Jenkins User Handbook

jenkinsci-docs@googlegroups.com

Table of Contents

Getting Started with Jenkins

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

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

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

If you are a Jenkins administrator and want to know more about managing Jenkins nodes and instances, see Managing Jenkins.

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

Installing Jenkins

NOTE	This is still very much a work in progress

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

Overview

Pre-install

System Requirements

WARNING	These are **starting points**. For a full discussion of factors see Discussion of hardware recommendations.

Minimum Recommended Configuration:

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

Recommended Configuration for Small Team:

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

Experimentation, Staging, or Production?

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

Stand-alone or Servlet?

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

Installation

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

Unix/Linux

Debian/Ubuntu

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

Recent versions are available in an apt repository. Older but stable LTS versions are in this apt repository.

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

This package installation will:

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

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

OS X

To install from the website, using a package:

- Download the latest package
- Open the package and follow the instructions

Jenkins can also be installed using `brew`:

- Install the latest release version

```
brew install jenkins
```

- Install the LTS version

```
brew install jenkins-lts
```

Windows

To install from the website, using the installer:

- Download the latest package
- Open the package and follow the instructions

Docker

You must have Docker properly installed on your machine. See the Docker installation guide for details.

First, pull the official jenkins image from Docker repository.

```
docker pull jenkins
```

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

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

Other

See Advanced Jenkins Installation

Post-install (Setup Wizard)

Create Admin User and Password for Jenkins

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

```
************************************************************

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

41d2b60b0e4cb5bf2025d33b21cb

************************************************************
```

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

Initial Plugin Installation

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

Using Jenkins

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

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

If you are not yet familiar with basic Jenkins terminology and features, start with Getting Started with Jenkins.

If you are a Jenkins administrator and want to know more about managing Jenkins nodes and instances, see Managing Jenkins.

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

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

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.

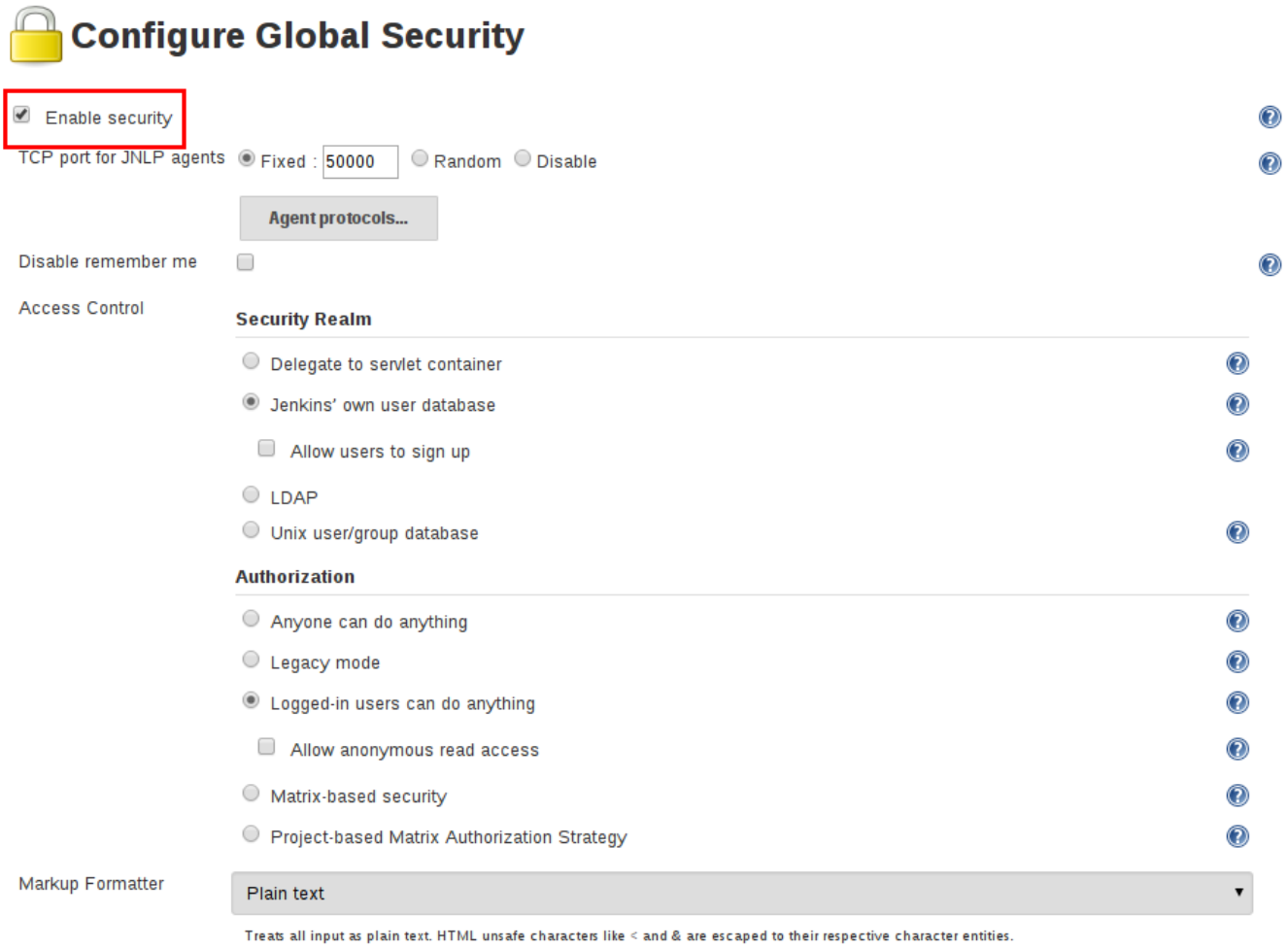
As of Jenkins 2.0, many of the security options were enabled by default to ensure that Jenkins environments remained secure unless an administrator explicitly disabled certain protections.

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

Security Realm

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

Delegate to servlet container

For delegating authentication a servlet container running the Jenkins master, such as Jetty. If using this option, please consult the servlet container's authentication documentation.

Jenkins' own user database

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

LDAP

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

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

Unix user/group database

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

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

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

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

Authorization

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

Anyone can do anything

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

Legacy mode

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

Logged in users can do anything

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

Matrix-based security

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

Project-based Matrix Authorization Strategy

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

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

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

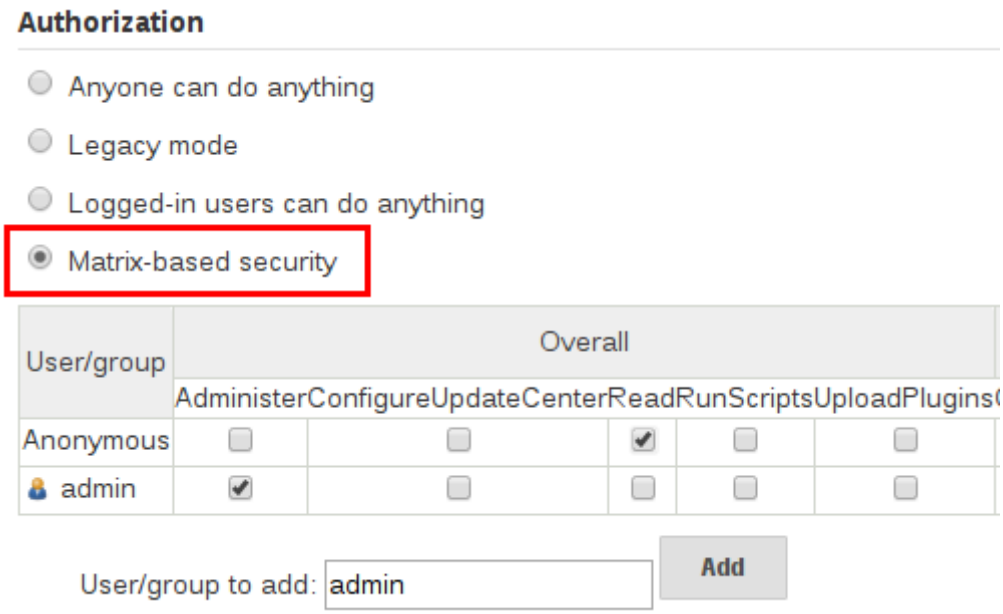


Figure 1. Matrix-based security

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

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

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

Markup Formatter

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

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

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

Cross Site Request Forgery

A cross site request forgery (or CSRF/XSRF) [1: www.owasp.org/index.php/Cross-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 is not writeable by the operating system user Jenkins run as.

File Access Rules

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

1. `allow` / `deny`: if the following two parameters match the current request being considered, an `allow` entry would allow the request to be carried out and a `deny` entry would deny the request to be rejected, regardless of what later rules might say.
2. *operation*: Type of the operation requested. The following 6 values exist. The operations can also be combined by comma-separating the values. The value of `all` indicates all the listed operations are allowed or denied.
    - `read`: read file content or list directory entries
    - `write`: write file content
    - `mkdirs`: create a new directory
    - `create`: create a file in an existing directory
    - `delete`: delete a file or directory

- `stat`: read metadata of a file/directory, such as timestamp, length, file access modes.

3. *file path*: regular expression that specifies file paths that matches this rule. In addition to the base regexp syntax, it supports the following tokens:
   - `<JENKINS_HOME>` can be used as a prefix to match the master's `JENKINS_HOME` directory.
   - `<BUILDDIR>` can be used as a prefix to match the build record directory, such as `/var/lib/jenkins/job/foo/builds/2014-10-17_12-34-56`.
   - `<BUILDID>` matches the timestamp-formatted build IDs, like `2014-10-17_12-34-56`.

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

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

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

```
allow all <JENKINS_HOME>/.*
deny all <JENKINS_HOME>/secrets/.*
```

Advanced

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

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

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

Disabling

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

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

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

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's file system.

Installing a plugin

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

1. Using the "Plugin Manager" in the web UI.
2. Using the Jenkins CLI `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.

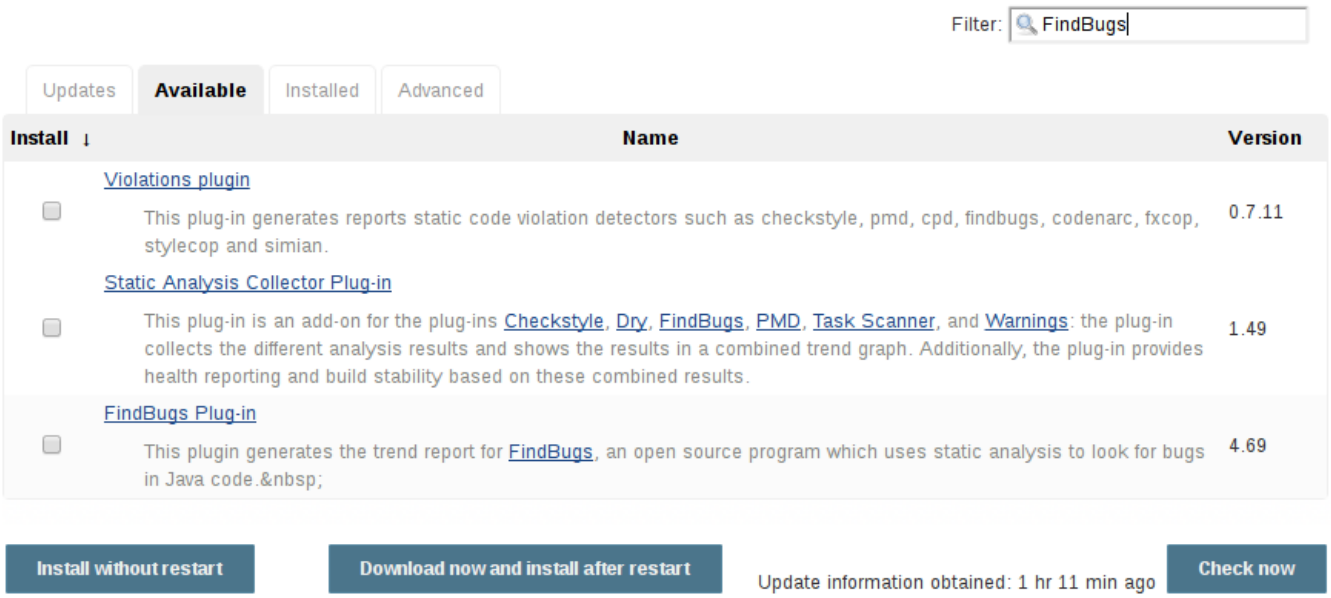
The two approaches require that the Jenkins master be able to download meta-data from an Update Center, whether the primary Update Center operated by the Jenkins project [4: updates.jenkins.io], or a custom Update Center.

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 **Manage Jenkins** > **Manage Plugins** view, available to administrators of a Jenkins environment.

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



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

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

Using the Jenkins CLI

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

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

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

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

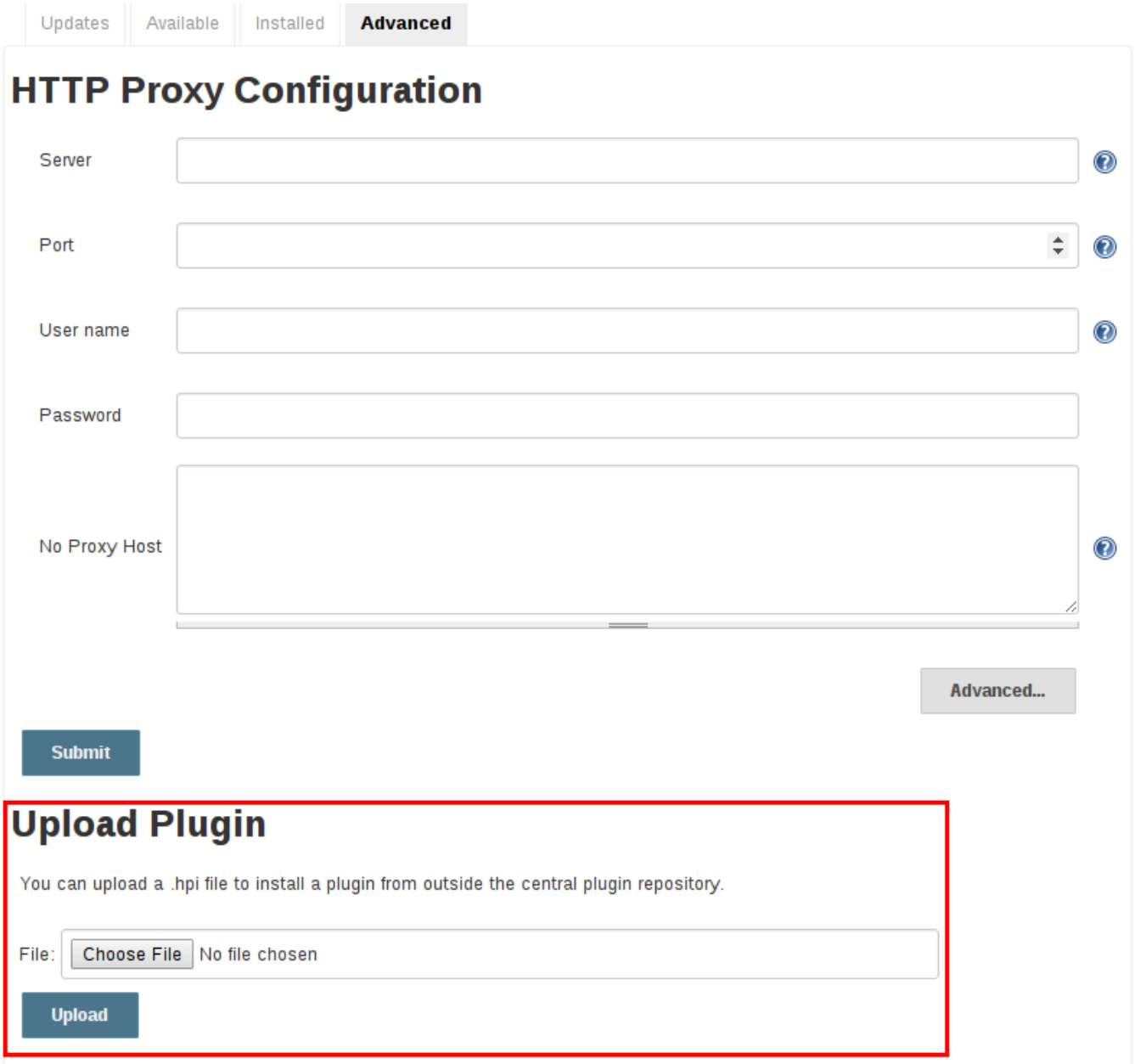
Advanced installation

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

From the web UI

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

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



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

On the master

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

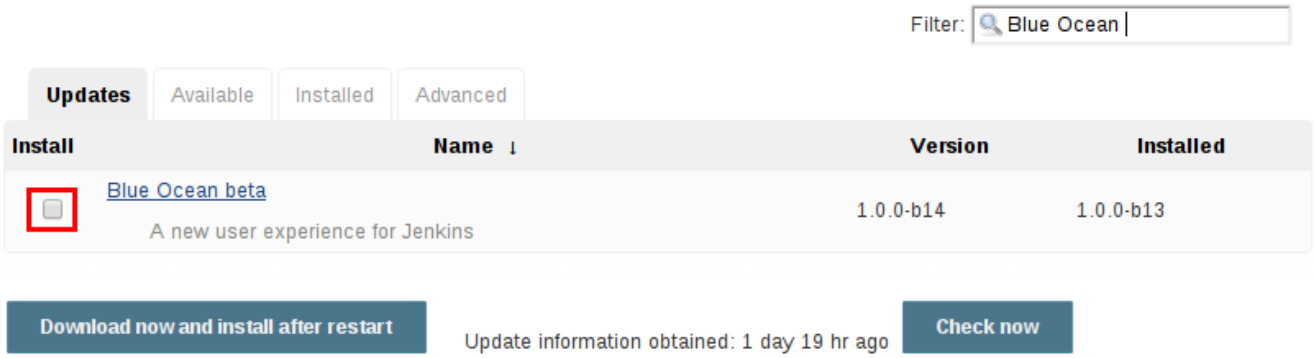
Copy the downloaded `.hpi`` file into the `JENKINS_HOME/plugins` directory on the Jenkins master (for example, on Debian systems `JENKINS_HOME` is generally `/var/lib/jenkins`).

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

NOTE	The names of the plugin directories in the Update Site [4: updates.jenkins.io] 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.



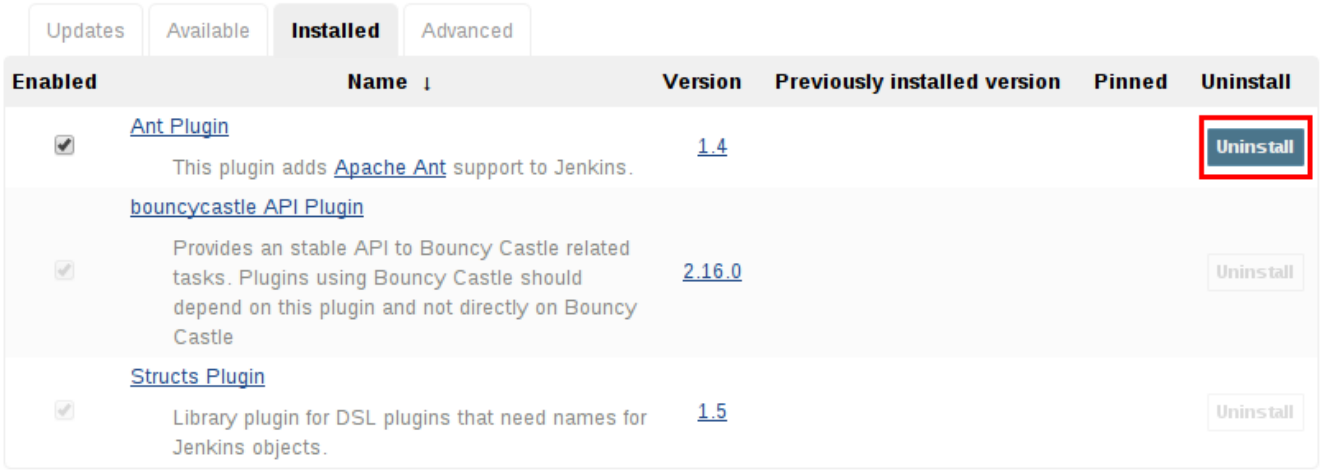
By default, the Jenkins master will check for updates from the Update Center once every 24 hours. To manually trigger a check for updates, simply click on the **Check now** button in the **Updates** tab.

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 plugin:matrix-auth[**Matrix Authorization plugin**].

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

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

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

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

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

Jenkins CLI

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

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

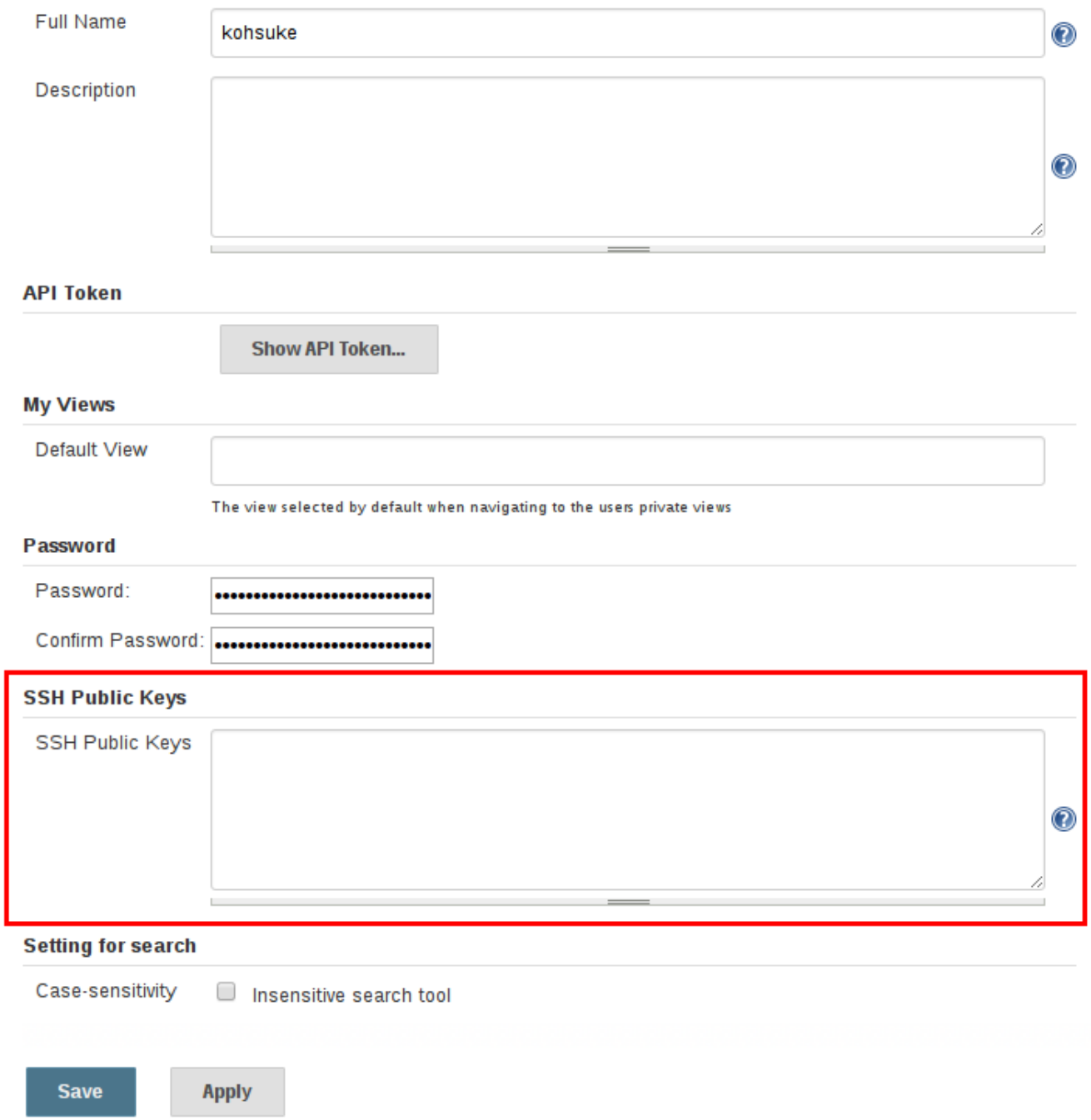
```
% curl -Lv https://JENKINS_URL/login 2>&1 | grep 'X-SSH-Endpoint'
< X-SSH-Endpoint: localhost:53801
%
```

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.
 JOB : Name of the job to build
 -c  : Check for SCM changes before starting the build, and if there's no
       change, exit without doing a build
 -f  : Follow the build progress. Like -s only interrupts are not passed
       through to the build.
 -p  : Specify the build parameters in the key=value format.
 -s  : Wait until the completion/abortion of the command. Interrupts are passed
       through to the build.
 -v  : Prints out the console output of the build. Use with -s
 -w  : Wait until the start of the command
% ssh -l kohsuke -p 53801 localhost build build-all-software -f -v
Started build-all-software #1
Started from command line by admin
Building in workspace /tmp/jenkins/workspace/build-all-software
[build-all-software] $ /bin/sh -xe /tmp/hudson1100603797526301795.sh
+ echo hello world
hello world
Finished: SUCCESS
Completed build-all-software #1 : SUCCESS
%
```

console

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

```
% ssh -l kohsuke -p 53801 localhost help console

java -jar jenkins-cli.jar console JOB [BUILD] [-f] [-n N]
Produces the console output of a specific build to stdout, as if you are doing 'cat
build.log'
 JOB   : Name of the job
 BUILD : Build number or permalink to point to the build. Defaults to the last
         build
 -f    : If the build is in progress, stay around and append console output as
         it comes, like 'tail -f'
 -n N  : Display the last N lines
% ssh -l kohsuke -p 53801 localhost console build-all-software
Started from command line by kohsuke
Building in workspace /tmp/jenkins/workspace/build-all-software
[build-all-software] $ /bin/sh -xe /tmp/hudson1100603797526301795.sh
+ echo hello world
yes
Finished: SUCCESS
%
```

who-am-i

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

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

java -jar jenkins-cli.jar who-am-i
Reports your credential and permissions.
% ssh -l kohsuke -p 53801 localhost who-am-i
Authenticated as: kohsuke
Authorities:
  authenticated
%
```

Using the CLI client

While the SSH-based CLI is 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)
    Suppressed: java.io.IOException: Server returned HTTP response code: 403 for URL:
http://citest.gce.px/cli
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.java:184
0)
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1441
)
        at hudson.cli.FullDuplexHttpStream.<init>(FullDuplexHttpStream.java:78)
        at hudson.cli.CLI.connectViaHttp(CLI.java:152)
        at hudson.cli.CLI.<init>(CLI.java:132)
        ... 3 more
```

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 Pipelines to writing Pipeline code, and even extending Pipeline itself.

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 any ①

    stages {
        stage('Build') { ②
            steps { ③
                sh 'make' ④
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml' ⑤
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
// Script //
node {
    stage('Build') {
        sh 'make'
    }

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

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

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

Pipeline.

② `stage` describes a stage of this Pipeline.

③ `steps` describes the steps to be run in this `stage`

④ `sh` executes the given shell command

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

Why Pipeline?

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

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

While Jenkins has always allowed rudimentary forms of chaining Freestyle Jobs together to perform sequential tasks, [6: Additional plugins have been used to implement complex behaviors utilizing Freestyle Jobs such as the Copy Artifact, Parameterized Trigger, and Promoted Builds plugins] Pipeline makes this concept a first-class citizen in Jenkins.

Building on the core Jenkins value of extensibility, Pipeline is also extensible both by users with Pipeline Shared Libraries and by plugin developers. [7: plugin:github-organization-folder[GitHub Organization Folder plugin]]

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

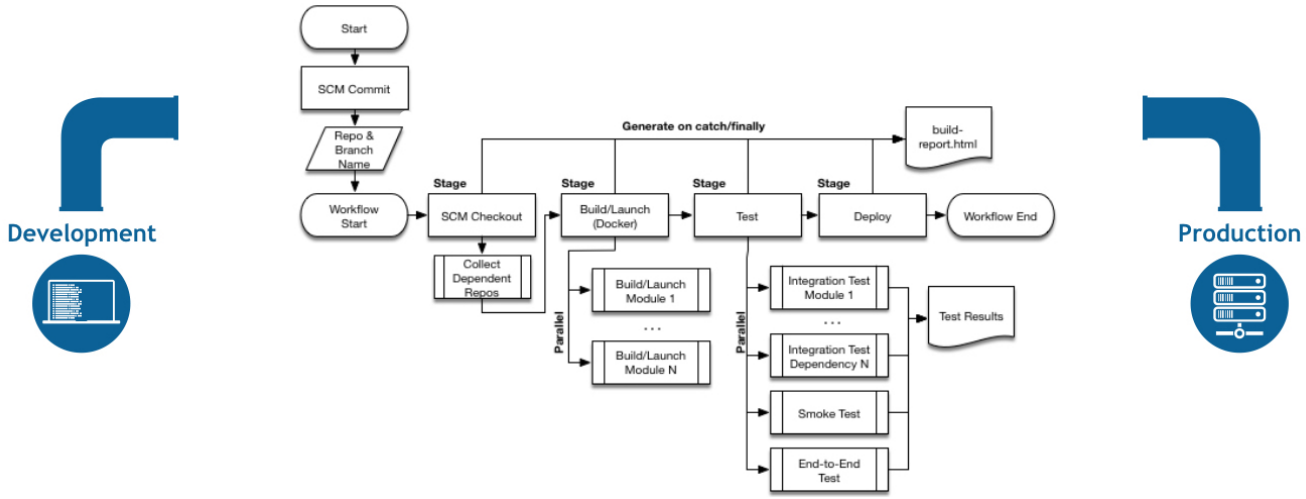


Figure 2. Pipeline Flow

Pipeline Terms

Step

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

Node

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

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

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

Stage

`stage` is a step for defining a conceptually distinct subset of the entire Pipeline, for example: "Build", "Test", and "Deploy", which is used by many plugins to visualize or present Jenkins Pipeline status/progress. [8: Blue Ocean, Pipeline Stage View plugin]

Getting Started with Pipeline

Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline DSL. [9: Domain-Specific Language]

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 [10: 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:

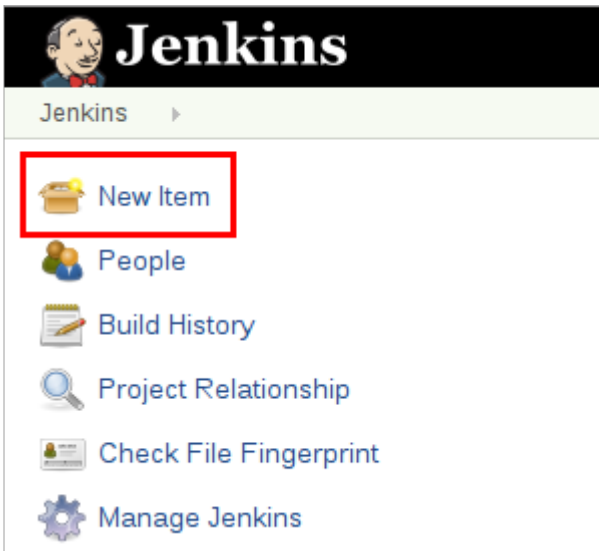
- By entering a script directly in the Jenkins web UI.
- By creating a `Jenkinsfile` which can be checked into a project's source control repository.

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. [11: en.wikipedia.org/wiki/Source_control_management]

Defining a Pipeline in the Web UI

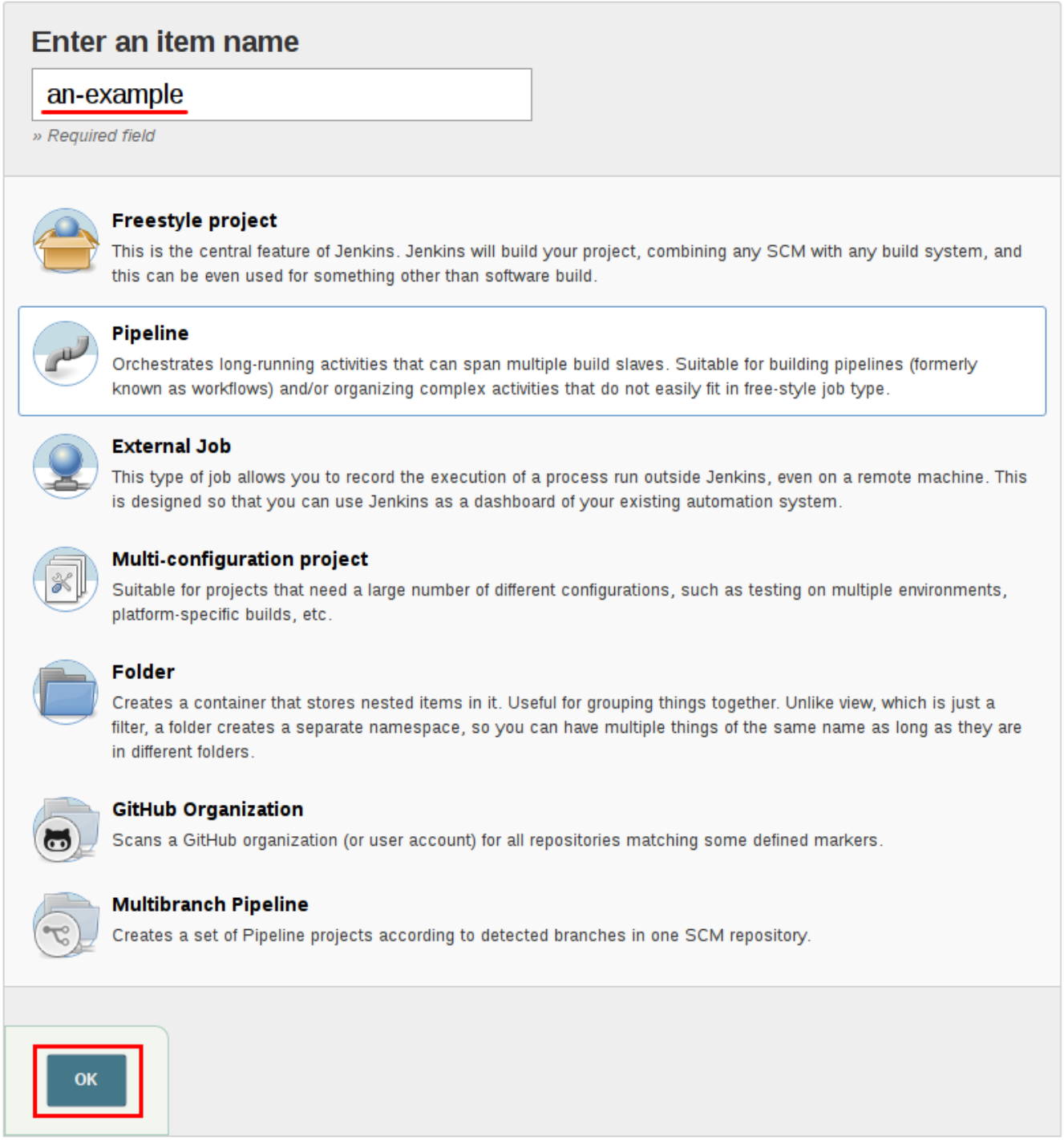
To create a basic Pipeline in the Jenkins web UI, follow these steps:

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

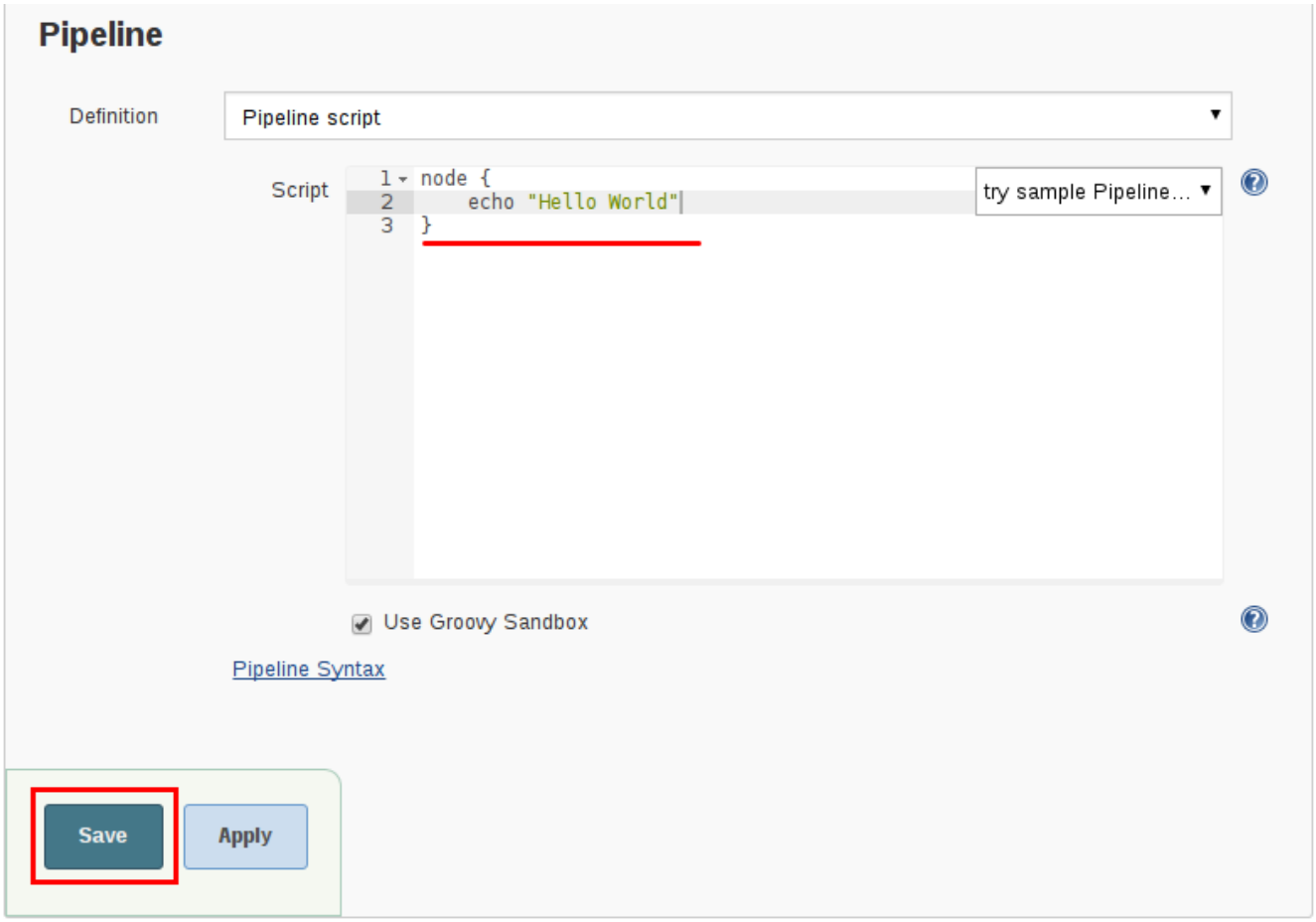


- Enter a name for your Pipeline, select **Pipeline** and click **OK**.

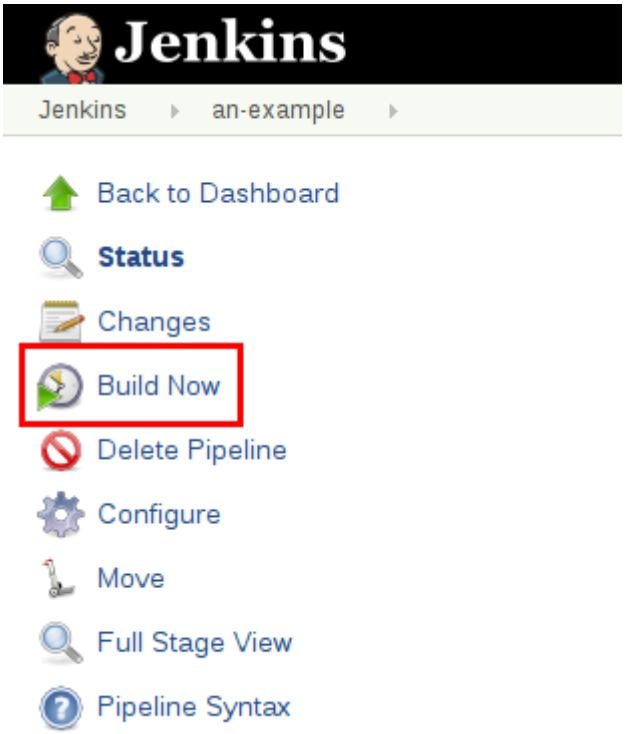
CAUTION	Jenkins uses the name of the Pipeline to create directories on disk. Pipeline names which include spaces may uncover bugs in scripts which do not expect paths to contain spaces.



- 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.


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

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

```
// Script //
node { ①
    echo 'Hello World' ②
}
// Declarative not yet implemented //
```

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

② `echo` writes simple string in the Console Output.

Defining a Pipeline in SCM

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

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

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

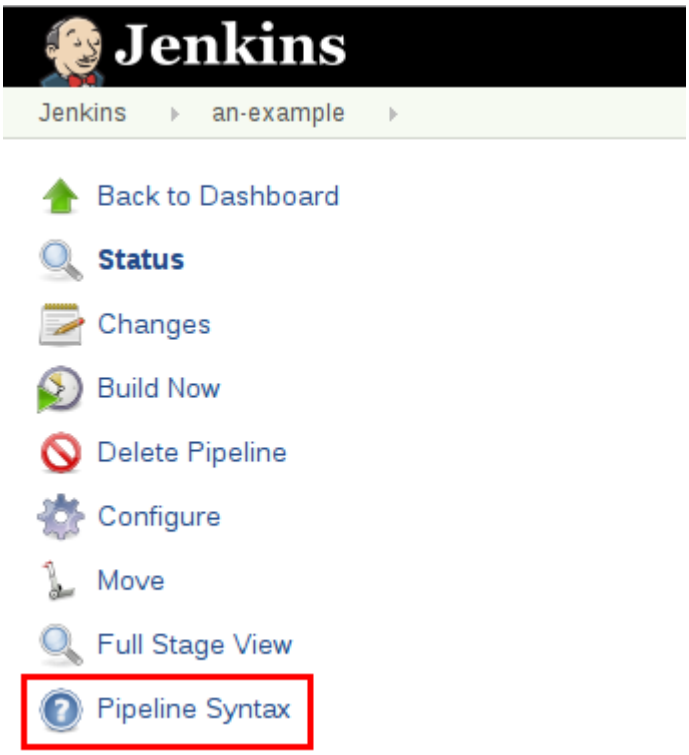
TIP

The first line of a `Jenkinsfile` should be `#!groovy` [13: 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

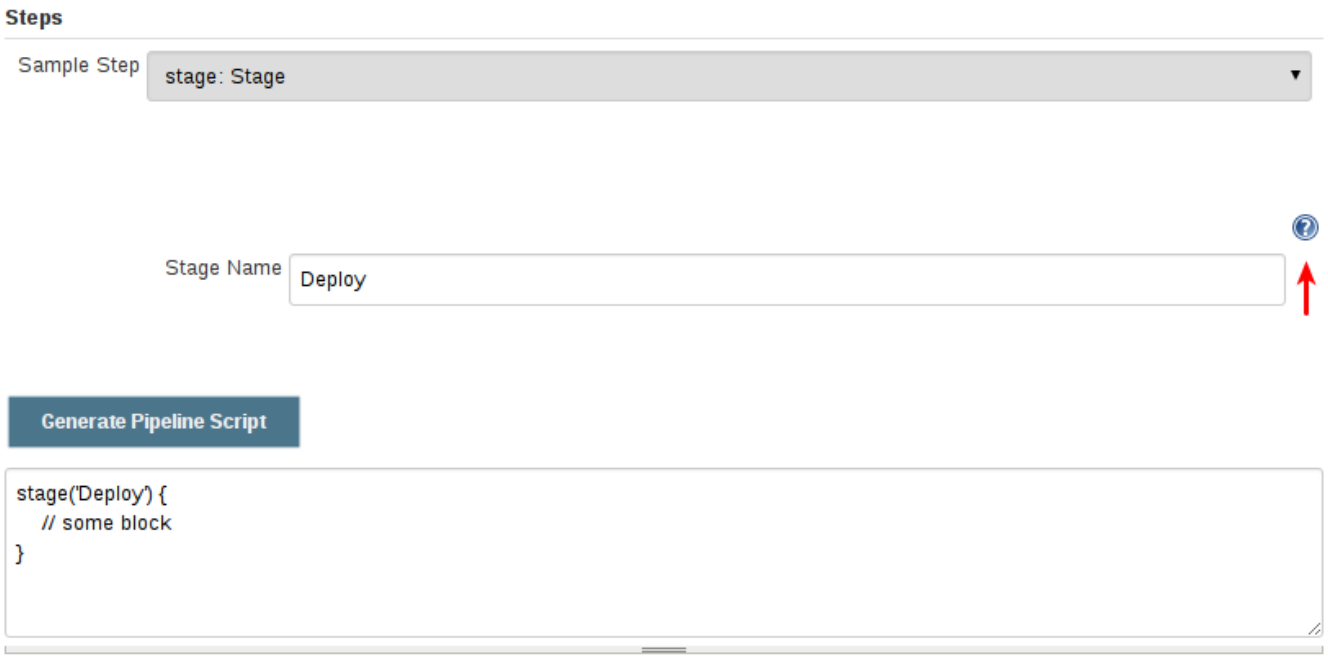
The built-in "Snippet Generator" utility is helpful for creating bits of code for individual steps, discovering new steps provided by plugins, or experimenting with different parameters for a particular step.

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.

Using a Jenkinsfile

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

Creating a `Jenkinsfile`, which is checked into source control [14: en.wikipedia.org/wiki/Source_control_management], provides a number of immediate benefits:

- Code review/iteration on the Pipeline
- Audit trail for the Pipeline
- Single source of truth [15: en.wikipedia.org/wiki/Single_Source_of_Truth] for the Pipeline, which can be viewed and edited by multiple members of the project.

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

Creating a Jenkinsfile

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

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

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

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

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

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

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

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

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

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

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

Build

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

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

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

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

① The `sh` step invokes the `make` command and will only continue if a zero exit code is returned by the command. Any non-zero exit code will fail the Pipeline.

② `archiveArtifacts` captures the files built matching the include pattern (`*/target/.jar`) and saves them to the Jenkins master for later retrieval.

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

Test

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

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

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

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

① Using an inline shell conditional (`sh 'make || true'`) ensures that the `sh` step always sees a zero exit code, giving the `junit` step the opportunity to capture and process the test reports. Alternative approaches to this are covered in more detail in the [handling-failures] section below.

② `junit` captures and associates the JUnit XML files matching the inclusion pattern (`*/target/.xml`).

Deploy

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

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

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

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

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

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

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

Advanced Syntax for Pipeline

String Interpolation

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

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

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

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

Would result in:

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

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

Working with the Environment

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

BUILD_ID

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

JOB_NAME

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

JENKINS_URL

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

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

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

Setting environment variables

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

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

```
// Declarative //
pipeline {
    agent any
    environment { ①
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment { ②
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
// Script //
node {
    /* .. snip .. */
    withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {
        sh 'mvn -B verify'
    }
}
```

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

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

Parameters

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

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

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

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

node {
    echo "${params.Greeting} World!"
}
```

Handling Failures

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

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

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

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

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

Using multiple agents

In all the previous examples, only a single agent has been used. This means Jenkins will allocate an

executor wherever one is available, regardless of how it is labeled or configured. Not only can this behavior be overridden, but Pipeline allows utilizing multiple agents in the Jenkins environment from within the *same* `Jenkinsfile`, which can helpful for more advanced use-cases such as executing builds/tests across multiple platforms.

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

```
// Declarative //
pipeline {
    agent none
    stages {
        stage('Build') {
            agent any
            steps {
                checkout scm
                sh 'make'
                stash includes: '**/target/*.jar', name: 'app' ①
            }
        }
        stage('Test on Linux') {
            agent { ②
                label 'linux'
            }
            steps {
                unstash 'app' ③
                sh 'make check'
            }
            post {
                always {
                    junit '**/target/*.xml'
                }
            }
        }
        stage('Test on Windows') {
            agent {
                label 'windows'
            }
            steps {
                unstash 'app'
                bat 'make check' ④
            }
            post {
                always {
                    junit '**/target/*.xml'
                }
            }
        }
    }
```

```
}
// Script //
stage('Build') {
    node {
        checkout scm
        sh 'make'
        stash includes: '**/target/*.jar', name: 'app' ①
    }
}

stage('Test') {
    node('linux') { ②
        checkout scm
        try {
            unstash 'app' ③
            sh 'make check'
        }
        finally {
            junit '**/target/*.xml'
        }
    }
    node('windows') {
        checkout scm
        try {
            unstash 'app'
            bat 'make check' ④
        }
        finally {
            junit '**/target/*.xml'
        }
    }
}
```

① The `stash` step allows capturing files matching an inclusion pattern (`*/target/.jar`) for reuse within the *same* Pipeline. Once the Pipeline has completed its execution, stashed files are deleted from the Jenkins master.

② The parameter in `agent`/`node` allows for any valid Jenkins label expression. Consult the Pipeline Syntax section for more details.

③ `unstash` will retrieve the named "stash" from the Jenkins master into the Pipeline's current workspace.

④ The `bat` script allows for executing batch scripts on Windows-based platforms.

Optional step arguments

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

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

functionally equivalent:

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

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

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

Advanced Scripted Pipeline

Scripted Pipeline is a domain-specific language [16: en.wikipedia.org/wiki/Domain-specific_language] based on Groovy, most Groovy syntax can be used in Scripted Pipeline without modification.

Executing in parallel

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

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

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

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

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

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

Branches and Pull Requests

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

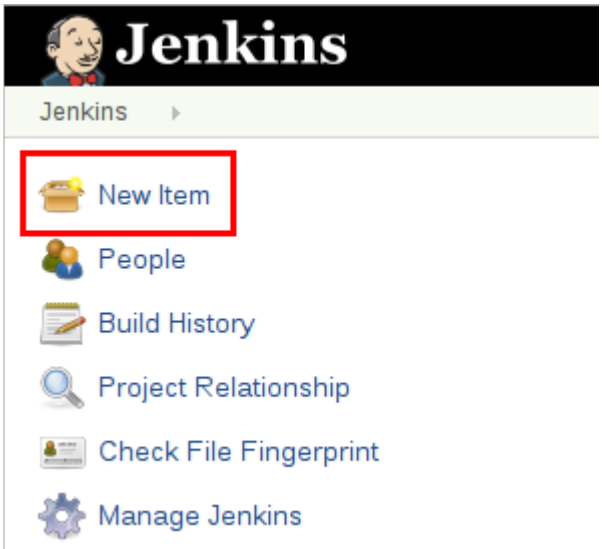
Creating a Multibranch Pipeline

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

This eliminates the need for manual Pipeline creation and management.

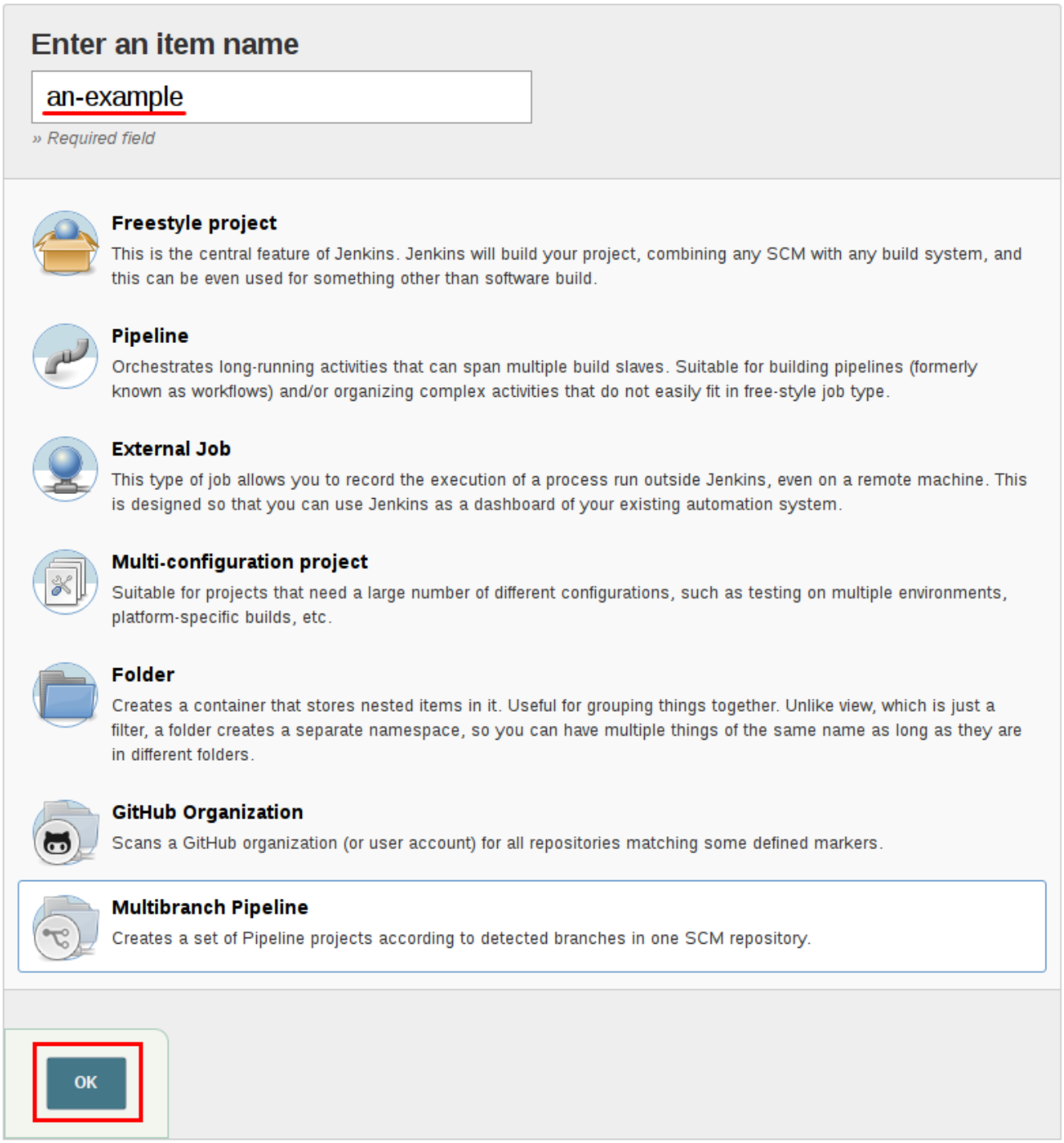
To create a Multibranch Pipeline:

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

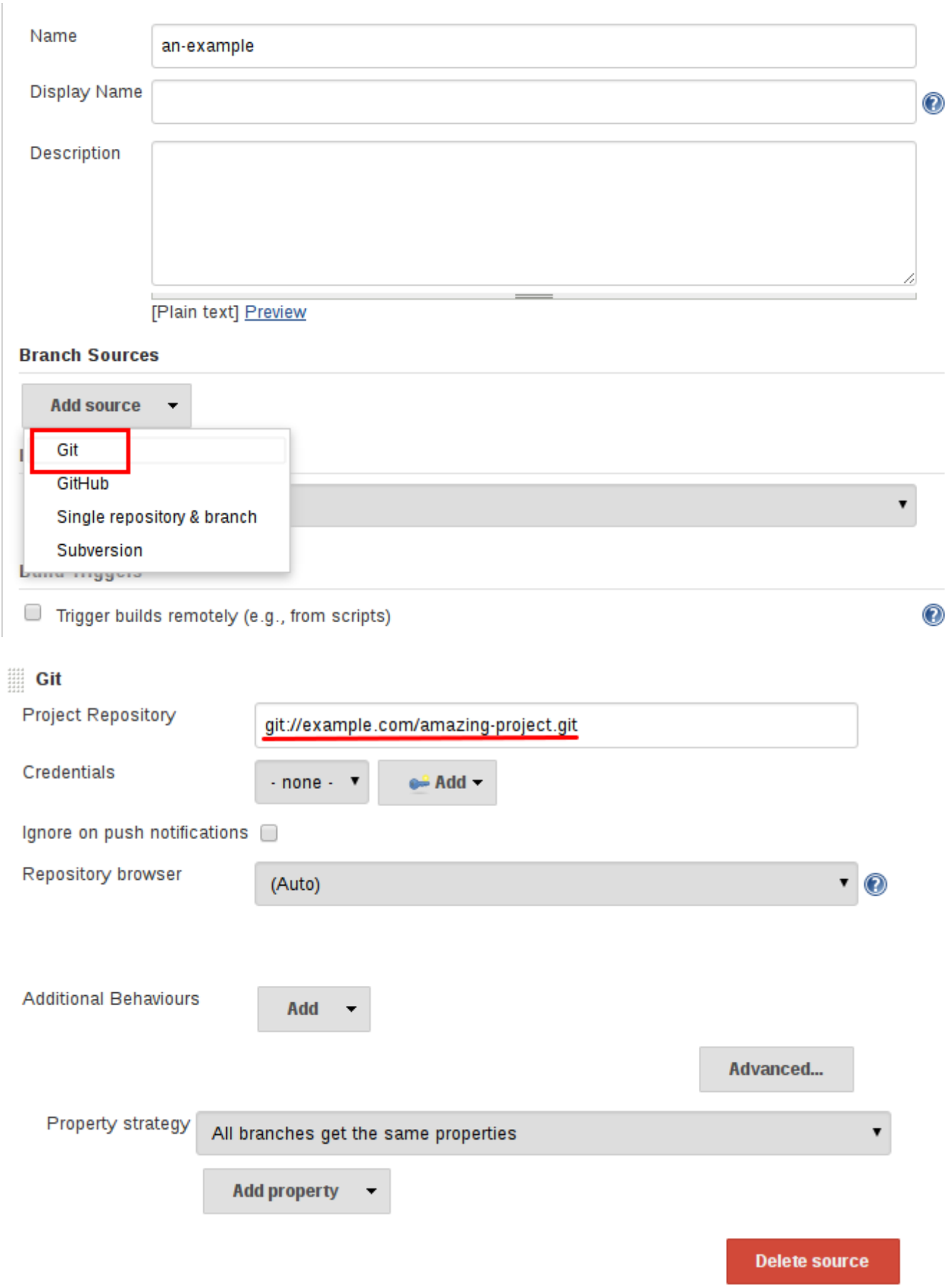


- Enter a name for your Pipeline, select **Multibranch Pipeline** and click **OK**.

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



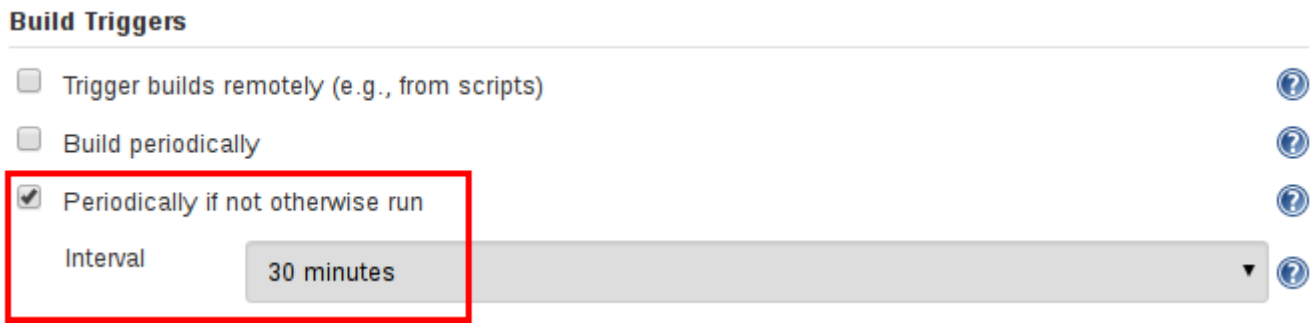
- Add a **Branch Source** (for example, Git) and enter the location of the repository.



- **Save** the Multibranch Pipeline project.

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

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



Additional Environment Variables

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

BRANCH_NAME

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

CHANGE_ID

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

Additional environment variables are listed in the Global Variable Reference.

Supporting Pull Requests

With the "GitHub" or "Bitbucket" Branch Sources, Multibranch Pipelines can be used for validating pull/change requests. This functionality is provided, respectively, by the plugin:github-branch-source[GitHub Branch Source] and plugin:cloudbees-bitbucket-branch-source[Bitbucket Branch Source] plugins. Please consult their documentation for further information on how to use those plugins.

Using Organization Folders

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

Currently, this functionality exists only for GitHub and Bitbucket, with functionality provided by the plugin:github-organization-folder[GitHub Organization Folder] and plugin:cloudbees-bitbucket-branch-source[Bitbucket Branch Source] plugins.

Extending with Shared Libraries

As Pipeline is adopted for more and more projects in an organization, common patterns are likely to emerge. Oftentimes it is useful to share parts of Pipelines between various projects to reduce redundancies and keep code "DRY" [17: en.wikipedia.org/wiki/Don't_repeat_yourself].

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

Defining Shared Libraries

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

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

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

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

Directory structure

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

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

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

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

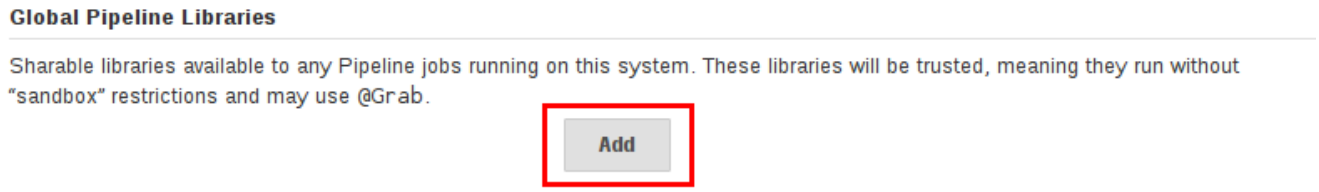
The Groovy source files in these directories get the same "CPS transformation" as in Scripted Pipeline.

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

Other directories under the root are reserved for future enhancements.

Global Shared Libraries

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



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

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

Folder-level Shared Libraries

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

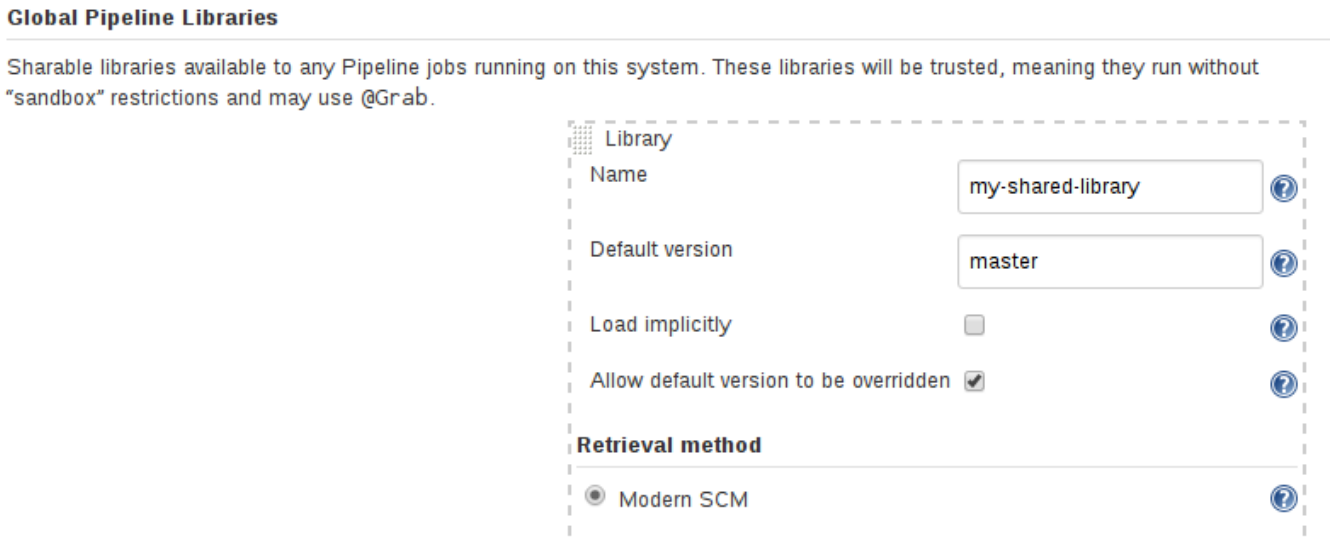
Folder-based libraries are not considered "trusted:" they run in the Groovy sandbox just like typical Pipelines.

Automatic Shared Libraries

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

Using libraries

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




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

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

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

TIP

For Shared Libraries which only define Global Variables (<code>vars/</code>), or a <code>Jenkinsfile</code> which only needs a Global Variable, the annotation pattern <code>@Library('my-shared-library') _</code> may be useful for keeping code concise. In essence, instead of annotating an unnecessary <code>import</code> statement, the symbol <code></code> is annotated.

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

Libraries are resolved and loaded during *compilation* of the script, before it starts executing. This allows the Groovy compiler to understand the meaning of symbols used in static type checking, and

permits them to be used in type declarations in the script, for example:

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

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

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

Global Variables however, are resolved at runtime.

Loading libraries dynamically

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

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

```
library 'my-shared-library'
```

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

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

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

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

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

```
def useSomeLib(helper) { // dynamic: cannot declare as Helper
    helper.prepare()
    return helper.count()
}

def lib = library('my-shared-library').com.mycorp.pipeline // preselect the package

echo useSomeLib(lib.Helper.new(lib.Constants.SOME_TEXT))
```

Library versions

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

If "Allow default version to be overridden" is enabled in the Shared Library's configuration, a `@Library` annotation may also override a default version defined for the library. This also allows a library with "Load implicitly" to be loaded from a different version if necessary.

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

```
library 'my-shared-library@master'
```

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

```
library "my-shared-library@$BRANCH_NAME"
```

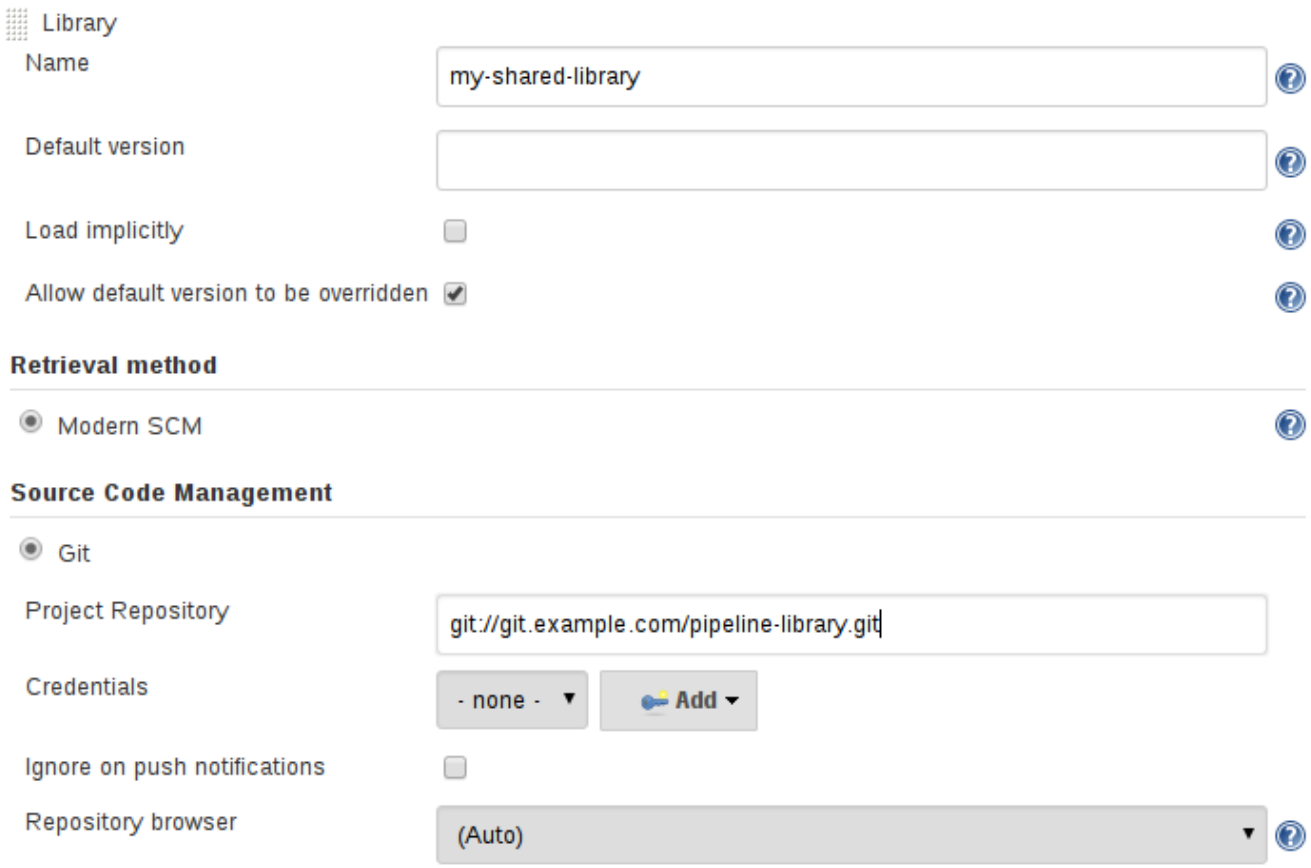
would load a library using the same SCM branch as the multibranch `Jenkinsfile`. As another example, you could pick a library by parameter:

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

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

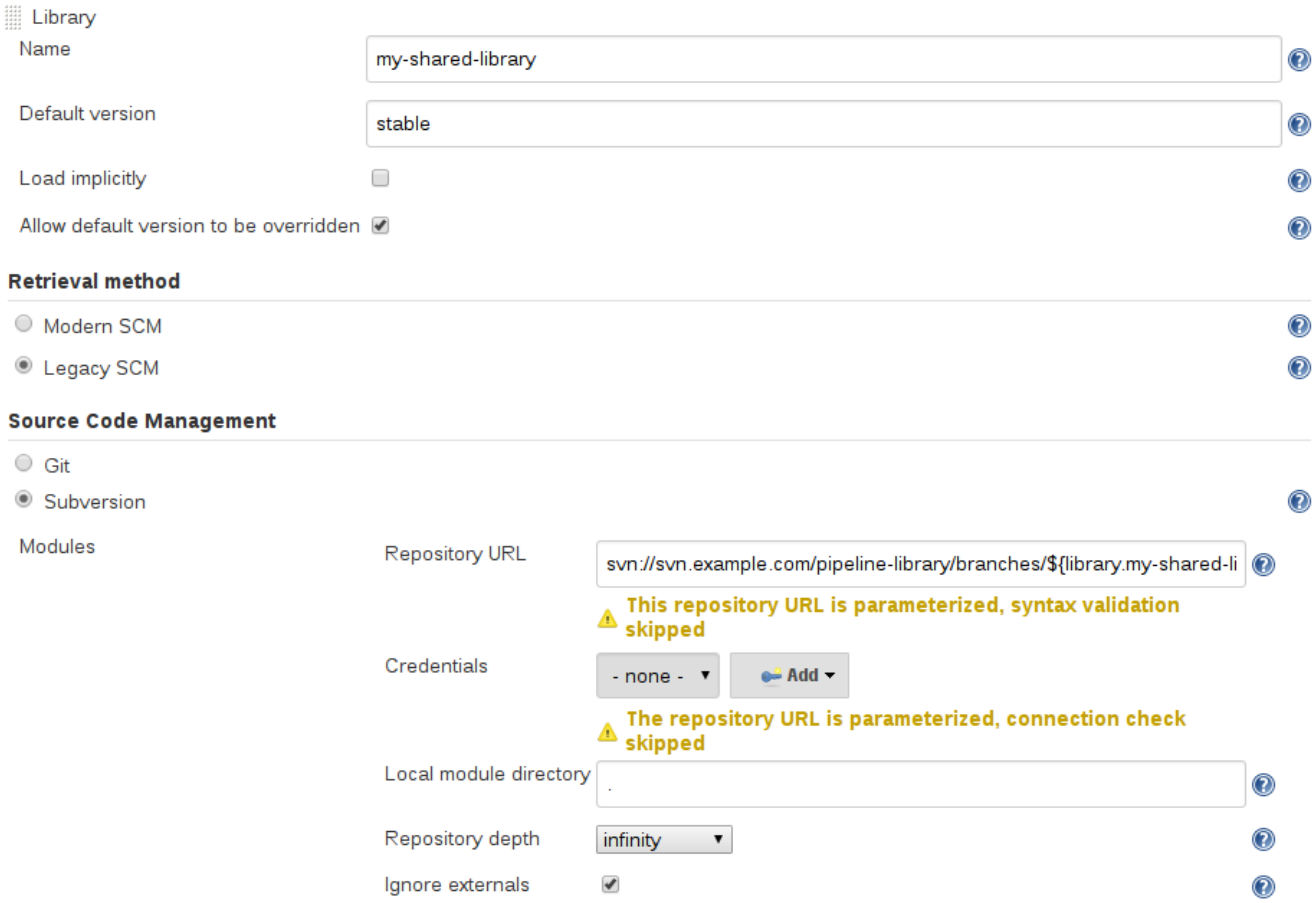
Retrieval Method

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



Legacy SCM

SCM plugins which have not yet been updated to support the newer features required by Shared Libraries, may still be used via the **Legacy SCM** option. In this case, include `${library.yourlibrarynamehere.version}` wherever a branch/tag/ref may be configured for that particular SCM plugin. This ensures that during checkout of the library's source code, the SCM plugin will expand this variable to checkout the appropriate version of the library.



Dynamic retrieval

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

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

```
library identifier: 'custom-lib@master', retriever: modernSCM(
  [$class: 'GitSCMSource',
   remote: 'git@git.mycorp.com:my-jenkins-utils.git',
   credentialsId: 'my-private-key'])
```

It is best to refer to **Pipeline Syntax** for the precise syntax for your SCM.

Note that the library version *must* be specified in these cases.

Writing libraries

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

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

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

Accessing steps

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

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

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

Which can then be called from a Scripted Pipeline:

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

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

Alternately, a set of `steps` can be passed explicitly to a library class, in a constructor, or just one method:

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

When saving state on classes, such as above, the class **must** implement the `Serializable` interface.

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

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

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

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

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

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

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

Defining global variables

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

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

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

```
// vars/acme.groovy
class acme implements Serializable {
    private String name
    def setName(value) {
        name = value
    }
    def getName() {
        name
    }
    def caution(message) {
        echo "Hello, ${name}! CAUTION: ${message}"
    }
}
```

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

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

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

Defining steps

Shared Libraries can also define global variables which behave similarly to built-in steps, such as `sh` or `git`. Global variables defined in Shared Libraries **must** be named with all lower-case or "camelCased" in order to be loaded properly by Pipeline. [18: 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 Pipelines that are mostly similar, the global variable mechanism provides a handy tool to build a higher-level DSL that captures the similarity. For example, all Jenkins plugins are built and tested in the same way, so we might write a step named `buildPlugin`:

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

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

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

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

Using third-party libraries

It is possible to use third-party Java libraries, typically found in Maven Central, 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*.

Pipeline Syntax

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

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

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

Declarative Pipeline

Declarative Pipeline is a relatively recent addition to Jenkins Pipeline [19: Version 2.5 of the "Pipeline plugin" introduces support for Declarative Pipeline syntax] which presents a more simplified and opinionated syntax on top of the Pipeline sub-systems.

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

```
pipeline {
    /* insert Declarative Pipeline here */
}
```

The basic statements and expressions which are valid in Declarative Pipeline follow the same rules as Groovy's syntax with the following exceptions:

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

Sections

Sections in Declarative Pipeline typically contain one or more Directives or Steps.

post

The `post` section defines actions which will be run at the end of the Pipeline run. A number of additional Conditions blocks are supported within the `post` section: `always`, `changed`, `failure`, `success`, and `unstable`. These blocks allow for the execution of steps at the tail-end of the Pipeline run, depending on the status of the Pipeline.

Required	No
Parameters	*None*
Allowed	In the top-level `pipeline` block and each `stage` block.

Conditions

`always`

Run regardless of the completion status of the Pipeline run.

`changed`

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

Pipeline.

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

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

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

Example

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
    post { ①
        always { ②
            echo 'I will always say Hello again!'
        }
    }
}
// Script //
```

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

② The Conditions blocks can use steps.

stages

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

Required	Yes
Parameters	*None*

Allowed	Only once, inside the `pipeline` block.

Example

```
// Declarative //
pipeline {
    agent any
    stages { ①
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
// Script //
```

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

steps

Defines a series of steps to be executed in a given `stage` directive.

Required	Yes
Parameters	*None*
Allowed	Inside each `stage` block.

Example

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example') {
            steps { ①
                echo 'Hello World'
            }
        }
    }
}
// Script //
```

① The `steps` section must contain one or more steps.

Directives

agent

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

Required	Yes
Parameters	Described below
Allowed	In the top-level `pipeline` block and each `stage` block.

Parameters

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

any

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

none

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

label

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

node

`agent { node { label 'labelName' } }` behaves the same as `agent { label 'labelName' }`, but `node` allows for additional options (such as `customWorkspace`).

docker

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

```
agent {
    docker {
        image 'maven:3-alpine'
        label 'my-defined-label'
        args  '-v /tmp:/tmp'
    }
}
```

dockerfile

Execute the Pipeline, or stage, with a container built from a `Dockerfile` contained in the source repository. Conventionally this is the `Dockerfile` in the root of the source repository: `agent { dockerfile true }`. If building a `Dockerfile` in another directory, use the `dir` option: `agent { dockerfile { dir 'someSubDir' } }`.

Common Options

These are a few options for two or more `agent` implementations. They are not required unless explicitly stated.

label

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

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

customWorkspace

A string. Run the Pipeline or individual `stage` this `agent` is applied to within this custom workspace, rather than the default. It can be either a relative path, in which case the custom workspace will be under the workspace root on the node, or an absolute path. For example:

```
agent {
    node {
        label 'my-defined-label'
        customWorkspace '/some/other/path'
    }
}
```

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

reuseNode

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

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

Example

```
// Declarative //
pipeline {
    agent { docker 'maven:3-alpine' } ①
    stages {
        stage('Example Build') {
            steps {
                sh 'mvn -B clean verify'
            }
        }
    }
}
// Script //
```

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

Stage-level `agent` directive

```
// Declarative //
pipeline {
    agent none ①
    stages {
        stage('Example Build') {
            agent { docker 'maven:3-alpine' } ②
            steps {
                echo 'Hello, Maven'
                sh 'mvn --version'
            }
        }
        stage('Example Test') {
            agent { docker 'openjdk:8-jre' } ③
            steps {
                echo 'Hello, JDK'
                sh 'java -version'
            }
        }
    }
}
// Script //
```

① Defining `agent none` at the top-level of the Pipeline ensures that an Executor will not be unnecessarily. Using `agent none` requires that each `stage` directive contain an `agent` directive.

② Execute the steps contained within this stage using the given container.

③ Execute the steps contained within this steps using a different image from the previous stage.

environment

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

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

Required	No
Parameters	*None*
Allowed	Inside the `pipeline` block, or within `stage` directives.

Example

```
// Declarative //
pipeline {
    agent any
    environment { ①
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment { ②
                AN_ACCESS_KEY = credentials('my-prefined-secret-text') ③
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
// Script //
```

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

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

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

options

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

Required	No
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

Available Options

buildDiscarder

Persist artifacts and console output for the specific number of recent Pipeline runs. For example: `options { buildDiscarder(logRotator(numToKeepStr: '1')) }`

disableConcurrentBuilds

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

skipDefaultCheckout

Skip checking out code from source control by default in the `agent` directive. For example: `options { skipDefaultCheckout() }`

timeout

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

retry

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

timestamps

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

Example

```
// Declarative //
pipeline {
    agent any
    options {
        timeout(time: 1, unit: 'HOURS') ①
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
// Script //
```

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

NOTE A comprehensive list of available options is pending the completion of INFRA-1503.

parameters

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

Required	No
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

Available Parameters

string

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

booleanParam

A boolean parameter, for example: `parameters { booleanParam(name: 'DEBUG_BUILD', defaultValue: true, description: '') }`

Example

```
// Declarative //
pipeline {
    agent any
    parameters {
        string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I
say hello to?')
    }
    stages {
        stage('Example') {
            steps {
                echo "Hello ${params.PERSON}"
            }
        }
    }
}
// Script //
```

NOTE	A comprehensive list of available parameters is pending the completion of INFRA-1503.

triggers

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

Required	No
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

cron

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

pollSCM

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

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

Example

```
// Declarative //
pipeline {
    agent any
    triggers {
        cron('H 4/* 0 0 1-5')
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
// Script //
```

stage

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

Required	At least one
Parameters	One mandatory parameter, a string for the name of the stage.
Allowed	Inside the `stages` section.

Example

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
// Script //
```

tools

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

Required	No
Parameters	*None*
Allowed	Inside the `pipeline` block or a `stage` block.

Supported Tools

maven

jdk

gradle

Example

```
// Declarative //
pipeline {
    agent any
    tools {
        maven 'apache-maven-3.0.1' ①
    }
    stages {
        stage('Example') {
            steps {
                sh 'mvn --version'
            }
        }
    }
}
// Script //
```

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

when

The `when` directive allows the Pipeline to determine whether the stage should be executed depending on the given condition.

Required	No
Parameters	*None*

Allowed	Inside a `stage` directive

Built-in Conditions

branch

Execute the stage when the branch being built matches the branch pattern given, for example: `when { branch 'master' }`

environment

Execute the stage when the specified environment variable is set to the given value, for example: `when { environment name: 'DEPLOY_TO', value: 'production' }`

expression

Execute the stage when the specified Groovy expression evaluates to true, for example: `when { expression { return params.DEBUG_BUILD } }`

Example

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                branch 'production'
            }
                echo 'Deploying'
            }
        }
    }
}
// Script //
```

Steps

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

script

The `script` step takes a block of [scripted-pipeline] and executes that in the Declarative Pipeline. For

most use-cases, the `script` step should be unnecessary in Declarative Pipelines, but it can provide a useful "escape hatch." `script` blocks of non-trivial size and/or complexity should be moved into Shared Libraries instead.

Example

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'

                script {
                    def browsers = ['chrome', 'firefox']
                    for (int i = 0; i < browsers.size(); ++i) {
                        echo "Testing the ${browsers[i]} browser"
                    }
                }
            }
        }
    }
}
// Script //
```

Scripted Pipeline

Scripted Pipeline, like [declarative-pipeline], is built on top of the underlying Pipeline sub-system. Unlike Declarative, Scripted Pipeline is effectively a general purpose DSL [20: Domain-specific Language] built with Groovy. Most functionality provided by the Groovy language is made available to users of Scripted Pipeline, which means it can be a very expressive and flexible tool with which one can author continuous delivery pipelines.

Flow Control

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

```
// Scripted //
node {
    stage('Example') {
        if (env.BRANCH_NAME == 'master') {
            echo 'I only execute on the master branch'
        } else {
            echo 'I execute elsewhere'
        }
    }
}
// Declarative //
```

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

```
// Scripted //
node {
    stage('Example') {
        try {
            sh 'exit 1'
        }
        catch (exc) {
            echo 'Something failed, I should sound the klaxons!'
            throw
        }
    }
}
// Declarative //
```

Steps

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

Scripted Pipeline does **not** introduce any steps which are specific to its syntax; Pipeline Steps reference which contains a comprehensive list of steps provided by Pipeline and plugins.

Differences from plain Groovy

In order to provide *durability*, which means that running Pipelines can survive a restart of the Jenkins master, Scripted Pipeline must serialize data back to the master. Due to this design requirement, some Groovy idioms such as `collection.each { item -> /* perform operation */ }` are not fully supported. See JENKINS-27421 and JENKINS-26481 for more information.

Syntax Comparison

When Jenkins Pipeline was first created, Groovy was selected as the foundation. Jenkins has long shipped with an embedded Groovy engine to provide advanced scripting capabilities for admins and users alike. Additionally, the implementors of Jenkins Pipeline found Groovy to be a solid foundation upon which to build what is now referred to as the "Scripted Pipeline" DSL. [5: Domain-Specific Language].

As it is a fully featured programming environment, Scripted Pipeline offers a tremendous amount of flexibility and extensibility to Jenkins users. The Groovy learning-curve isn't typically desirable for all members of a given team, so Declarative Pipeline was created to offer a simpler and more opinionated syntax for authoring Jenkins Pipeline.

The two are both fundamentally the same Pipeline sub-system underneath. They are both durable implementations of "Pipeline as code." They are both able to use steps built into Pipeline or provided by plugins. Both are able utilize Shared Libraries

Where they differ however is in syntax and flexibility. Declarative limits what is available to the user with a more strict and pre-defined structure, making it an ideal choice for simpler continuous delivery pipelines. Scripted provides very few limits, insofar that the only limits on structure and syntax tend to be defined by Groovy itself, rather than any Pipeline-specific systems, making it an ideal choice for power-users and those with more complex requirements. As the name implies, Declarative Pipeline is encourages a declarative programming model. [21: Declarative Programming] Whereas Scripted Pipelines follow a more imperative programming model.. [22: Imperative Programming]

Blue Ocean User Experience

This chapter will cover all aspects of Jenkins Blue Ocean, from view Pipelines to writing Pipeline code, and even extending Pipeline itself.

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.

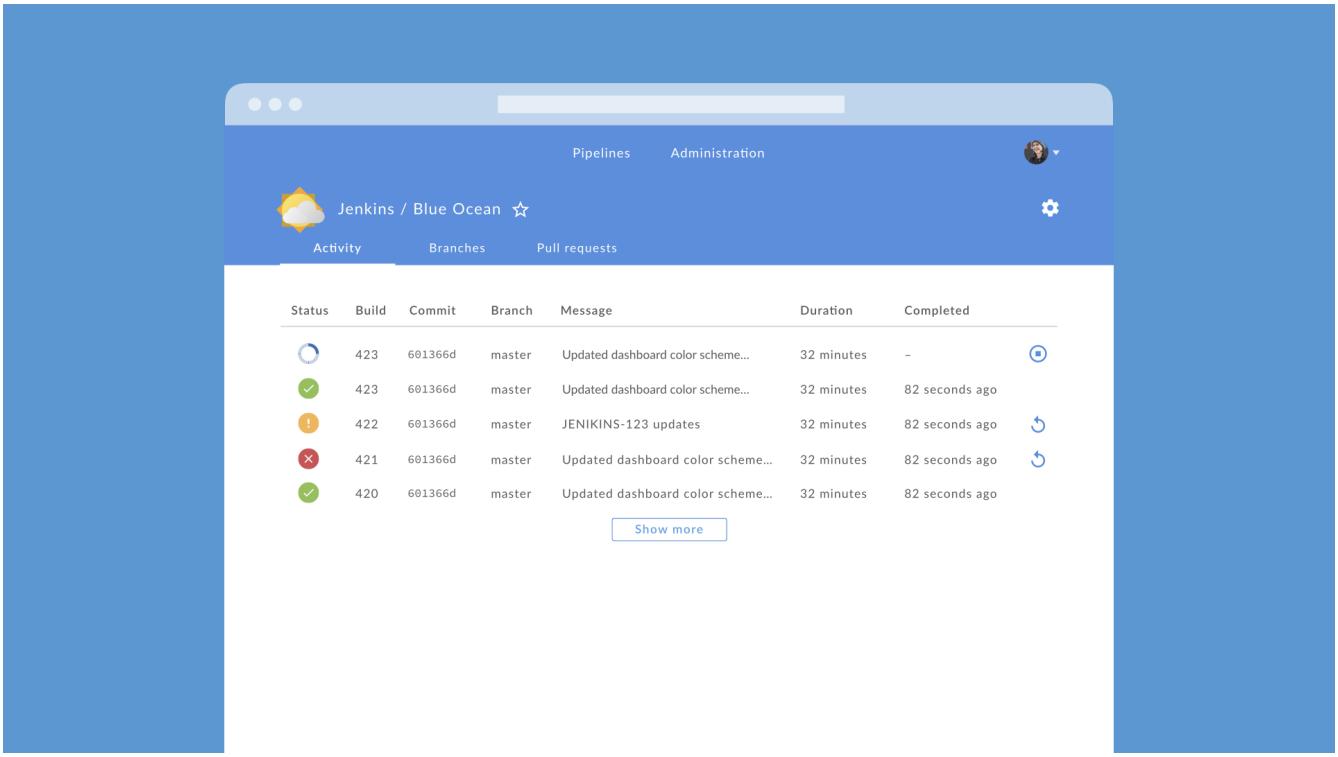
If you are not yet familiar with basic Jenkins terminology and features, start with Getting Started with Jenkins.

What is Blue Ocean?

Blue Ocean is a new project that rethinks the user experience of Jenkins. Designed from the ground up for Jenkins Pipeline and compatible with Freestyle jobs, Blue Ocean reduces clutter and increases clarity for every member of your team through the following key features:

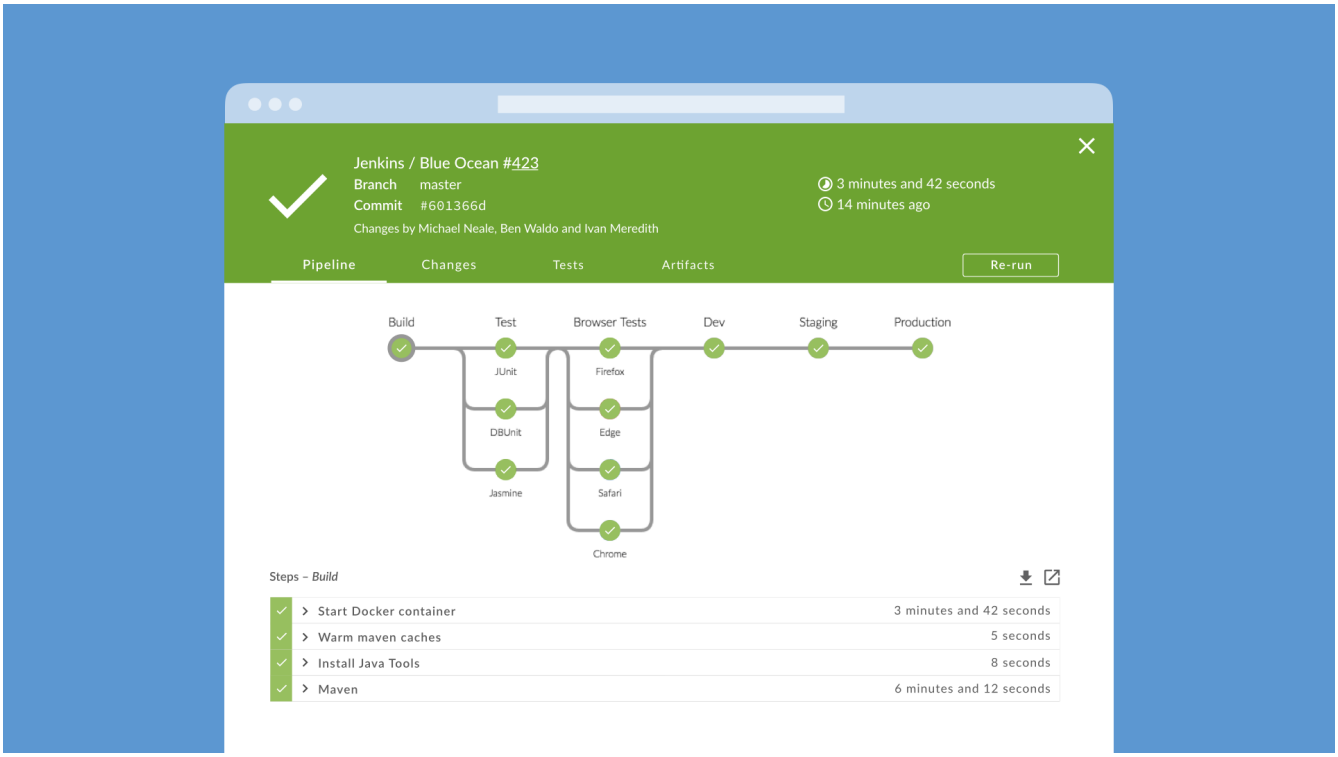
- **Sophisticated visualizations** of CD pipelines, allowing for fast and intuitive comprehension of software pipeline status.
- **Pipeline editor** (In Development) that makes automating CD pipelines approachable by guiding the user through an intuitive and visual process to create a pipeline.
- **Personalization** of the Jenkins UI to suit the role-based needs of each member of the DevOps team.
- **Pinpoint precision** when intervention is needed and/or issues arise. The Blue Ocean UI shows where in the pipeline attention is needed, facilitating exception handling and increasing productivity.
- **Native integration for branch and pull requests** enables maximum developer productivity when collaborating on code with others in GitHub and Bitbucket.

Here is a quick tour of Blue Ocean.



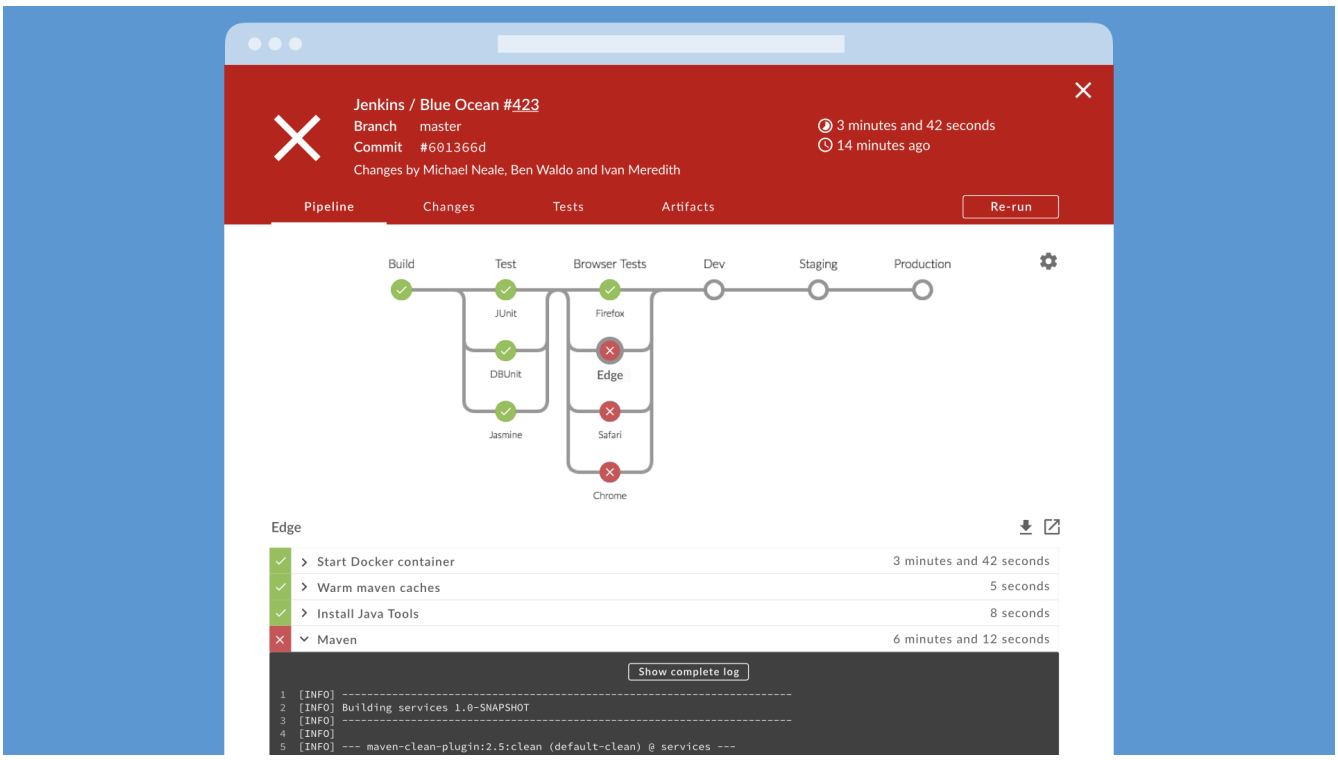
Pipelines

Blue Ocean is built from the ground up for Pipeline and uses its related concepts as a foundation to enable use cases that Jenkins hasn't handled before. The visualization makes it easy to follow along the flow of your Pipeline and gives you clarity into your Continuous Delivery processes.



We know many developers spend a lot of time staring at logs in Jenkins. That's why we've broken up the log broken up step by step to make it much simpler to navigate. Gone are the days that we all spend scrolling through log files that are 10s of thousands or 100s of thousands of lines long.

When you open the pipeline visualisation you can immediately identify what problem that are causing the Pipeline to fail.



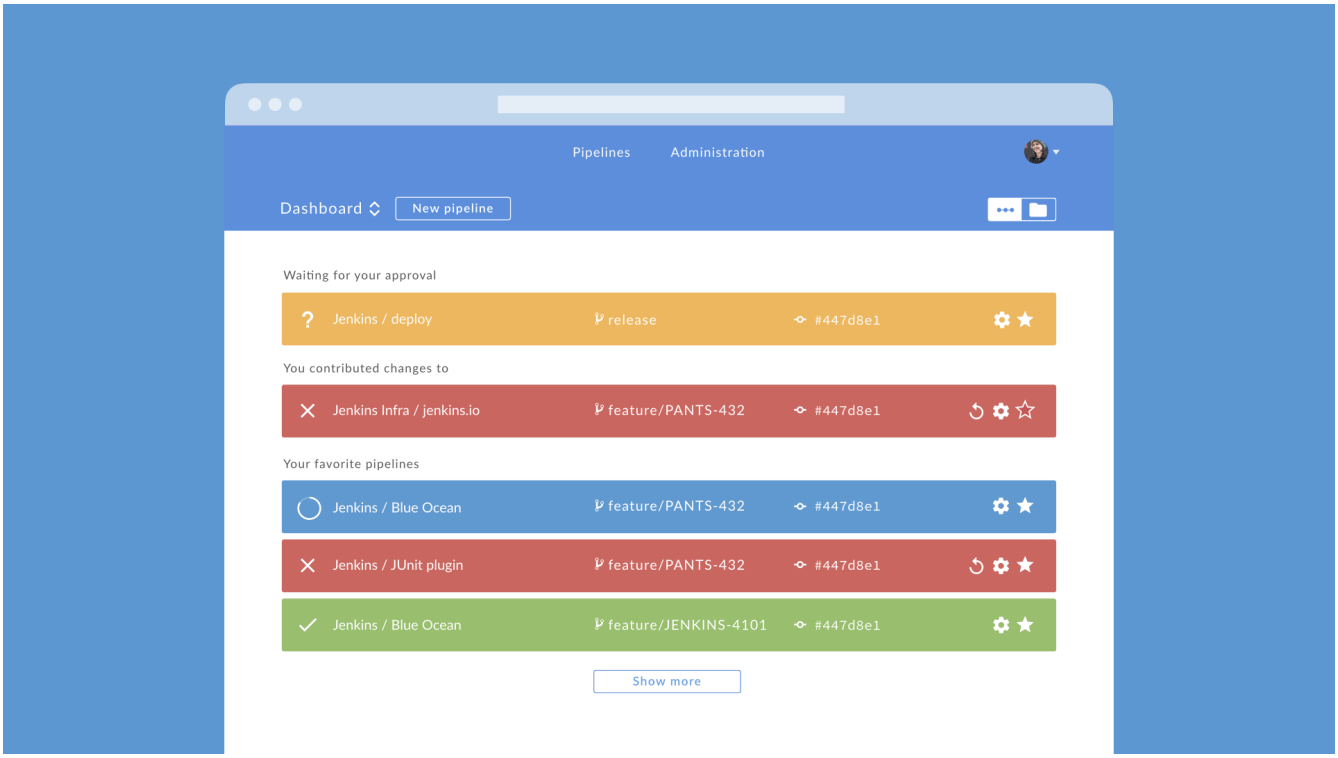
Personalized Dashboard

The Personalized Dashboard surfaces a single developer the things that they care about so that they

can work better with their team. It responds preemptively to your activity showing you the status of Pipelines that you are interested in.

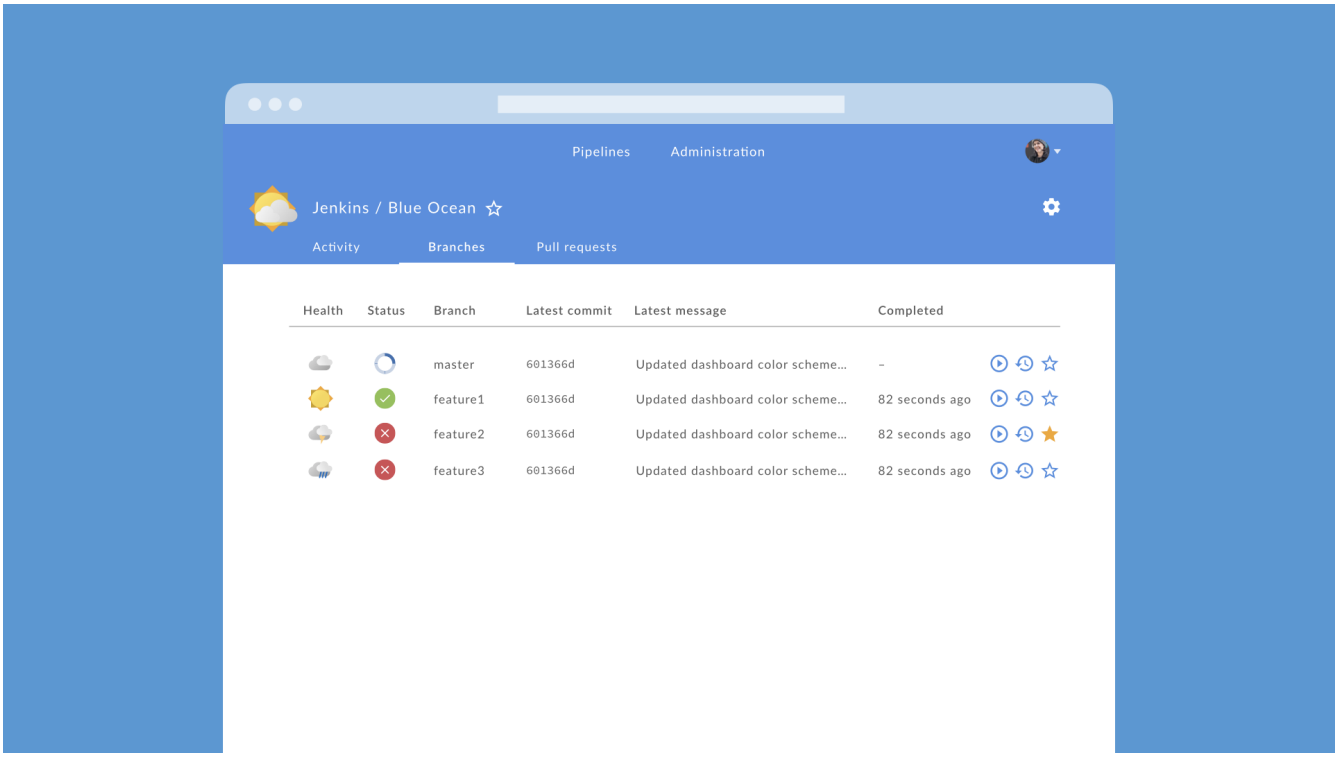
If a developer favorites a Pipeline or one of its branches it will appear on the dashboard. Pipelines needed the developers attention, such as failing Pipelines, appear at the top of the dashboard.

And when the developer creates a branch in Git, Blue Ocean will automatically favourite it for them. When the branch is deleted the Pipeline's branch is removed from view.

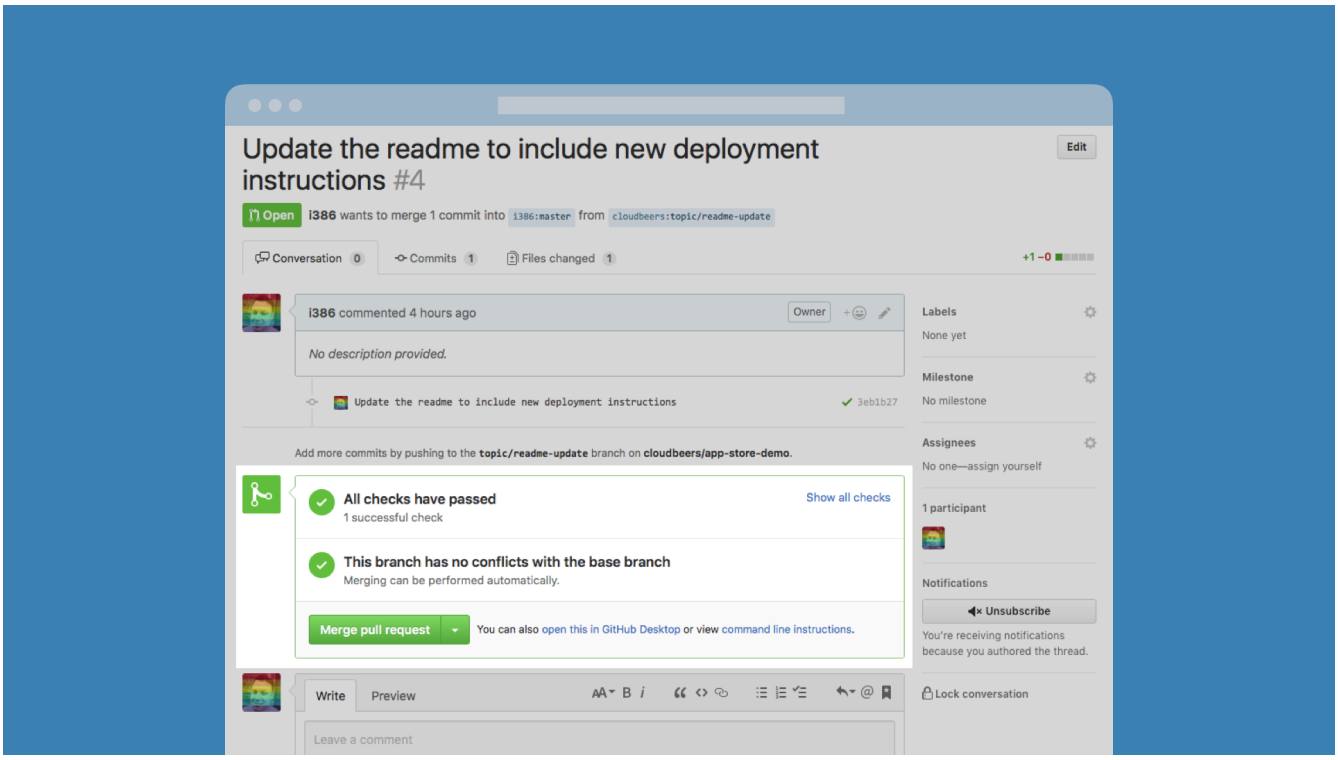


Branches and Pull Requests

Blue Ocean uses Jenkins Pipeline to discover new branches with `Jenkinsfile`s and runs them for you so that you are testing your Pipeline with every new feature branch that you create.



If a developer opens a Pull Request against your repository, Jenkins will build it using your Pipeline and update Github or Bitbucket with the status of the Pipeline. This is extremely useful for reviewers of Pull Requests as successful Pipelines improve confidence that changes will integrate without causing havoc.

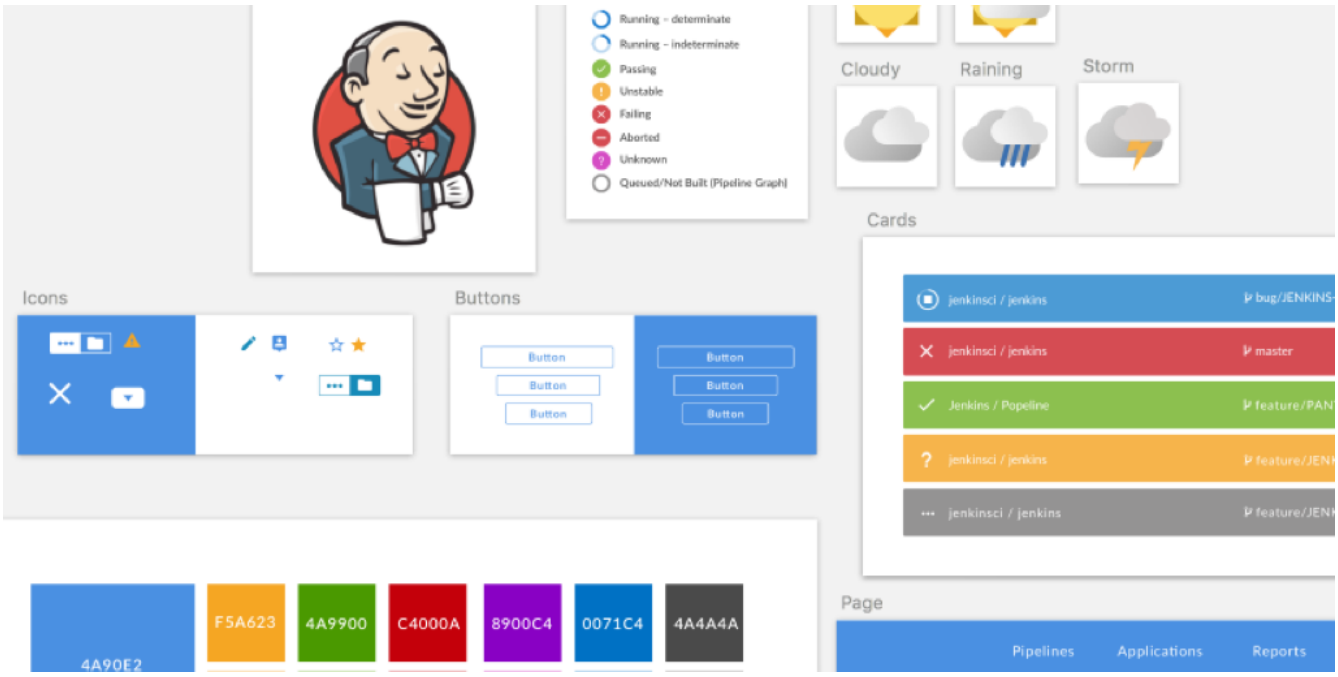


Design Language

Jenkins is nothing without its ecosystem of 1000+ plugins and we believe it's important that Blue Ocean brings them along for the ride. Most developers in our community don't consider themselves hot shot frontend developers or designers, that's why we are introducing a consistent design

language and accompanying JavaScript and CSS framework. If you're familiar with Google's Material Design, Twitter Bootstrap, Apple's Human Interface Guidelines or Microsoft Modern UI you will be right at home here.

For those curious about the technology stack here, we are working with a combination of ES6 and React which we believe will be provide a smooth learning experience for those strong in Java skills but not so confident in CSS/JS. React's component based way of describing user interfaces also complements the way that Jenkins Extensibility works. We've extended the plugin system and toolchain to work with React components.



Limitations

Freestyle jobs

Blue Ocean aims to deliver a great experience around Pipeline and be compatible with any Freestyle jobs that you have configured in your system. However, they won't be able to benefit from any of the features built for Pipelines – for example, Pipeline visualization.

As Blue Ocean is designed to be extensible it will be possible for the community to extend it for other job types in the future.

FAQ

Why does Blue Ocean exist?

The world has moved on from developer tools that are purely functional to developer tools being part of a "developer experience". That is to say, it's no longer about a single tool but the many tools developers use throughout the day and how they work together to achieve a workflow that's beneficial for the developer - this is Developer Experience.

Developer tools companies like Heroku, Atlassian and Github have raised the bar for what is considered good developer experience, and developers are increasingly expecting exceptional design. In recent years developers are becoming more rapidly attracted to tools that are not only functional but are designed to fit into their workflow seamlessly and are a joy to use. This shift represents a higher standard of design and user experience that Jenkins needs to rise to meet.

Creating and visualising continuous delivery pipelines is something valuable for many Jenkins users and this is demonstrated in the 5+ plugins that the community has created to meet their needs. To us this indicates a need to revisit how Jenkins currently expresses these concepts and consider delivery pipelines as a central theme to the Jenkins user experience.

It's not just continuous delivery concepts but the tools that developers use every day – Github, Bitbucket, Slack, HipChat, Puppet or Docker. It's about more than Jenkins – it's the developer workflow that surrounds Jenkins that spans multiple tools.

New teams have little time for learning to assemble their own Jenkins experience – they want to improve their time to market by shipping better software faster. Assembling that ideal Jenkins experience is something we can work together as a community of Jenkins users and contributors to define. As time progresses, developers' expectations of good user experience will change and the mission of Blue Ocean will enable the Jenkins project to respond.

The Jenkins community has poured its sweat and tears into building the most technically capable and extensible software automation tool in existence. Not doing anything to revolutionize the Jenkins developer experience today is just inviting someone else – in closed source – to do it.

Where is the name from?

The name Blue Ocean comes from the book Blue Ocean Strategy where instead of looking at strategic problems within a contested space you look at problems in the larger uncontested space. To put this more simply, consider this quote from ice hockey legend Wayne Gretzky: "skate to where the puck is going to be, not where it has been".

Where can I find the source code?

The source code can be found on Github:

- Blue Ocean
- Jenkins Design Language

What does this mean for the classic Jenkins UI?

The intention is that as Blue Ocean matures there will be less and less reasons for users to go back to the existing UI.

For example, in the first version we will mainly be targeting Pipeline jobs. You might be able to see your existing non-pipeline jobs in Blue Ocean but it might not be possible to configure them from the new UI for some time. This means users will have to jump back to the classic UI for configuration of non-pipeline jobs.

There are likely going to be more examples of this and that's why the classic UI will still be important in the long term.

Is this a CloudBees project?

The short answer is **"no"**. The project has been originated and sponsored by CloudBees, but it is a **100% open project** (including sources, roadmaps, public discussions, etc.). Everybody is invited to contribute to it.

To quote James Dumay (Blue Ocean Product Manager at CloudBees):

> While the project's inception has happened within CloudBees we see this project being one owned by the community. At CloudBees we recognize the importance of a vibrant and healthy Jenkins community, we see the company and community working in symbiosis: a thriving developer community is good for CloudBees and CloudBees provides time and money back into the community to make it stronger. Blue Ocean is our way of giving back and strengthening. To that effect we've put together a new a team of product, UX, frontend and backend developers (some old faces and a lot of new ones!) that will be working on this project with the community full time

What does this mean for my plugins?

Extensibility is a pretty core concept to Jenkins, so being able to extend the Blue Ocean UI is important. Based on some research, we worked out a way to allow "<ExtensionPoint name=..>" to be used in the markup of Blue Ocean, leaving places for plugins to contribute to the UI (plugins can have their own Blue Ocean extension points, just like they do today in Jenkins). Blue Ocean itself (as it is so far) is implemented using these extension points. Extensions are delivered by plugins, as normal, only if they wish to contribute to the Blue Ocean experience they will have some additional javascript that provides extensions.

What technologies are currently in use?

Blue Ocean is built as a collection of Jenkins plugins itself. There is one key difference, however. It

provides both its own endpoint for http requests and delivers up html/javascript via a different path, without the existing Jenkins UI markup/scripts. React.js and ES6 are used to deliver the javascript components of Blue Ocean. Inspired by this excellent open source project (react-plugins) an <ExtensionPoint>pattern was established, that allows extensions to come from any Jenkins plugin (only with Javascript) and should they fail to load, have failures isolated.

Getting Started with Blue Ocean

Jenkins Blue Ocean is a suite of plugins which provide a new an awesome way to live your life.

Prerequisites

To use Jenkins Blue Ocean, you will need:

- Jenkins 2.7.x or later

Installing

To start using theBlue Ocean plugin [23: Blue Ocean plugin] from an existing Jenkins:

1. Login to your Jenkins server
2. Click `Manage Jenkins` in the sidebar then `Manage Plugins`
3. Choose the `Available` tab and use the search bar to find `Blue Ocean`
4. Click the checkbox in the Install column
5. Click either `Install without restart` or `Download now and install after restart`
6. When installation is complete click the `Use Blue Ocean` button in the classic UI

For in-depth description on how to install and manage plugins, consult Managing Plugins.

Configuring

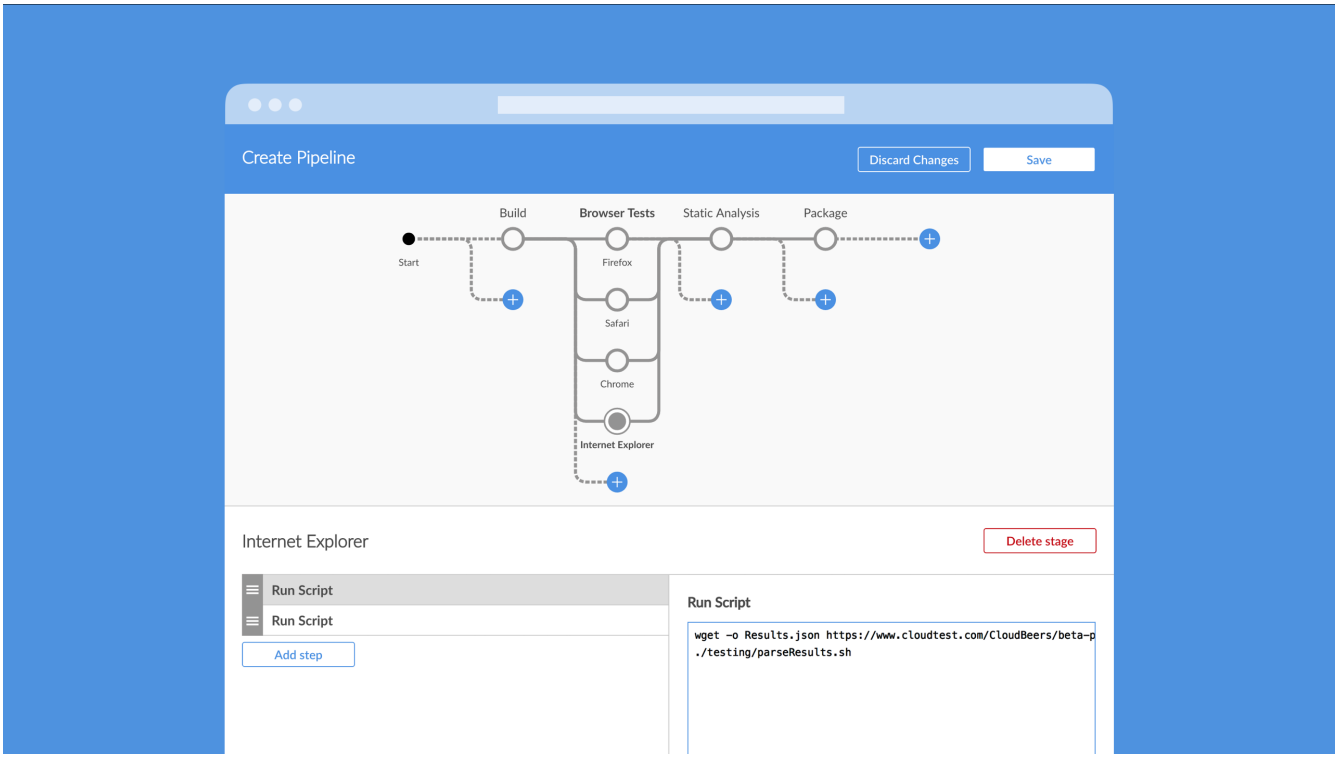
Switch to and from Blue Ocean

Pipeline Editor

The Blue Ocean Pipeline Editor is the simplest way for anyone wanting to get started with creating Pipelines in Jenkins. It's also great way for advanced Jenkins users to start adopting Pipeline.

It allows developers to break up their Pipeline into different stages and parallelize tasks that can occur at the same time - its all up to you. When you are done authoring your Pipeline, the pipeline definition is saved back to your repository as a `Jenkinsfile`.

If you ever need to change the Pipeline again, the user interfaces allows you to jump back in into the editor to author the Pipeline at any time or you can do so in code by editing the `Jenkinsfile` directly.



Jenkins Use-Cases

An alternate title for this chapter would be "Jenkins for X". Each section of this chapter focuses on how to 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 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 programatically 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 able to reconfigure Jenkins.

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

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

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

Managing Jenkins with Chef

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 plugin:azure-vm-agents[Azure VM Agents] or plugin:ec2[Amazon EC2] plugins.

Core

The primary Jenkins application (`jenkins.war`) which provides the basic web UI, configuration, and foundation upon which Plugins 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.