code for individual steps, discovering new steps provided by plugins, or experimenting with different parameters for a particular step.

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

To generate a step snippet with the Snippet Generator:

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

into a Pipeline.



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

Global Variable Reference

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

The variables provided by default in Pipeline are:

env

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

params

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

currentBuild

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

Further Reading

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

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

Additional Resources

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

Multibranch Pipelines

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

Creating a Multibranch Pipeline

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

This eliminates the need for manual Pipeline creation and management.

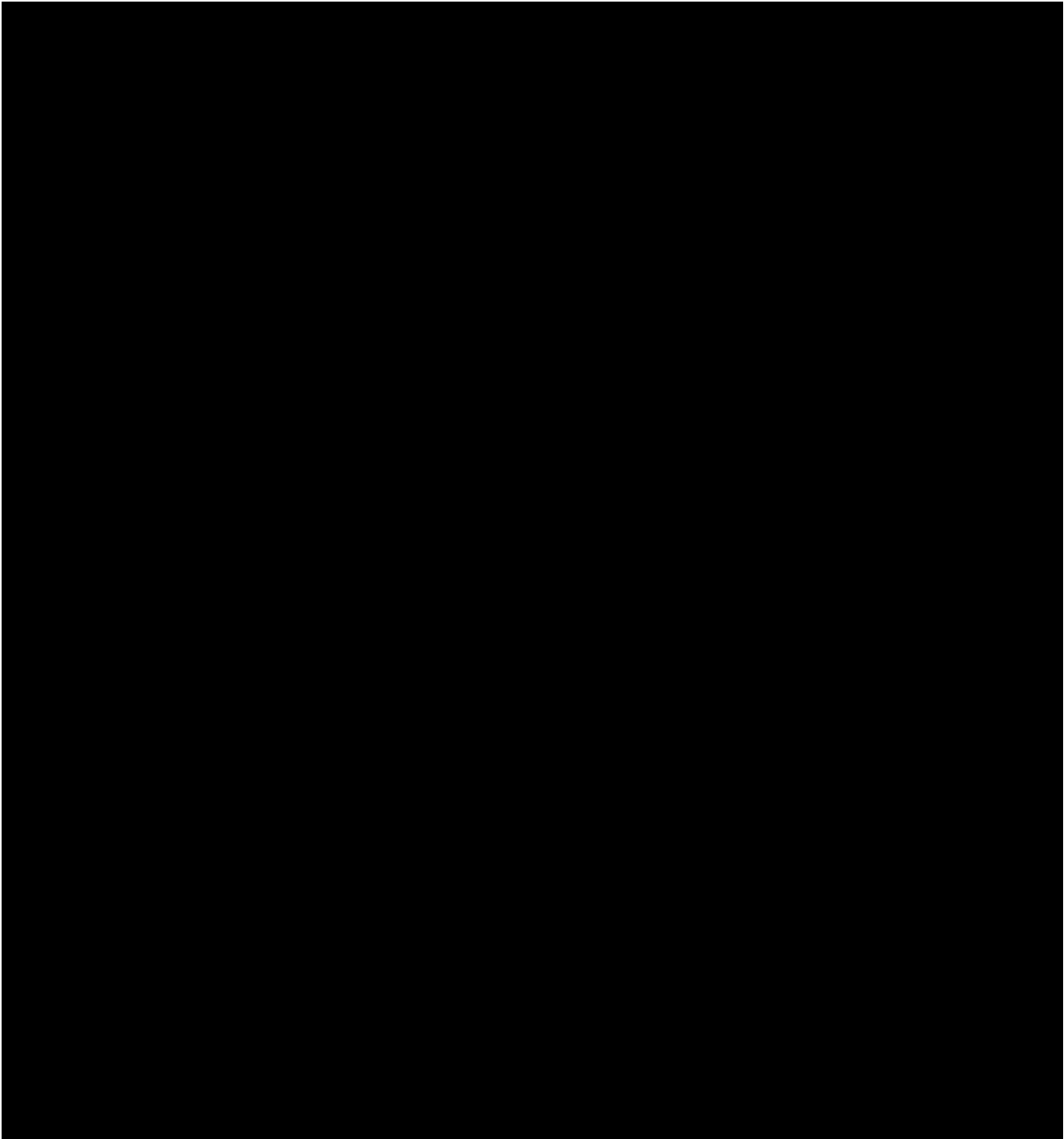
To create a Multibranch Pipeline:

¥ Click New Item on Jenkins home page.

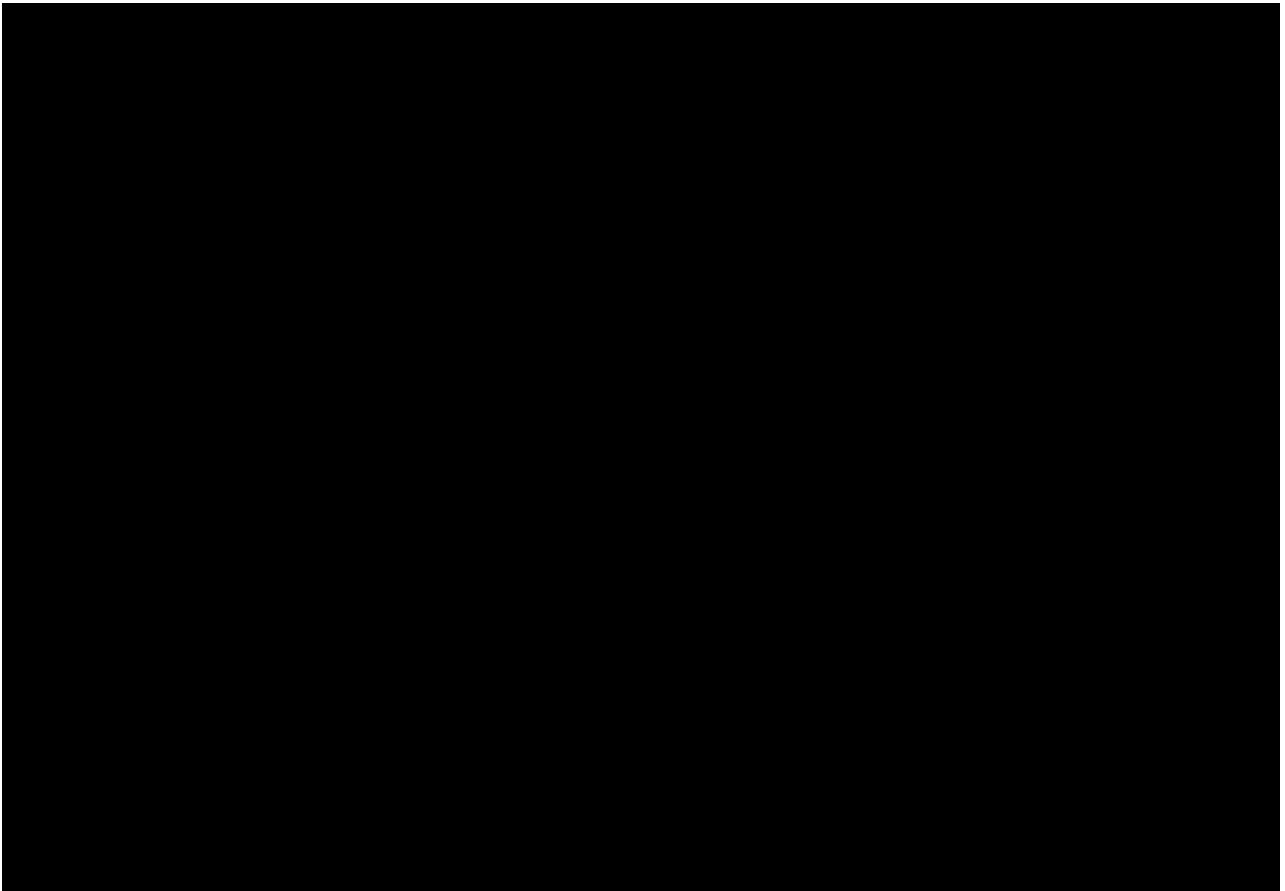
¥ Enter a name for your Pipeline, select Multibranch Pipeline and click OK.

CAUTION

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



¥ 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](#)), 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 [GitHub Branch Source](#) and [Bitbucket Branch Source](#) plugins. Please consult their documentation for further information on how to use those plugins.

Using Organization Folders

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

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

The Jenkinsfile

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

Creating a `Jenkinsfile`, which is checked into source control [16: 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] for the Pipeline, which can be viewed and edited by multiple members of the project.

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

Creating a Jenkinsfile

As discussed in the [Getting Started](#) section, a **Jenkinsfile** is a text file that contains the definition of a Jenkins Pipeline and is checked into source control. Consider the following Pipeline which implements a basic three-stage continuous delivery pipeline.

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

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

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

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

NOTE

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

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

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

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

! The **checkout** step will checkout code from source control; **scm** is a special variable which

instructs the `checkout` step to clone the specific revision which triggered this Pipeline run.

Build

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

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

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

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

CAUTION

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

Test

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

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

```
// Script //
node {
    Ê /* .. snip .. */
    Ê stage('Test') {
    Ê     /* `make check` returns non-zero on test failures,
    Ê     * using `true` to allow the Pipeline to continue nonetheless
    Ê     */
    Ê     sh 'make check || true' !
    Ê     junit '**/target/*.xml' ""
    Ê }
    Ê /* .. snip .. */
}
// Declarative not yet implemented //
```

! 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.

" `junit` captures and associates the JUnit XML files matching the inclusion pattern (`*/target/*.xml`).

Deploy

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

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

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

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

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

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

Advanced Syntax for Scripted Pipeline

Scripted Pipeline is a domain-specific language [18: en.wikipedia.org/wiki/Domain-specific_language] based on Groovy, most [Groovy syntax](#) can be used in Scripted Pipeline without modification.

String Interpolation

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

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

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

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

Would result in:

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

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

Working with the Environment

Jenkins Pipeline exposes environment variables via the global variable `env`, which is available from anywhere within a `Jenkinsfile`. The full list of environment variables accessible from within Jenkins Pipeline is documented at localhost:8080/pipeline-syntax/globals#env, assuming a Jenkins master is running on `localhost:8080`, and includes:

BUILD_ID

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

JOB_NAME

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

JENKINS_URL

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

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

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

Setting environment variables

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

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

Build Parameters

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

Assuming that a String parameter named "Greeting" has been configured for the Pipeline project in the web UI, a [Jenkinsfile](#) can access that parameter via [\\$Greeting](#):

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

Handling Failures

Scripted Pipeline relies on Groovy's built-in [try/catch/finally](#) semantics for handling failures during execution of the Pipeline.

In the [test](#) example above, the [sh](#) step was modified to never return a non-zero exit code ([sh 'make check || true'](#)). This approach, while valid, means the following stages need to check

`currentBuild.result` to know if there has been a test failure or not.

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

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

Using multiple nodes

In all previous uses of the `node` step, it has been used without any arguments. This means Jenkins will allocate an executor wherever one is available. The `node` step can take an optional "label" parameter, which is helpful for more advanced use-cases such as executing builds/tests across multiple platforms.

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

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

stage('Test') {
    node('linux') { "
    checkout scm
    try {
        unstash 'app' #
        sh 'make check'
    }
    finally {
        junit '**/target/*.xml'
    }
}
    node('windows') {
        checkout scm
        try {
            unstash 'app'
            bat 'make check' $
        }
        finally {
            junit '**/target/*.xml'
        }
    }
}
// Declarative not yet implemented //
```

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

Executing in parallel

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

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

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

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

stage('Test') {
    parallel linux: {
        node('linux') {
            checkout scm
            try {
                unstash 'app'
                sh 'make check'
            }
            finally {
                junit '**/target/*.xml'
            }
        }
    },
    windows: {
        node('windows') {
            /* .. snip .. */
        }
    }
}
// Declarative not yet implemented //
```

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

Optional step arguments

Groovy allows parentheses around function arguments to be omitted.

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

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

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

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

Unresolved directive in user-handbook.adoc - include::abbr.asc[]

Docker Pipeline Plugin

Introduction

Many organizations are using [Docker](#) to unify their build and test environments across machines and provide an efficient way to deploy applications into production. This plugin offers a convenient domain-specific language (DSL) for performing some of the most commonly needed Docker operations in a continuous-deployment pipeline from a Pipeline script.

The entry point for all of this plugin's functionality is a `docker` global variable, available without import to any flow script when the plugin is enabled. To get detailed information on available methods, open the *Pipeline Syntax* available on any Pipeline Job page:

 | [docker-workflow-screenshots/dsl-help.png](#)

Running build steps inside containers

It is commonplace for Jenkins projects to require a specific toolset or libraries to be available during a build. If many projects in a Jenkins installation have the same requirements, and there are few agents, it is not hard to just preconfigure those agents accordingly. In other cases it is feasible to keep such files in project source control. Finally, for some tools—especially those with a self-contained, platform-independent download, like Maven—it is possible to use the Jenkins tool installer system with the Pipeline `tool` step to retrieve tools on demand. However many cases remain where these techniques are not practical.

For builds which can run on Linux, Docker provides an ideal solution to this problem. Each project need merely select an image containing all the tools and libraries it would need. (This might be a publicly available image like [maven](#), or it might have been built by this or another Jenkins project.) Developers can also run build steps locally using an environment identical to that used by the Jenkins project.

There are two ways to run Jenkins build steps in such an image. One is to include a Java runtime and Jenkins slave agent (`slave.jar`) inside the image, and add a *Docker* cloud using the Docker plugin. Then the entire agent runs wholly inside the image, so any builds tied to that cloud label can assume a particular environment.

For cases where you do not want to “pollute” the image with Java and the Jenkins agent, or just want a simpler and more flexible setup, this plugin provides a way to run build steps inside an arbitrary image. In the simplest case, just select an `image` and run your whole build `inside` it:

```
docker.image('maven:3.3.3-jdk-8').inside {  
    git 'É your-sourcesÉ'  
    sh 'mvn -B clean install'  
}
```

The above is a complete Pipeline script. `inside` will:

1. Automatically grab an agent and a workspace (no extra `node` block is required).
2. Pull the requested image to the Docker server (if not already cached).
3. Start a container running that image.
4. Mount the Jenkins workspace as a `volume` inside the container, using the same file path.
5. Run your build steps. External processes like `sh` will be wrapped in `docker exec` so they are run inside the container. Other steps (such as test reporting) run unmodified: they can still access workspace files created by build steps.
6. At the end of the block, stop the container and discard any storage it consumed.
7. Record the fact that the build used the specified image. This unlocks features in other Jenkins plugins: you can track all projects using an image, or configure this project to be triggered automatically when an updated image is pushed to the Docker registry. If you use [Cloudbees Docker Traceability](#), you will be also able to see a history of the image deployments on Docker servers.

If you are running a tool like Maven which has a large download cache, running each build inside its own image will mean that a large quantity of data is downloaded from the network on every build, which is usually undesirable. The easiest way to avoid this is to redirect the cache to the agent workspace, so that if you run another build on the same agent, it will run much more quickly. In the case of Maven:

TIP

```
docker.image('maven:3.3.3-jdk-8').inside {
    git 'your-sources'
    writeFile file: 'settings.xml', text:
    "<settings><localRepository>${pwd()}/.m2repo</localRepository></settings>"
    sh 'mvn -B -s settings.xml clean install'
}
```

(If you wanted to use a cache location elsewhere on the agent, you would need to pass an extra `--volume` option to `inside` so that the container could see that path.)

Another solution is to pass an argument to `inside` to mount a sharable volume, such as `-v m2repo:/m2repo`, and use that path as the `localRepository`. Just beware that the default local repository management in Maven is not thread-safe for concurrent builds, and `install:install` could pollute the local repository across builds or even across jobs. The safest solution is to use a nearby repository mirror as a cache.

For `inside` to work, the Docker server and the Jenkins agent must use the same filesystem, so that the workspace can be mounted. The easiest way to ensure this is for the Docker server to be running on localhost (the same computer as the agent). Currently neither the Jenkins plugin nor the Docker CLI will automatically detect the case that the server is running remotely; a typical symptom would be errors from nested `sh` commands such as

NOTE

```
cannot create /É@tmp/durable-É/pid: Directory nonexistent
```

or negative exit codes.

When Jenkins can detect that the agent is itself running inside a Docker container, it will automatically pass the `--volumes-from` argument to the `inside` container, ensuring that it can share a workspace with the agent.

Customizing agent allocation

All DSL functions which run some Docker command automatically acquire an agent (executor) and a workspace if necessary. For more complex scripts which perform several commands using the DSL, you will typically want to run a block, or the whole script, on the *same* agent and workspace. In that case just wrap the block in `node`, selecting a label if desired:

```
node('linux') {
  É def maven = docker.image('maven:latest')
  É maven.pull() // make sure we have the latest available from Docker Hub
  É maven.inside {
  É   // É as above
  É }
}
```

Here we ensure that the same agent runs both `pull` and `inside`, so the local image cache update by the first step is seen by the second.

Building and publishing images

If your build needs to create a Docker image, use the `build` method, which takes an image name with an optional tag and creates it from a `Dockerfile`. This also returns a handle to the result image, so you can work with it further:

```
node {
  Ê git 'Ê' // checks out Dockerfile & Makefile
  Ê def myEnv = docker.build 'my-environment:snapshot'
  Ê myEnv.inside {
  Ê   sh 'make test'
  Ê }
}
```

Here the `build` method takes a `Dockerfile` in your source tree specifying a build environment (for example `RUN apt-get install -y libapr1-dev`). Then a `Makefile` in the same source tree describes how to build your actual project in that environment.

If you want to publish a newly created image to Docker Hub (or your own registry—discussed below), use `push`:

```
node {
  Ê git 'Ê' // checks out Dockerfile and some project sources
  Ê def newApp = docker.build "mycorp/myapp: ${env.BUILD_TAG}"
  Ê newApp.push()
}
```

Here we are giving the image a tag which identifies the Jenkins project and build number that created it. (See the documentation for the `env` global variable.) The image is pushed under this tag name to the registry.

To push an image into a staging or production environment, a common style is to update a predefined tag such as `latest` in the registry. In this case just specify the tag name:

```
node {
  Ê stage 'Building image'
  Ê git 'Ê'
  Ê def newApp = docker.build "mycorp/myapp: ${env.BUILD_TAG}"
  Ê newApp.push() // record this snapshot (optional)
  Ê stage 'Test image'
  Ê // run some tests on it (see below), then if everything looks good:
  Ê stage 'Approve image'
  Ê newApp.push 'latest'
}
```

The `build` method records information in Jenkins tied to this project build: what image was built, and what image that was derived from (the `FROM` instruction at the top of your `Dockerfile`). Other plugins can then identify the build which created an image known to have been used by a downstream build, or deployed to a particular environment. You can also have this project be triggered when an update is pushed to the ancestor image (`FROM`) in a registry.

Running and testing containers

To run an image you built, or pulled from a registry, you can use the `run` method. This returns a handle to the running container. More safely, you can use the `withRun` method, which automatically stops the container at the end of a block:

```
node {  
  É git 'É'  
  É docker.image('mysql').withRun {c ->  
    É sh './test-with-local-db'  
  É }  
}
```

The above simply starts a container running a test MySQL database and runs a regular build while that container is running. Unlike `inside`, shell steps inside the block are not run inside the container, but they could connect to it using a local TCP port for example.

You can also access the `id` of the running container, which is passed as an argument to the block, in case you need to do anything further with it:

```
// É as above, but also dump logs before we finish:  
sh "docker logs ${c.id}"
```

Like `inside`, `run` and `withRun` record the fact that the build used the specified image.

Specifying a custom registry and server

So far we have assumed that you are using the public Docker Hub as the image registry, and connecting to a Docker server in the default location (typically a daemon running locally on a Linux agent). Either or both of these settings can be easily customized.

To select a custom registry, wrap build steps which need it in the `withRegistry` method on `docker` (inside `node` if you want to specify an agent explicitly). You should pass in a registry URL. If the registry requires authentication, you can add the ID of username/password credentials. (*Credentials* link in the Jenkins index page or in a folder; when creating the credentials, use the *Advanced* section to specify a memorable ID for use in your pipelines.)

```
docker.withRegistry('https://docker.mycorp.com/', 'docker-login') {  
  É git 'É'  
  É docker.build('myapp').push('latest')  
}
```

The above builds an image from a `Dockerfile`, and then publishes it (under the `latest` tag) to a password-protected registry. There is no need to preconfigure authentication on the agent.

To select a non-default Docker server, such as for [Docker Swarm](#), use the `withServer` method. You

pass in a URI, and optionally the ID of *Docker Server Certificate Authentication* credentials (which encode a client key and client/server certificates to support TLS).

```
docker.withServer('tcp://swarm.mycorp.com:2376', 'swarm-certs') {
  Ê docker.image('httpd').withRun('-p 8080:80') {c ->
  Ê   sh "curl -i http://${hostIp(c)}:8080/"
  Ê }
  }
  def hostIp(container) {
  Ê sh "docker inspect -f {{.Node.Ip}} ${container.id} > hostIp"
  Ê readFile('hostIp').trim()
  }
```

Note that you cannot use `inside` or `build` with a Swarm server, and some versions of Swarm do not support interacting with a custom registry either.

Advanced usage

If your script needs to run other Docker client commands or options not covered by the DSL, just use a `sh` step. You can still take advantage of some DSL methods like `imageName` to prepend a registry ID:

```
docker.withRegistry('https://docker.mycorp.com/') {
  Ê def myImg = docker.image('myImg')
  Ê // or docker.build, etc.
  Ê sh "docker pull --all-tags ${myImg.imageName()}"
  Ê // runs: docker pull --all-tags docker.mycorp.com/myImg
  }
```

and you can be assured that the environment variables and files needed to connect to any custom registry and/or server will be prepared.

Demonstrations

A Docker image is available for you to run which demonstrates a complete flow script which builds an application as an image, tests it from another container, and publishes the result to a private registry. [Instructions on running this demonstration](#)

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].

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

Defining Shared Libraries

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

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

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

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

Directory structure

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

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

The `src` directory should look like standard Java source directory structure. This directory is added to the classpath when executing Pipelines.

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

The Groovy source files in these directories get the same `ØCPS transformation` as in Scripted

Pipeline.

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

Other directories under the root are reserved for future enhancements.

Global Shared Libraries

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

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

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

Folder-level Shared Libraries

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

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

Automatic Shared Libraries

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

Using libraries

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

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

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

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

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

NOTE

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

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

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

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

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

Global Variables however, are resolved at runtime.

Overriding versions

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

Writing libraries

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

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

// point in 3D space
class Point {
    Ê 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) {
    Ê 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 {
    Ê def steps
    Ê Utilities(steps) {this.steps = steps}
    Ê def mvn(args) {
    Ê     steps.sh "${steps.tool 'Maven'}/bin/mvn -o ${args}"
    Ê }
}
```

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

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 {
    Ê mvn this, 'clean package'
}
```

Defining global variables

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

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

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

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

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

Defining steps

Shared Libraries can also define global variables which behave similarly to built-in steps, such as `sh` or `git`. Global variables defined in Shared Libraries must be named with all lower-case or "camelCased" in order to be loaded properly by Pipeline. [20: gist.github.com/rtyler/e5e57f075af381fce4ed3ae57aa1f0c2]

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

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

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

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

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

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

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

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

Defining a more structured DSL

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

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

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

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

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

Using third-party libraries

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

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

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

Loading resources

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

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

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

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

Pretesting library changes

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

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

checked out again.)

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

Plugin Developer Guide

If you are maintaining (or creating) a plugin and wish its features to work smoothly with Pipeline, there are a number of special considerations.

Extension points accessible via metastep

Several common types of plugin features (`@Extension`s`) can be invoked from a Pipeline script without any special plugin dependencies so long as you use newer Jenkins core APIs. Then there is `0metastep0` in Pipeline (``step, checkout, wrap`) which loads the extension by class name and calls it.

General guidelines

There are several considerations common to the various metasteps.

Jenkins core dependency

First, make sure the baseline Jenkins version in your `pom.xml` is sufficiently new.

Suggested versions for: - `[Basic usage](#user-content-basic-update)` - `[Build wrappers](#user-content-build-wrappers-1)`

This introduces some new API methods, and deprecates some old ones.

If you are nervous about making your plugin depend on a recent Jenkins version, remember that you can always create a branch from your previous release (setting the version to `x.y.1-SNAPSHOT`) that works with older versions of Jenkins and `git cherry-pick -x` trunk changes into it as needed; or merge from one branch to another if that is easier. (`mvn -B release:prepare release:perform` works fine on a branch and knows to increment just the last version component.)

More general APIs

Replace `AbstractBuild.getProject` with `Run.getParent`.

`BuildListener` has also been replaced with `TaskListener` in new method overloads.

If you need a `Node` where the build is running to replace `getBuildOn`, you can use `FilePath.toComputer`.

`TransientProjectActionFactory` can be replaced by `TransientActionFactory<Job>`.

Variable substitutions

There is no equivalent to `AbstractBuild.getBuildVariables()` for `WorkflowRun` (any Groovy local variables are not accessible as such). Also, `WorkflowRun.getEnvironment(TaskListener)` is implemented, but only yields the initial build environment, irrespective of `withEnv` blocks and the like. (To get the *contextual* environment in a `Step`, you can inject `EnvVars` using `@StepContextParameter`; pending [JENKINS-29144](issues.jenkins-ci.org/browse/JENKINS-29144) there is no equivalent for a `SimpleBuildStep`. A `SimpleBuildWrapper` does have access to an `initialEnvironment` if required.)

Anyway code run from Pipeline should take any configuration values as literal strings and make no attempt to perform variable substitution (including via the `token-macro` plugin), since the script

author would be using Groovy facilities ("like `_${this}`") for any desired dynamic behavior. To have a single code fragment support both Pipeline and traditional builds, you can use idioms such as:

```
private final String location;
public String getLocation() {
    return location;
}
@DataBoundSetter public void setLocation(String location) {
    this.location = location;
}
private String actualLocation(Run<?,?> build, TaskListener listener) {
    if (build instanceof AbstractBuild) {
        EnvVars env = build.getEnvironment(listener);
        env.overrideAll(((AbstractBuild) build).getBuildVariables());
        return env.expand(location);
    } else {
        return location;
    }
}
```

[JENKINS-35671](issues.jenkins-ci.org/browse/JENKINS-35671) would simplify this.

Constructor vs. setters

It is a good idea to replace a lengthy `@DataBoundConstructor` with a short one taking just truly mandatory parameters (such as a server location). For all optional parameters, create a public setter marked with `@DataBoundSetter` (with any non-null default value set in the constructor or field initializer). This allows most parameters to be left at their default values in a Pipeline script, not to mention simplifying ongoing code maintenance because it is much easier to introduce new options this way.

For Java-level compatibility, leave any previous constructors in place, but mark them `@Deprecated`. Also remove `@DataBoundConstructor` from them (there can be only one).

Handling default values

To ensure *Snippet Generator* enumerates only those options the user has actually customized from their form defaults, ensure that Jelly `default` attributes match the property defaults as seen from the getter. For a cleaner XStream serial form in freestyle projects, it is best for the default value to also be represented as a null in the Java field. So for example if you want a textual property which can sensibly default to blank, your configuration form would look like

```
<f:entry field="stuff" title="${%Stuff}">
    <f:textbox/>
</f:entry>
```

and your `Describable` should use

```
private @CheckForNull String stuff;
public @CheckForNull String getStuff() {
    return stuff;
}
@DataBoundSetter public void setStuff(@CheckForNull String stuff) {
    this.stuff = Util.fixNull(stuff);
}
```

If you want a nonblank default, it is a little more complicated. If you do not care about XStream hygiene, for example because the `Describable` is a Pipeline `Step` (or is only being used as part of one):

```
<f:entry field="stuff" title="{%Stuff}">
    <f:textbox default="{%descriptor.defaultStuff}"/>
</f:entry>
```

```
private @Nonnull String stuff = DescriptorImpl.defaultStuff;
public @Nonnull String getStuff() {
    return stuff;
}
@DataBoundSetter public void setStuff(@Nonnull String stuff) {
    this.stuff = stuff;
}
@Extension public static class DescriptorImpl extends Descriptor<Whatever> {
    public static final String defaultStuff = "junk";
    //
}
```

(The `Descriptor` is the most convenient place to put a constant for use from a Jelly view: `descriptor` is always defined even if `instance` is null, and Jelly/JEXL allows a `static` field to be loaded using instance-field notation. From a Groovy view you could use any syntax supported by Java to refer to a constant, but Jelly in Jenkins is weaker: `getStatic` will not work on classes defined in plugins.)

To make sure the field is omitted from the XStream form when unmodified, you can use the same `Descriptor` and configuration form but null out the default:

```
private @CheckForNull String stuff;
public @Nonnull String getStuff() {
    return stuff == null ? DescriptorImpl.defaultStuff : stuff;
}
@DataBoundSetter public void setStuff(@Nonnull String stuff) {
    this.stuff = stuff.equals(DescriptorImpl.defaultStuff) ? null : stuff;
}
```

None of these considerations apply to mandatory parameters with no default, which should be requested in the `@DataBoundConstructor` and have a simple getter. (You could still have a `default` in

the configuration form as a hint to new users, as a complement to a full description in `help-stuff.html`, but the value chosen will always be saved.)

Handling secrets

If your plugin ever stored secrets (such as passwords) in a plain `String`-valued fields, it was already insecure and should at least have been using `Secret`. `Secret`-valued fields are more secure, but are not really appropriate for projects defined in source code, like Pipeline jobs.

Instead you should integrate with the [Credentials plugin](wiki.jenkins-ci.org/display/JENKINS/Credentials+Plugin). Then your builder etc. would typically have a `credentialId` field which just refers to the ID of the credentials. (The user can pick a mnemonic ID for use in scripted jobs.) Typically the `config.jelly` used in *Snippet Generator* will have a `<c:select/>` control, backed by a `doFillCredentialsId` web method on the `Descriptor` to enumerate credentials currently available of the intended type (such as `StandardUsernamePasswordCredentials`) and perhaps restricted to some domain (such as a hostname obtained via a `@QueryParameter` from a nearby form field). At runtime, you will look up the credentials by ID and use them.

Plugins formerly using `Secret` will generally need to use an `@Initializer` to migrate the configuration of freestyle projects to use Credentials instead.

The details of adopting Credentials are too numerous to list here. Pending a proper developer's guide, it is best to follow the example of well-maintained plugins which have already made such a conversion.

Defining symbols

By default, scripts making use of your plugin will need to refer to the (simple) Java class name of the extension. For example, if you defined

```
public class ForgetBuilder extends Builder implements SimpleBuildStep {
    private final String what;
    @DataBoundConstructor public ForgetBuilder(String what) {this.what = what;}
    public String getWhat() {return what;}
    @Override public void perform(Run build, FilePath workspace, Launcher launcher,
        TaskListener listener) throws InterruptedException, IOException {
        listener.getLogger().println("What was " + what + "?");
    }
    @Extension public static class DescriptorImpl extends BuildStepDescriptor<Builder>
    {
        @Override public String getDisplayName() {return "Forget things";}
        @Override public boolean isApplicable(Class<? extends AbstractProject> t)
        {return true;}
    }
}
```

then scripts would use this builder as follows:

```
step([$class: 'ForgetBuilder', what: 'everything'])
```

To make for a more attractive and mnemonic usage style, you can depend on `org.jenkins-ci:symbol-annotation` and add a `@Symbol` to your `Descriptor`, uniquely identifying it among extensions of its kind (in this example, `SimpleBuildStep``s):

```
// É
@Symbol("forget")
@Extension public static class DescriptorImpl extends BuildStepDescriptor<Builder> {
// É
```

Now when users of sufficiently new versions of Pipeline wish to run your builder, they can use a shorter syntax:

```
forget 'everything'
```

`@Symbol`s` are not limited to extensions used at 0top level0 by metasteps such as ``step`. Any `Descriptor` can have an associated symbol. Therefore if your plugin uses other ``Describable`s` for any kind of structured configuration, you should also annotate those implementations. For example if you have defined an extension point

```
public abstract Timeframe extends AbstractDescribableImpl<Timeframe> implements
ExtensionPoint {
É   public abstract boolean areWeThereYet();
}
```

with some implementations such as

```
@Extension public class Immediately extends Timeframe {
É   @DataBoundConstructor public Immediately() {}
É   @Override public boolean areWeThereYet() {return true;}
É   @Symbol("now")
É   @Extension public static DescriptorImpl extends Descriptor<Timeframe> {
É       @Override public String getDisplayName() {return "Right now";}
É   }
}
```

or

```
@Extension public class HoursAway extends Timeframe {
    private final long hours;
    @DataBoundConstructor public HoursAway(long hours) {this.hours = hours;}
    public long getHours() {return hours;}
    @Override public boolean areWeThereYet() {/* É */}
    @Symbol("soon")
    @Extension public static DescriptorImpl extends Descriptor<Timeframe> {
        @Override public String getDisplayName() {return "Pretty soon";}
    }
}
```

which are selectable in your configuration

```
private Timeframe when = new Immediately();
public Timeframe getWhen() {return when;}
@DataBoundSetter public void setWhen(Timeframe when) {this.when = when;}
```

then a script could select a timeframe using the symbols you have defined:

```
forget 'nothing' // whenever
forget what: 'something', when: now()
forget what: 'everything else', when: soon(1)
```

Snippet Generator will offer the simplified syntax wherever available. Freestyle project configuration will ignore the symbol, though a future version of the Job DSL plugin may take advantage of it.

SCMs

See the [user documentation](github.com/jenkinsci/workflow-scm-step-plugin/blob/master/README.md) for background. The `checkout` metastep uses an `SCM`.

As the author of an SCM plugin, there are some changes you should make to ensure your plugin can be used from pipelines. You can use `mercurial-plugin` as a relatively straightforward code example.

Basic update

Make sure your Jenkins baseline is at least 1.568 (or 1.580.1, the next LTS). Check your plugin for compilation warnings relating to `hudson.scm.*` classes to see outstanding changes you need to make. Most importantly, various methods in `SCM` which formerly took an `AbstractBuild` now take a more generic `Run` (i.e., potentially a Pipeline build) plus a `FilePath` (i.e., a workspace). Use the specified workspace rather than the former `build.getWorkspace()`, which only worked for traditional projects with a single workspace. Similarly, some methods formerly taking `AbstractProject` now take the more generic `Job`. Be sure to use `@Override` wherever possible to make sure you are using the right overloads.

Note that `changelogFile` may now be null in `checkout`. If so, just skip changelog generation. `checkout` also now takes an `SCMRevisionState` so you can know what to compare against without referring back to the build.

`SCMDescriptor.isApplicable` should be switched to the `Job` overload. Typically you will unconditionally return `true`.

Checkout key

You should override the new `getKey`. This allows a Pipeline job to match up checkouts from build to build so it knows how to look for changes.

Browser selection

You may override the new `guessBrowser`, so that scripts do not need to specify the changelog browser to display.

Commit triggers

If you have a commit trigger, generally an `UnprotectedRootAction` which schedules builds, it will need a few changes. Use `SCMTriggerItem` rather than the deprecated `SCMItem`; use `SCMTriggerItem.SCMTriggerItems.asSCMTriggerItem` rather than checking `instanceof`. Its `getSCMs` method can be used to enumerate configured SCMs, which in the case of a pipeline will be those run in the last build. Use its `getSCMTrigger` method to look for a configured trigger (for example to check `isIgnorePostCommitHooks`).

Ideally you will already be integrated with the `scm-api` plugin and implementing `SCMSource`; if not, now is a good time to try it. In the future pipelines may take advantage of this API to support automatic creation of subprojects for each detected branch.

Explicit integration

If you want to provide a smoother experience for Pipeline users than is possible via the generic `scm` step, you can add a (perhaps optional) dependency on `workflow-scm-step` to your plugin. Define a `SCMStep` using `SCMStepDescriptor` and you can define a friendly, script-oriented syntax. You still need to make the aforementioned changes, since at the end you are just preconfiguring an `SCM`.

Build steps

See the [user documentation](github.com/jenkinsci/workflow-basic-steps-plugin/blob/master/CORE-STEPS.md) for background. The metastep is `step`.

To add support for use of a `Builder` or `Publisher` from a pipeline, depend on Jenkins 1.577+, typically 1.580.1 ([tips](#basic-update)). Then implement `SimpleBuildStep`, following the guidelines in [its Javadoc](javadoc.jenkins-ci.org/jenkins/tasks/SimpleBuildStep.html). Also prefer `@DataBoundSetter`'s to a sprawling `@DataBoundConstructor` ([tips](#constructor-vs-setters)).

Mandatory workspace context

Note that a `SimpleBuildStep` is designed to work also in a freestyle project, and thus assumes that a `FilePath workspace` is available (as well as some associated services, like a `Launcher`). That is always true in a freestyle build, but is a potential limitation for use from a Pipeline build. For example, you might legitimately want to take some action outside the context of any workspace:

```
node('win64') {
  É bat 'make all'
  É archive 'myapp.exe'
}
input 'Ready to tell the world?' // could pause indefinitely, do not tie up a slave
step([$class: 'FunkyNotificationBuilder', artifact: 'myapp.exe']) // ! FAILS!
```

Even if `FunkyNotificationBuilder` implements `SimpleBuildStep`, the above will fail, because the `workspace` required by `SimpleBuildStep.perform` is missing. You could grab an arbitrary workspace just to run the builder:

```
node('win64') {
  É bat 'make all'
  É archive 'myapp.exe'
}
input 'Ready to tell the world?'
node {
  É step([$class: 'FunkyNotificationBuilder', artifact: 'myapp.exe']) // OK
}
```

but if the `workspace` is being ignored anyway (in this case because `FunkyNotificationBuilder` only cares about artifacts that have already been archived), it may be better to just write a custom step (described below).

Run listeners vs. publishers

For code which genuinely has to run after the build completes, there is `RunListener`. If the behavior of this hook needs to be customizable at the job level, the usual technique would be to define a `JobProperty`. (One distinction from freestyle projects is that in the case of Pipeline there is no way to introspect the `list of build steps` or `list of publishers` or `list of build wrappers` so any decisions based on such metadata are impossible.)

In most other cases, you just want some code to run after some *portion* of the build completes, which is typically handled with a `Publisher` if you wish to share a code base with freestyle projects. For regular `Publisher`'s, which are run as part of the build, a Pipeline script would use the `step` metastep. There are two subtypes: * `Recorder`'s generally should be placed inline with other build steps in whatever order makes sense. * `Notifier`'s can be placed in a `finally` block, or you can use the `catchError` step. [This document](github.com/jenkinsci/workflow-basic-steps-plugin/blob/master/CORE-STEPS.md#interacting-with-build-status) goes into depth.

Build wrappers

Here the metastep is `wrap`. To add support for a `BuildWrapper`, depend on Jenkins 1.599+ (typically 1.609.1), and implement `SimpleBuildWrapper`, following the guidelines in [its Javadoc](javadoc.jenkins-ci.org/jenkins/tasks/SimpleBuildWrapper.html).

Like `SimpleBuildStep`, wrappers written this way always require a workspace. If that would be constricting, consider writing a custom step instead.

Triggers

Replace `Trigger<AbstractProject>` with `Trigger<X>` where `X` is `Job` or perhaps `ParameterizedJob` or `SCMTriggerItem` and implement `TriggerDescriptor.isApplicable` accordingly.

Use `EnvironmentContributor` rather than `RunListener.setUpEnvironment`.

Clouds

Do not necessarily need any special integration, but are encouraged to use `OnceRetentionStrategy` from `durable-task` to allow Pipeline builds to survive restarts.

Custom steps

Plugins can also implement custom Pipeline steps with specialized behavior. See [here](github.com/jenkinsci/workflow-step-api-plugin/blob/master/README.md) for more.

Historical background

Traditional Jenkins `Job`'s are defined in a fairly deep type hierarchy: `FreestyleProject` " `Project` " `AbstractProject` " `Job` " `AbstractItem` " `Item`. (As well as paired `Run` types: `FreestyleBuild`, etc.) In older versions of Jenkins, much of the interesting implementation was in `AbstractProject` (or `AbstractBuild`), which was packed full of assorted features not present in `Job` (or `Run`). Some of these features were also needed by Pipeline, like having a programmatic way to start a build (optionally with parameters), or lazy-load build records, or integrate with SCM triggers. Others were not applicable to Pipeline, like declaring a single SCM and a single workspace per build, or being tied to a specific label, or running a linear sequence of build steps within the scope of a single Java method call, or having a simple list of build steps and wrappers whose configuration is guaranteed to remain the same from build to build.

`WorkflowJob` directly extends `Job` since it cannot act like an `AbstractProject`. Therefore some refactoring was needed, to make the relevant features available to other `Job` types without code or API duplication. Rather than introduce yet another level into the type hierarchy (and freezing for all time the decision about which features are more "generic" than others), mixins were introduced. Each encapsulates a set of related functionality originally tied to `AbstractProject` but now also usable from `WorkflowJob` (and potentially other future `Job` types).

- ¥ `ParameterizedJobMixin` allows a job to be scheduled to the queue (the older `BuildableItem` was inadequate), taking care also of build parameters and the REST build trigger.
- ¥ `SCMTriggerItem` integrates with `SCMTrigger`, including a definition of which SCM or SCMs a job is using, and how it should perform polling. It also allows various plugins to interoperate with the Multiple SCMs plugin without needing an explicit dependency. Supersedes and deprecates `SCMItem`.
- ¥ `LazyBuildMixin` handles the plumbing of lazy-loading build records (a system introduced in Jenkins 1.485).

For Pipeline compatibility, plugins formerly referring to `AbstractProject/AbstractBuild` will generally need to start dealing with `Job/Run` but may also need to refer to `ParameterizedJobMixin` and/or `SCMTriggerItem`. (`LazyBuildMixin` is rarely needed from outside code, as the methods defined in `Job/Run` suffice for typical purposes.)

Future improvements to Pipeline may well require yet more implementation code to be extracted from `AbstractProject/AbstractBuild`. The main constraint is the need to retain binary compatibility.

Jenkins Use-Cases

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

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

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

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

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

WARNING

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

Jenkins with .NET

NOTE | This is still very much a work in progress

Jenkins with Java

NOTE | This is still very much a work in progress

Jenkins with Python

NOTE | This is still very much a work in progress

Test Reports

Jenkins with Ruby

NOTE | This is still very much a work in progress

Test Reports

Coverage Reports

Rails

Operating Jenkins

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

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

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

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

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

Backing-up/Restoring Jenkins

NOTE | This is still very much a work in progress

Monitoring Jenkins

NOTE | This is still very much a work in progress

Securing Jenkins

NOTE | This is still very much a work in progress

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

Securing Jenkins has two aspects to it.

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

Access Control

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

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

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

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

Protect users of Jenkins from other threats

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

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

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

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

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

Disabling Security

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

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

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

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

Managing Jenkins with Chef

NOTE | This is still very much a work in progress

Managing Jenkins with Puppet

NOTE | This is still very much a work in progress

Scaling Jenkins

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

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

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

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

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

Appendix

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

WARNING

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

Advanced Jenkins Installation

NOTE | This is still very much a work in progress

Glossary

General Terms

Agent

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

Artifact

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

Build

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

Cloud

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

Core

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

Downstream

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

Executor

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

Fingerprint

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

Folder

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

Item

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

Job

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

Master

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

Node

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

Project

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

Pipeline

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

Plugin

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

Publisher

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

Step

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

Trigger

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

Update Center

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

Upstream

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

Workspace

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