Jenkins User Handbook

jenkinsci-docs@googlegroups.com

Table of Contents

User Handbook overview

This page provides an overview of the documentation in the Jenkins User Handbook.

If you want to get up and running with Jenkins, see Installing Jenkins for procedures on how to install Jenkins on your supported platform of choice.

If you are a typical Jenkins user (of any skill level) who wants to know more about Jenkins usage, see Using Jenkins. Also refer to the separate Pipeline and Blue Ocean chapters for more information about these core Jenkins features.

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 Jenkins System Administration.

Installing Jenkins

The procedures on this page are for new installations of Jenkins on a single/local machine.

Jenkins is typically run as a standalone application in its own process with the built-in Java servlet container/application server (Jetty).

Jenkins can also be run as a servlet in different Java servlet containers such as Apache Tomcat or GlassFish. However, instructions for setting up these types of installations are beyond the scope of this page.

Note: Although this page focuses on local installations of Jenkins, this content can also be used to help set up Jenkins in production environments.

Prerequisites

Minimum hardware requirements:

- 256 MB of RAM
- 1 GB of drive space (although 10 GB is a recommended minimum if running Jenkins as a Docker container)

Recommended hardware configuration for a small team:

- 1 GB+ of RAM
- 50 GB+ of drive space

Software requirements:

- Java: see the Java Requirements page
- Web browser: see the Web Browser Compatibility page

Installation platforms

This section describes how to install/run Jenkins on different platforms and operating systems.

Docker

Docker is a platform for running applications in an isolated environment called a "container" (or Docker container). Applications like Jenkins can be downloaded as read-only "images" (or Docker images), each of which is run in Docker as a container. A Docker container is in effect a "running instance" of a Docker image. From this perspective, an image is stored permanently more or less (i.e. insofar as image updates are published), whereas containers are stored temporarily. Read more about these concepts in the Docker documentation’s Getting Started, Part 1: Orientation and setup page.

Docker’s fundamental platform and container design means that a single Docker image (for any given application like Jenkins) can be run on any supported operating system (macOS, Linux and

Windows) or cloud service (AWS and Azure) which is also running Docker.

Installing Docker

To install Docker on your operating system, visit the Docker store website and click the **Docker Community Edition** box which is suitable for your operating system or cloud service. Follow the installation instructions on their website.

Jenkins can also run on Docker Enterprise Edition, which you can access through **Docker EE** on the Docker store website.

CAUTION If you are installing Docker on a Linux-based operating system, ensure you configure Docker so it can be managed as a non-root user. Read more about this in Docker's Post-installation steps for Linux page of their documentation. This page also contains information about how to configure Docker to start on boot.

Downloading and running Jenkins in Docker

There are several Docker images of Jenkins available.

The recommended Docker image to use is the `jenkinsci/blueocean` image (from the Docker Hub repository). This image contains the current Long-Term Support (LTS) release of Jenkins (which is production-ready) bundled with all Blue Ocean plugins and features. This means that you do not need to install the Blue Ocean plugins separately.

NOTE A new `jenkinsci/blueocean` image is published each time a new release of Blue Ocean is published. You can see a list of previously published versions of the `jenkinsci/blueocean` image on the tags page.

There are also other Jenkins Docker images you can use (accessible through `jenkins/jenkins` on Docker Hub). However, these do not come with Blue Ocean, which would need to be installed via the **Manage Jenkins** > **Manage Plugins** page in Jenkins. Read more about this in Getting started with Blue Ocean.

On macOS and Linux

1. Open up a terminal window.
2. Download the `jenkinsci/blueocean` image and run it as a container in Docker using the following `docker run` command:

```
docker run \
  -u root \
  --rm \  ①
  -d \ ②
  -p 8080:8080 \ ③
  -p 50000:50000 \ ④
  -v jenkins-data:/var/jenkins_home \ ⑤
  -v /var/run/docker.sock:/var/run/docker.sock \ ⑥
  jenkinsci/blueocean ⑦
```

① (*Optional*) Automatically removes the Docker container (which is the instantiation of the `jenkinsci/blueocean` image below) when it is shut down. This keeps things tidy if you need to quit Jenkins.

② (*Optional*) Runs the `jenkinsci/blueocean` container in the background (i.e. "detached" mode) and outputs the container ID. If you do not specify this option, then the running Docker log for this container is output in the terminal window.

③ Maps (i.e. "publishes") port 8080 of the `jenkinsci/blueocean` container to port 8080 on the host machine. The first number represents the port on the host while the last represents the container's port. Therefore, if you specified `-p 49000:8080` for this option, you would be accessing Jenkins on your host machine through port 49000.

④ (*Optional*) Maps port 50000 of the `jenkinsci/blueocean` container to port 50000 on the host machine. This is only necessary if you have set up one or more JNLP-based Jenkins agents on other machines, which in turn interact with the `jenkinsci/blueocean` container (acting as the "master" Jenkins server, or simply "Jenkins master"). JNLP-based Jenkins agents communicate with the Jenkins master through TCP port 50000 by default. You can change this port number on your Jenkins master through the Configure Global Security page. If you were to change your Jenkins master's **TCP port for JNLP agents** value to 51000 (for example), then you would need to re-run Jenkins (via this `docker run …` command) and specify this "publish" option with something like `-p 52000:51000`, where the last value matches this changed value on the Jenkins master and the first value is the port number on the Jenkins master's host machine through which the JNLP-based Jenkins agents communicate (to the Jenkins master) - i.e. 52000.

⑤ (*Optional but highly recommended*) Maps the `/var/jenkins_home` directory in the container to the Docker volume with the name `jenkins-data`. If this volume does not exist, then this `docker run` command will automatically create the volume for you. This option is required if you want your Jenkins state to persist each time you restart Jenkins (via this `docker run …` command). If you do not specify this option, then Jenkins will effectively reset to a new instance after each restart.
Notes: The `jenkins-data` volume could also be created independently using the `docker volume create` command:
`docker volume create jenkins-data`
Instead of mapping the `/var/jenkins_home` directory to a Docker volume, you could also map this directory to one on your machine's local file system. For example, specifying the option `-v $HOME/jenkins:/var/jenkins_home` would map the container's `/var/jenkins_home` directory to the `jenkins` subdirectory within the `$HOME` directory on your local machine, which would typically be `/Users/<your-username>/jenkins` or `/home/<your-username>/jenkins`.

⑥ (*Optional*) `/var/run/docker.sock` represents the Unix-based socket through which the Docker daemon listens on. This mapping allows the `jenkinsci/blueocean` container to communicate with the Docker daemon, which is required if the `jenkinsci/blueocean` container needs to instantiate other Docker containers. This option is necessary if you run declarative Pipelines whose syntax contains the `agent` section with the `docker` parameter - i.e. `agent { docker { … } }`. Read more about this on the Pipeline Syntax page.

⑦ The `jenkinsci/blueocean` Docker image itself. If this image has not already been downloaded, then this `docker run` command will automtically download the image for you. Furthermore, if any updates to this image were published since you last ran this command, then running this command again will automatically download these published image updates for you. **Note:** This Docker image could also be downloaded (or updated) independently using the `docker pull` command:
`docker pull jenkinsci/blueocean`

Note: If copying and pasting the command snippet above does not work, try copying and pasting this annotation-free version here:

```
docker run \
  -u root \
  --rm \
  -d \
  -p 8080:8080 \
  -p 50000:50000 \
  -v jenkins-data:/var/jenkins_home \
  -v /var/run/docker.sock:/var/run/docker.sock \
  jenkinsci/blueocean
```

3. Proceed to the Post-installation setup wizard.

On Windows

1. Open up a command prompt window.
2. Download the `jenkinsci/blueocean` image and run it as a container in Docker using the following `docker run` command:

```
docker run ^
  -u root ^
  --rm ^
  -d ^
  -p 8080:8080 ^
  -p 50000:50000 ^
  -v jenkins-data:/var/jenkins_home ^
  -v /var/run/docker.sock:/var/run/docker.sock ^
  jenkinsci/blueocean
```

For an explanation of each of these options, refer to the macOS and Linux instructions above.

3. Proceed to the Post-installation setup wizard.

Accessing the Jenkins/Blue Ocean Docker container

If you have some experience with Docker and you wish or need to access the `jenkinsci/blueocean` container through a terminal/command prompt using the `docker exec` command, you can add an option like `--name jenkins-blueocean` (with the `docker run` above), which would give the `jenkinsci/blueocean` container the name "jenkins-blueocean".

This means you could access the container (through a separate terminal/command prompt window) with a `docker exec` command like:

```
docker exec -it jenkins-blueocean bash
```

Accessing the Jenkins console log through Docker logs

There is a possibility you may need to access the Jenkins console log, for instance, when Unlocking Jenkins as part of the Post-installation setup wizard.

If you did not specify the detached mode option `-d` with the `docker run …` command above, then the Jenkins console log is easily accessible through the terminal/command prompt window from which you ran this Docker command.

Otherwise, you can access the Jenkins console log through the Docker logs of the `jenkinsci/blueocean` container using the following command:

```
docker logs <docker-container-name>
```

Your `<docker-container-name>` can be obtained using the `docker ps` command. If you specified the `--name jenkins-blueocean` option in the `docker run …` command above (see also Accessing the Jenkins/Blue Ocean Docker container), you can simply use the `docker logs` command:

```
docker logs jenkins-blueocean
```

Accessing the Jenkins home directory

There is a possibility you may need to access the Jenkins home directory, for instance, to check the details of a Jenkins build in the `workspace` subdirectory.

If you mapped the Jenkins home directory (`/var/jenkins_home`) to one on your machine's local file system (i.e. in the `docker run …` command above), then you can access the contents of this directory through your machine's usual terminal/command prompt.

Otherwise, if you specified the `-v jenkins-data:/var/jenkins_home` option in the `docker run …` command, you can access the contents of the Jenkins home directory through the `jenkinsci/blueocean` container's terminal/command prompt using the `docker exec` command:

```
docker exec -it <docker-container-name> bash
```

As mentioned above, your `<docker-container-name>` can be obtained using the `docker ps` command.
If you specified the
`--name jenkins-blueocean` option in the `docker run …` command above (see also Accessing the

Jenkins/Blue Ocean Docker container), you can simply use the `docker exec` command:

`docker exec -it jenkins-blueocean bash`

WAR file

The Web application ARchive (WAR) file version of Jenkins can be installed on any operating system or platform that supports Java.

To download and run the WAR file version of Jenkins:

1. Download the latest stable Jenkins WAR file to an appropriate directory on your machine.
2. Open up a terminal/command prompt window to the download directory.
3. Run the command `java -jar jenkins.war`.
4. Browse to `http://localhost:8080` and wait until the **Unlock Jenkins** page appears.
5. Continue on with the Post-installation setup wizard below.

Notes:

- Unlike downloading and running Jenkins with Blue Ocean in Docker (above), this process does not automatically install the Blue Ocean features, which would need to installed separately via the **Manage Jenkins** > **Manage Plugins** page in Jenkins. Read more about the specifics for installing Blue Ocean on the Getting started with Blue Ocean page.
- You can change the port by specifying the `--httpPort` option when you run the `java -jar jenkins.war` command. For example, to make Jenkins accessible through port 9090, then run Jenkins using the command:
  `java -jar jenkins.war --httpPort=9090`

macOS

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
```

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.

Fedora

You can install Jenkins through `dnf`. You need to add the Jenkins repository from the Jenkins website to the package manager first.

```
sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-
ci.org/redhat/jenkins.repo
sudo rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key
```

Then, you can install Jenkins. The following command also ensures you have java installed.

```
sudo dnf upgrade && sudo dnf install jenkins java
```

Next, start the Jenkins service.

```
sudo service jenkins start
sudo chkconfig jenkins on
```

You can check the status of the Jenkins service using this systemctl command:

```
systemctl status jenkins
```

If everything has been set up correctly, you should see an output like this:

```
Loaded: loaded (/etc/rc.d/init.d/jenkins; generated)
Active: active (running) since Tue 2018-11-13 16:19:01 +03; 4min 57s ago
...
```

NOTE

If you have a firewall installed, you must add Jenkins as an exception. You must change `YOURPORT` in the script below to the port you want to use. Port `8080` is the most common.

```
firewall-cmd --permanent --new-service=jenkins
firewall-cmd --permanent --service=jenkins --set-short="Jenkins Service
Ports"
firewall-cmd --permanent --service=jenkins --set-description="Jenkins
service firewalld port exceptions"
firewall-cmd --permanent --service=jenkins --add-port=YOURPORT/tcp
firewall-cmd --permanent --add-service=jenkins
firewall-cmd --zone=public --add-service=http --permanent
firewall-cmd --reload
```

Windows

To install from the website, using the installer:

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

Other operating systems

OpenIndiana Hipster

On a system running OpenIndiana Hipster Jenkins can be installed in either the local or global zone using the Image Packaging System (IPS).

IMPORTANT Disclaimer: This platform is NOT officially supported by the Jenkins team, use it at your own risk. Packaging and integration described in this section is maintained by the OpenIndiana Hipster team, bundling the generic `jenkins.war` to work in that operating environment.

For the common case of running the newest packaged weekly build as a standalone (Jetty) server, simply execute:

```
pkg install jenkins
svcadm enable jenkins
```

The common packaging integration for a standalone service will:

- Create a `jenkins` user to run the service and to own the directory structures under `/var/lib/jenkins`.
- Pull the OpenJDK8 and other packages required to execute Jenkins, including the `jenkins-core-weekly` package with the latest `jenkins.war`.

CAUTION Long-Term Support (LTS) Jenkins releases currently do not support OpenZFS-based systems, so no packaging is provided at this time.

- Set up Jenkins as an SMF service instance (`svc:/network/http:jenkins`) which can then be enabled with the `svcadm` command demonstrated above.
- Set up Jenkins to listen on port 8080.
- Configure the log output to be managed by SMF at `/var/svc/log/network-http:jenkins.log`.

Once Jenkins is running, consult the log (`/var/svc/log/network-http:jenkins.log`) to retrieve the generated administrator password for the initial set up of Jenkins, usually it will be found at `/var/lib/jenkins/home/secrets/initialAdminPassword`. Then navigate to localhost:8080 to complete configuration of the Jenkins instance.

To change attributes of the service, such as environment variables like `JENKINS_HOME` or the port number used for the Jetty web server, use the `svccfg` utility:

```
svccfg -s svc:/network/http:jenkins editprop
svcadm refresh svc:/network/http:jenkins
```

You can also refer to `/lib/svc/manifest/network/jenkins-standalone.xml` for more details and comments about currently supported tunables of the SMF service. Note that the `jenkins` user account created by the packaging is specially privileged to allow binding to port numbers under 1024.

The current status of Jenkins-related packages available for the given release of OpenIndiana can be queried with:

```
pkg info -r '*jenkins*'
```

Upgrades to the package can be performed by updating the entire operating environment with `pkg update`, or specifically for Jenkins core software with:

```
pkg update jenkins-core-weekly
```

CAUTION Procedure for updating the package will restart the currently running Jenkins process. Make sure to prepare it for shutdown and finish all running jobs before updating, if needed.

Solaris, OmniOS, SmartOS, and other siblings

Generally it should suffice to install Java 8 and download the `jenkins.war` and run it as a standalone process or under an application server such as Apache Tomcat.

Some caveats apply:

- Headless JVM and fonts: For OpenJDK builds on minimalized-footprint systems, there may be issues running the headless JVM, because Jenkins needs some fonts to render certain pages.
- ZFS-related JVM crashes: When Jenkins runs on a system detected as a `SunOS`, it tries to load integration for advanced ZFS features using the bundled `libzfs.jar` which maps calls from Java to native `libzfs.so` routines provided by the host OS. Unfortunately, that library was made for binary utilities built and bundled by the OS along with it at the same time, and was never intended as a stable interface exposed to consumers. As the forks of Solaris legacy, including ZFS and later the OpenZFS initiative evolved, many different binary function signatures were provided by different host operating systems - and when Jenkins `libzfs.jar` invoked the wrong signature, the whole JVM process crashed. A solution was proposed and integrated in `jenkins.war` since weekly release 2.55 (and not yet in any LTS to date) which enables the administrator to configure which function signatures should be used for each function known to have different variants, apply it to their application server initialization options and then run and update the generic `jenkins.war` without further workarounds. See the libzfs4j Git repository for more details, including a script to try and "lock-pick" the configuration needed for your particular distribution (in particular if your kernel updates bring a new incompatible `libzfs.so`).

Also note that forks of the OpenZFS initiative may provide ZFS on various BSD, Linux, and macOS distributions. Once Jenkins supports detecting ZFS capabilities, rather than relying on the `SunOS` check, the above caveats for ZFS integration with Jenkins should be considered.

Post-installation setup wizard

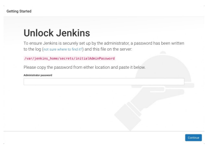
After downloading, installing and running Jenkins using one of the procedures above, the post-installation setup wizard begins.

This setup wizard takes you through are a few quick "one-off" steps to unlock Jenkins, customize it with plugins and create the first administrator user through which you can continue accessing Jenkins.

Unlocking Jenkins

When you first access a new Jenkins instance, you are asked to unlock it using an automatically-generated password.

1. Browse to `http://localhost:8080` (or whichever port you configured for Jenkins when installing it) and wait until the **Unlock Jenkins** page appears.



Getting Started

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

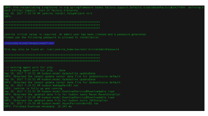
`/var/jenkins_home/secrets/initialAdminPassword`

Please copy the password from either location and paste it below.

Administrator password

Continue

2. From the Jenkins console log output, copy the automatically-generated alphanumeric password (between the 2 sets of asterisks).



```
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@24cf7404: defining b
eans [filter,legacy]; root of factory hierarchy
Sep 30, 2017 7:18:39 AM jenkins.install.SetupWizard init
INFO:

*************************************************************
*************************************************************
*************************************************************

Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

[illegible]

This may also be found at: /var/jenkins_home/secrets/initialAdminPassword

*************************************************************
*************************************************************
*************************************************************

--> setting agent port for jnlp
--> setting agent port for jnlp... done
Sep 30, 2017 7:18:51 AM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Sep 30, 2017 7:18:52 AM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Sep 30, 2017 7:18:52 AM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
Sep 30, 2017 7:18:52 AM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
Sep 30, 2017 7:18:58 AM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tools.JDKInstaller
Sep 30, 2017 7:18:59 AM hudson.model.AsyncPeriodicWork$1 run
INFO: Finished Download metadata. 25,543 ms
```

3. On the **Unlock Jenkins** page, paste this password into the **Administrator password** field and click **Continue**.
   Notes:
   - If you ran Jenkins in Docker in detached mode, you can access the Jenkins console log from the Docker logs (above).
   - The Jenkins console log indicates the location (in the Jenkins home directory) where this password can also be obtained. This password must be entered in the setup wizard on new Jenkins installations before you can access Jenkins's main UI. This password also serves as the default admininstrator account's password (with username "admin") if you happen to skip the subsequent user-creation step in the setup wizard.

Customizing Jenkins with plugins

After unlocking Jenkins, the **Customize Jenkins** page appears. Here you can install any number of useful plugins as part of your initial setup.

Click one of the two options shown:

- **Install suggested plugins** - to install the recommended set of plugins, which are based on most common use cases.
- **Select plugins to install** - to choose which set of plugins to initially install. When you first access the plugin selection page, the suggested plugins are selected by default.

NOTE	If you are not sure what plugins you need, choose **Install suggested plugins**. You can install (or remove) additional Jenkins plugins at a later point in time via the **Manage Jenkins** > **Manage Plugins** page in Jenkins.

The setup wizard shows the progression of Jenkins being configured and your chosen set of Jenkins plugins being installed. This process may take a few minutes.

Creating the first administrator user

Finally, after customizing Jenkins with plugins, Jenkins asks you to create your first administrator user.

1. When the **Create First Admin User** page appears, specify the details for your administrator user in the respective fields and click **Save and Finish**.
2. When the **Jenkins is ready** page appears, click **Start using Jenkins**.
   Notes:
   - This page may indicate **Jenkins is almost ready!** instead and if so, click **Restart**.
   - If the page does not automatically refresh after a minute, use your web browser to refresh the page manually.
3. If required, log in to Jenkins with the credentials of the user you just created and you are ready to start using Jenkins!

IMPORTANT From this point on, the Jenkins UI is only accessible by providing valid username and password credentials.

Using Jenkins

This chapter contains topics for typical Jenkins users (of all skill levels) about Jenkins usage which is outside the scope of the core Jenkins features: Pipeline and Blue Ocean.

If you want to create and configure a Pipeline project through a `Jenkinsfile` or through Blue Ocean, or you wish to find out more about these core Jenkins features, refer to the relevant topics within the respective Pipeline and Blue Ocean chapters.

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 Jenkins System Administration.

For an overview of content in the Jenkins User Handbook, see User Handbook overview.

Using credentials

There are numerous 3rd-party sites and applications that can interact with Jenkins, for example, artifact repositories, cloud-based storage systems and services, and so on.

A systems administrator of such an application can configure credentials in the application for dedicated use by Jenkins. This would typically be done to "lock down" areas of the application's functionality available to Jenkins, usually by applying access controls to these credentials. Once a Jenkins manager (i.e. a Jenkins user who administers a Jenkins site) adds/configures these credentials in Jenkins, the credentials can be used by Pipeline projects to interact with these 3rd party applications.

Note: The Jenkins credentials functionality described on this and related pages is provided by the plugin:credentials-binding[Credentials Binding plugin].

Credentials stored in Jenkins can be used:

- anywhere applicable throughout Jenkins (i.e. global credentials),
- by a specific Pipeline project/item (read more about this in the Handling credentials section of Using a Jenkinsfile),
- by a specific Jenkins user (as is the case for Pipeline projects created in Blue Ocean).

Jenkins can store the following types of credentials:

- **Secret text** - a token such as an API token (e.g. a GitHub personal access token),
- **Username and password** - which could be handled as separate components or as a colon separated string in the format `username:password` (read more about this in Handling credentials),
- **Secret file** - which is essentially secret content in a file,
- **SSH Username with private key** - an SSH public/private key pair,
- **Certificate** - a PKCS#12 certificate file and optional password, or
- **Docker Host Certificate Authentication** credentials.

Credential security

To maximize security, credentials configured in Jenkins are stored in an encrypted form on the master Jenkins instance (encrypted by the Jenkins instance ID) and are only handled in Pipeline projects via their credential IDs.

This minimizes the chances of exposing the actual credentials themselves to Jenkins users and hinders the ability to copy functional credentials from one Jenkins instance to another.

Configuring credentials

This section describes procedures for configuring credentials in Jenkins.

Credentials can be added to Jenkins by any Jenkins user who has the **Credentials** > **Create** permission (set through **Matrix-based security**). These permissions can be configured by a Jenkins user with the **Administer** permission. Read more about this in the Authorization section of Managing Security.

Otherwise, any Jenkins user can add and configure credentials if the **Authorization** settings of your Jenkins instance's **Configure Global Security** settings page is set to the default **Logged-in users can do anything** setting or **Anyone can do anything** setting.

Adding new global credentials

To add new global credentials to your Jenkins instance:

1. If required, ensure you are logged in to Jenkins (as a user with the **Credentials** > **Create** permission).
2. From the Jenkins home page (i.e. the Dashboard of the Jenkins classic UI), click **Credentials** > **System** on the left.
3. Under **System**, click the **Global credentials (unrestricted)** link to access this default domain.
4. Click **Add Credentials** on the left.
   Note: If there are no credentials in this default domain, you could also click the **add some credentials** link (which is the same as clicking the **Add Credentials** link).
5. From the **Kind** field, choose the type of credentials to add.
6. From the **Scope** field, choose either:
   - **Global** - if the credential/s to be added is/are for a Pipeline project/item. Choosing this option applies the scope of the crendential/s to the Pipeline project/item "object" and all its descendent objects.
   - **System** - if the credential/s to be added is/are for the Jenkins instance itself to interact with system administration functions, such as email authentication, agent connection, etc. Choosing this option applies the scope of the crendential/s to a single object only.
7. Add the credentials themselves into the appropriate fields for your chosen credential type:
   - **Secret text** - copy the secret text and paste it into the **Secret** field.
   - **Username and password** - specify the credential's **Username** and **Password** in their respective fields.
   - **Secret file** - click the **Choose file** button next to the **File** field to select the secret file to upload to Jenkins.
   - **SSH Username with private key** - specify the credentials **Username**, **Private Key** and optional **Passphrase** into their respective fields.
     Note: Choosing **Enter directly** allows you to copy the private key's text and paste it into the resulting **Key** text box.
   - **Certificate** - specify the **Certificate** and optional **Password**. Choosing **Upload PKCS#12 certificate** allows you to upload the certificate as a file via the resulting **Upload certificate** button.
   - **Docker Host Certificate Authentication** - copy and paste the appropriate details into the **Client Key**, **Client Certificate** and **Server CA Certificate** fields.

8. In the **ID** field, specify a meaningful credential ID value - for example, `jenkins-user-for-xyz-artifact-repository`. You can use upper- or lower-case letters for the credential ID, as well as any valid separator character. However, for the benefit of all users on your Jenkins instance, it is best to use a single and consistent convention for specifying credential IDs.
   Note: This field is optional. If you do not specify its value, Jenkins assigns a globally unique ID (GUID) value for the credential ID. Bear in mind that once a credential ID is set, it can no longer be changed.
9. Specify an optional **Description** for the credential/s.
10. Click **OK** to save the credentials.

Pipeline

This chapter covers all recommended aspects of Jenkins Pipeline functionality, including how to:

- get started with Pipeline - covers how to define a Jenkins Pipeline (i.e. your `Pipeline`) through Blue Ocean, through the classic UI or in SCM,
- create and use a `Jenkinsfile` - covers use-case scenarios on how to craft and construct your `Jenkinsfile`,
- work with branches and pull requests,
- use Docker with Pipeline - covers how Jenkins can invoke Docker containers on agents/nodes (from a `Jenkinsfile`) to build your Pipeline projects,
- extend Pipeline with shared libraries,
- use different development tools to facilitate the creation of your Pipeline, and
- work with Pipeline syntax - this page is a comprehensive reference of all Declarative Pipeline syntax.

For an overview of content in the Jenkins User Handbook, see User Handbook overview.

What is Jenkins Pipeline?

Jenkins Pipeline (or simply "Pipeline" with a capital "P") is a suite of plugins which supports implementing and integrating *continuous delivery pipelines* into Jenkins.

A *continuous delivery (CD) pipeline* is an automated expression of your process for getting software from version control right through to your users and customers. Every change to your software (committed in source control) goes through a complex process on its way to being released. This process involves building the software in a reliable and repeatable manner, as well as progressing the built software (called a "build") through multiple stages of testing and deployment.

Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline domain-specific language (DSL) syntax. [1: Domain-specific language]

The definition of a Jenkins Pipeline is written into a text file (called a `Jenkinsfile`) which in turn can be committed to a project's source control repository. [2: Source control management] This is the foundation of "Pipeline-as-code"; treating the CD pipeline a part of the application to be versioned and reviewed like any other code.

Creating a `Jenkinsfile` and committing it to source control provides a number of immediate benefits:

- Automatically creates a Pipeline build process for all branches and pull requests.
- Code review/iteration on the Pipeline (along with the remaining source code).
- Audit trail for the Pipeline.
- Single source of truth [3: Single source of truth] for the Pipeline, which can be viewed and edited by multiple members of the project.

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

Declarative versus Scripted Pipeline syntax

A `Jenkinsfile` can be written using two types of syntax - Declarative and Scripted.

Declarative and Scripted Pipelines are constructed fundamentally differently. Declarative Pipeline is a more recent feature of Jenkins Pipeline which:

- provides richer syntactical features over Scripted Pipeline syntax, and
- is designed to make writing and reading Pipeline code easier.

Many of the individual syntactical components (or "steps") written into a `Jenkinsfile`, however, are common to both Declarative and Scripted Pipeline. Read more about how these two types of syntax differ in [pipeline-concepts] and [pipeline-syntax-overview] below.

Why Pipeline?

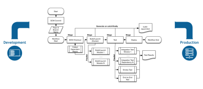
Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive CD pipelines. By modeling a series of related tasks, users 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 CD requirements, including the ability to fork/join, loop, and perform work in parallel.
- **Extensible**: The Pipeline plugin supports custom extensions to its DSL [1: Domain-specific language] and multiple options for integration with other plugins.

While Jenkins has always allowed rudimentary forms of chaining Freestyle Jobs together to perform sequential tasks, [4: 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. [5: plugin:github-organization-folder[GitHub Organization Folder plugin]]

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



Pipeline concepts

The following concepts are key aspects of Jenkins Pipeline, which tie in closely to Pipeline syntax (see the overview below).

Pipeline

A Pipeline is a user-defined model of a CD pipeline. A Pipeline's code defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.

Also, a `pipeline` block is a key part of Declarative Pipeline syntax.

Node

A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.

Also, a `node` block is a key part of Scripted Pipeline syntax.

Stage

A `stage` block defines a conceptually distinct subset of tasks performed through the entire Pipeline (e.g. "Build", "Test" and "Deploy" stages), which is used by many plugins to visualize or present Jenkins Pipeline status/progress. [6: Blue Ocean, plugin:pipeline-stage-view[Pipeline: Stage View plugin]]

Step

A single task. Fundamentally, a step tells Jenkins *what* to do at a particular point in time (or "step" in the process). For example, to execute the shell command `make` use the `sh` step: `sh 'make'`. When a plugin extends the Pipeline DSL, [1: Domain-specific language] that typically means the plugin has implemented a new *step*.

Pipeline syntax overview

The following Pipeline code skeletons illustrate the fundamental differences between Declarative Pipeline syntax and Scripted Pipeline syntax.

Be aware that both stages and steps (above) are common elements of both Declarative and Scripted Pipeline syntax.

Declarative Pipeline fundamentals

In Declarative Pipeline syntax, the `pipeline` block defines all the work done throughout your entire Pipeline.

```
// Declarative //
pipeline {
    agent any ①
    stages {
        stage('Build') { ②
            steps {
                // ③
            }
        }
        stage('Test') { ④
            steps {
                // ⑤
            }
        }
        stage('Deploy') { ⑥
            steps {
                // ⑦
            }
        }
    }
}
// Script //
```

① Execute this Pipeline or any of its stages, on any available agent.

② Defines the "Build" stage.

③ Perform some steps related to the "Build" stage.

④ Defines the "Test" stage.

⑤ Perform some steps related to the "Test" stage.

⑥ Defines the "Deploy" stage.

⑦ Perform some steps related to the "Deploy" stage.

Scripted Pipeline fundamentals

In Scripted Pipeline syntax, one or more `node` blocks do the core work throughout the entire Pipeline. Although this is not a mandatory requirement of Scripted Pipeline syntax, confining your Pipeline's work inside of a `node` block 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.

```
// Declarative //
// Script //
node {  ①
    stage('Build') { ②
        // ③
    }
    stage('Test') { ④
        // ⑤
    }
    stage('Deploy') { ⑥
        // ⑦
    }
}
```

① Execute this Pipeline or any of its stages, on any available agent.

② Defines the "Build" stage. `stage` blocks are optional in Scripted Pipeline syntax. However, implementing `stage` blocks in a Scripted Pipeline provides clearer visualization of each `stage's subset of tasks/steps in the Jenkins UI.

③ Perform some steps related to the "Build" stage.

④ Defines the "Test" stage.

⑤ Perform some steps related to the "Test" stage.

⑥ Defines the "Deploy" stage.

⑦ Perform some steps related to the "Deploy" stage.

Pipeline example

Here is an example of a `Jenkinsfile` using Declarative Pipeline syntax - its Scripted syntax equivalent can be accessed by clicking the **Toggle Scripted Pipeline** link below:

```
// Declarative //
pipeline { ①
    agent any ②
    stages {
        stage('Build') { ③
            steps { ④
                sh 'make' ⑤
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml' ⑥
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
// Script //
node { ⑦
    stage('Build') { ③
        sh 'make' ⑤
    }
    stage('Test') {
        sh 'make check'
        junit 'reports/**/*.xml' ⑥
    }
    stage('Deploy') {
        sh 'make publish'
    }
}
```

① `pipeline` is Declarative Pipeline-specific syntax that defines a "block" containing all content and instructions for executing the entire Pipeline.

② `agent` is Declarative Pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline.

③ `stage` is a syntax block that describes a stage of this Pipeline. Read more about `stage` blocks in Declarative Pipeline syntax on the Pipeline syntax page. As mentioned above, `stage` blocks are optional in Scripted Pipeline syntax.

④ `steps` is Declarative Pipeline-specific syntax that describes the steps to be run in this `stage`.

⑤ `sh` is a Pipeline step (provided by the plugin:workflow-durable-task-step[Pipeline: Nodes and Processes plugin]) that executes the given shell command.

⑥ `junit` is another a Pipeline step (provided by the plugin:junit[JUnit plugin]) for aggregating test reports.

⑦ `node` is Scripted Pipeline-specific syntax that instructs Jenkins to execute this Pipeline (and any stages contained within it), on any available agent/node. This is effectively equivalent to `agent` in Declarative Pipeline-specific syntax.

Read more about Pipeline syntax on the Pipeline Syntax page.

Getting started with Pipeline

As mentioned previously, Jenkins Pipeline is a suite of plugins that 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. [7: Domain-specific language]

This section describes how to get started with creating your Pipeline project in Jenkins and introduces you to the various ways that a `Jenkinsfile` can be created and stored.

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, [8: Pipeline plugin] which is installed as part of the "suggested plugins" (specified when running through the Post-installation setup wizard after installing Jenkins).

Read more about how to install and manage plugins in Managing Plugins.

Defining a Pipeline

Both Declarative and Scripted Pipeline are DSLs [1: Domain-specific language] to describe portions of your software delivery pipeline. Scripted Pipeline is written in a limited form of Groovy syntax.

Relevant components of Groovy syntax will be introduced as required throughout this documentation, so while an understanding of Groovy is helpful, it is not required to work with Pipeline.

A Pipeline can be created in one of the following ways:

- [through-blue-ocean] - after setting up a Pipeline project in Blue Ocean, the Blue Ocean UI helps you write your Pipeline's `Jenkinsfile` and commit it to source control.
- [through-the-classic-ui] - you can enter a basic Pipeline directly in Jenkins through the classic UI.
- In SCM - you can write a `Jenkinsfile` manually, which you can commit to your project's source control repository. [9: Source control management]

The syntax for defining a Pipeline with either approach is the same, but while Jenkins supports entering Pipeline directly into the classic UI, it is generally considered best practice to define the Pipeline in a `Jenkinsfile` which Jenkins will then load directly from source control.

Through Blue Ocean

If you are new to Jenkins Pipeline, the Blue Ocean UI helps you set up your Pipeline project, and automatically creates and writes your Pipeline (i.e. the `Jenkinsfile`) for you through the graphical Pipeline editor.

As part of setting up your Pipeline project in Blue Ocean, Jenkins configures a secure and appropriately authenticated connection to your project's source control repository. Therefore, any changes you make to the `Jenkinsfile` via Blue Ocean's Pipeline editor are automatically saved and committed to source control.

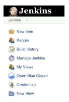
Read more about Blue Ocean in the Blue Ocean chapter and Getting started with Blue Ocean page.

Through the classic UI

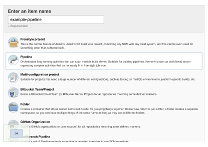
A `Jenkinsfile` created using the classic UI is stored by Jenkins itself (within the Jenkins home directory).

To create a basic Pipeline through the Jenkins classic UI:

1. If required, ensure you are logged in to Jenkins.
2. From the Jenkins home page (i.e. the Dashboard of the Jenkins classic UI), click **New Item** at the top left.



3. In the **Enter an item name** field, specify the name for your new Pipeline project.
   Caution: Jenkins uses this item name to create directories on disk. It is recommended to avoid using spaces in item names, since doing so may uncover bugs in scripts that do not properly handle spaces in directory paths.
4. Scroll down and click **Pipeline**, then click **OK** at the end of the page to open the Pipeline configuration page (whose **General** tab is selected).



Enter an item name
example-pipeline
» Required field

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

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

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

Bitbucket Team/Project
Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.

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

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

OK

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

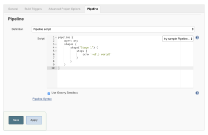
5. Click the **Pipeline** tab at the top of the page to scroll down to the **Pipeline** section.
   Note: If instead you are defining your `Jenkinsfile` in source control, follow the instructions in In SCM below.
6. In the **Pipeline** section, ensure that the **Definition** field indicates the **Pipeline script** option.
7. Enter your Pipeline code into the **Script** text area.
   For instance, copy the following Declarative example Pipeline code (below the *Jenkinsfile (…)* heading) or its Scripted version equivalent and paste this into the **Script** text area. (The Declarative example below is used throughout the remainder of this procedure.)

```
// Declarative //
pipeline {
    agent any ①
    stages {
        stage('Stage 1') {
            steps {
                echo 'Hello world!' ②
            }
        }
    }
}
// Script //
node { ③
    stage('Stage 1') {
        echo 'Hello World' ②
    }
}
```

① `agent` instructs Jenkins to allocate an executor (on any available agent/node in the Jenkins environment) and workspace for the entire Pipeline.

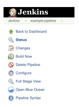
② `echo` writes simple string in the console output.

③ `node` effectively does the same as `agent` (above).

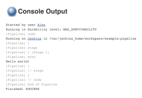


Note: You can also select from canned *Scripted* Pipeline examples from the **try sample Pipeline** option at the top right of the **Script** text area. Be aware that there are no canned Declarative Pipeline examples available from this field.

8. Click **Save** to open the Pipeline project/item view page.
9. On this page, click **Build Now** on the left to run the Pipeline.



10. Under **Build History** on the left, click **#1** to access the details for this particular Pipeline run.
11. Click **Console Output** to see the full output from the Pipeline run. The following output shows a successful run of your Pipeline.



Console Output

```
Started by user Alex
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/example-pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Stage 1)
[Pipeline] echo
Hello world!
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Notes:

- You can also access the console output directly from the Dashboard by clicking the colored globe to the left of the build number (e.g. **#1**).

    - Defining a Pipeline through the classic UI is convenient for testing Pipeline code snippets, or for handling simple Pipelines or Pipelines that do not require source code to be checked out/cloned from a repository. As mentioned above, unlike `Jenkinsfile`s you define through Blue Ocean (above) or in source control (below), `Jenkinsfile`s entered into the **Script** text area area of Pipeline projects are stored by Jenkins itself, within the Jenkins home directory. Therefore, for greater control and flexibility over your Pipeline, particularly for projects in source control that are likely to gain complexity, it is recommended that you use Blue Ocean or source control to define your `Jenkinsfile`.

In SCM

Complex Pipelines are difficult to write and maintain within the classic UI's **Script** text area of the Pipeline configuration page.

To make this easier, your Pipeline's `Jenkinsfile` can be written in a text editor or integrated development environment (IDE) and committed to source control [2: Source control management] (optionally with the application code that Jenkins will build). Jenkins can then check out your `Jenkinsfile` from source control as part of your Pipeline project's build process and then proceed to execute your Pipeline.

To configure your Pipeline project to use a `Jenkinsfile` from source control:

1. Follow the procedure above for defining your Pipeline through the classic UI until you reach step 5 (accessing the **Pipeline** section on the Pipeline configuration page).
2. From the **Definition** field, choose the **Pipeline script from SCM** option.
3. From the **SCM** field, choose the type of source control system of the repository containing your `Jenkinsfile`.
4. Complete the fields specific to your repository's source control system.
   Tip: If you are uncertain of what value to specify for a given field, click its **?** icon to the right for more information.
5. In the **Script Path** field, specify the location (and name) of your `Jenkinsfile`. This location is the one that Jenkins checks out/clones the repository containing your `Jenkinsfile`, which should match that of the repository's file structure. The default value of this field assumes that your `Jenkinsfile` is named "Jenkinsfile" and is located at the root of the repository.

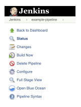
When you update the designated repository, a new build is triggered, as long as the Pipeline is configured with an SCM polling trigger.

TIP Since Pipeline code (i.e. Scripted Pipeline in particular) is written in Groovy-like syntax, if your IDE is not correctly syntax highlighting your `Jenkinsfile`, try inserting the line `#!/usr/bin/env groovy` at the top of the `Jenkinsfile`, [12: Shebang (general definition)] [13: Shebang line (Groovy syntax)] which may rectify the issue.

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.



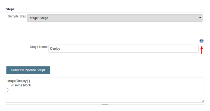
Snippet Generator

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.



Sample Step: stage: Stage

Stage Name: Deploy

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

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

Global Variable Reference

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

The variables provided by default in Pipeline are:

env

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

params

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

currentBuild

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

Declarative Directive Generator

While the Snippet Generator helps with generating steps for a Scripted Pipeline or for the `steps` block in a `stage` in a Declarative Pipeline, it does not cover the sections and directives used to define a Declarative Pipeline. The "Declarative Directive Generator" utility helps with that. Similar to the Snippet Generator, the Directive Generator allows you to choose a Declarative directive, configure it in a form, and generate the configuration for that directive, which you can then use in your Declarative Pipeline.

To generate a Declarative directive using the Declarative Directive Generator:

1. Navigate to the **Pipeline Syntax** link (referenced above) from a configured Pipeline, and then click on the **Declarative Directive Generator** link in the sidepanel, or go directly to localhost:8080/directive-generator.
2. Select the desired directive in the dropdown menu
3. Use the dynamically populated area below the dropdown to configure the selected directive.
4. Click **Generate Directive** to create the directive's configuration to copy into your Pipeline.

The Directive Generator can generate configuration for nested directives, such as conditions inside a `when` directive, but it cannot generate Pipeline steps. For the contents of directives which contain steps, such as `steps` insde a `stage` or conditions like `always` or `failure` inside `post`, the Directive Generator adds a placeholder comment instead. You will still need to add steps to your Pipeline by hand.

```
// Declarative //
stage('Stage 1') {
    steps {
        // One or more steps need to be included within the steps block.
    }
}
// Script //
```

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 with Pipeline 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 Defining a Pipeline in SCM, 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 check || 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-failure] section below.

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

Deploy

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

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

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

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

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

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

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

Working with your Jenkinsfile

The following sections provide details about handling:

- specific Pipeline syntax in your `Jenkinsfile` and
- features and functionality of Pipeline syntax which are essential in building your application or Pipeline project.

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.

Using environment variables

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`.

Setting environment variables dynamically

In the case where environment variable need to be set dynamically at run time this can be done with the use of a shell scripts (`sh`), Windows Batch Script (`bat`) or PowerShell Script (`powershell`). Each script can either `returnStatus` or `returnStdout`. More information on scripts.

Below is an example in a declarative pipeline using `sh` (shell) with both `returnStatus` and `returnStdout`.

```
// Declarative //
pipeline {
    agent any ①
    environment {
        // Using returnStdout
        CC = """${sh(
                returnStdout: true,
                script: 'echo "clang"'
            )}""" ②
        // Using returnStatus
        EXIT_STATUS = """${sh(
                returnStatus: true,
                script: 'exit 1'
            )}"""
    }
    stages {
        stage('Example') {
            environment {
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
// Script //
```

① An `agent` must be set at the top level of the pipeline. This will fail if agent is set as `agent none`.

② When using `returnStdout` a trailing whitespace will be append to the returned string. Use `.trim()` to remove this.

Handling credentials

Credentials configured in Jenkins can be handled in Pipelines for immediate use. Read more about using credentials in Jenkins on the Using credentials page.

For secret text, usernames and passwords, and secret files

Jenkins' declarative Pipeline syntax has the `credentials()` helper method (used within the `environment` directive) which supports secret text, username and password, as well as secret file credentials. If you want to handle other types of credentials, refer to the For other credential types section (below).

Secret text

The following Pipeline code shows an example of how to create a Pipeline using environment variables for secret text credentials.

In this example, two secret text credentials are assigned to separate environment variables to access Amazon Web Services (AWS). These credentials would have been configured in Jenkins with their respective credential IDs
`jenkins-aws-secret-key-id` and `jenkins-aws-secret-access-key`.

```
// Declarative //
pipeline {
    agent {
        // Define agent details here
    }
    environment {
        AWS_ACCESS_KEY_ID     = credentials('jenkins-aws-secret-key-id')
        AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')
    }
    stages {
        stage('Example stage 1') {
            steps {
                // ①
            }
        }
        stage('Example stage 2') {
            steps {
                // ②
            }
        }
    }
}
// Script //
```

① You can reference the two credential environment variables (defined in this Pipeline's `environment` directive), within this stage's steps using the syntax `$AWS_ACCESS_KEY_ID` and `$AWS_SECRET_ACCESS_KEY`. For example, here you can authenticate to AWS using the secret text credentials assigned to these credential variables.
To maintain the security and anonymity of these credentials, if you attempt to retrieve the value of these credential variables from within the Pipeline (e.g. `echo $AWS_SECRET_ACCESS_KEY`), Jenkins only returns the value "****" to prevent secret information from being written to the console

output and any logs. Any sensitive information in credential IDs themselves (such as usernames) are also returned as "****" in the Pipeline run's output.

② In this Pipeline example, the credentials assigned to the two `AWS_…` environment variables are scoped globally for the entire Pipeline, so these credential variables could also be used in this stage's steps. If, however, the `environment` directive in this Pipeline were moved to a specific stage (as is the case in the Usernames and passwords Pipeline example below), then these `AWS_…` environment variables would only be scoped to the steps in that stage.

Usernames and passwords

The following Pipeline code snippets show an example of how to create a Pipeline using environment variables for username and password credentials.

In this example, username and password credentials are assigned to environment variables to access a Bitbucket repository in a common account or team for your organization; these credentials would have been configured in Jenkins with the credential ID `jenkins-bitbucket-common-creds`.

When setting the credential environment variable in the `environment` directive:

```
environment {
    BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-creds')
}
```

this actually sets the following three environment variables:

- `BITBUCKET_COMMON_CREDS` - contains a username and a password separated by a colon in the format `username:password`.
- `BITBUCKET_COMMON_CREDS_USR` - an additional variable containing the username component only.
- `BITBUCKET_COMMON_CREDS_PSW` - an additional variable containing the password component only.

NOTE

By convention, variable names for environment variables are typically specified in capital case, with individual words separated by underscores. You can, however, specify any legitimate variable name using lower case characters. Bear in mind that the additional environment variables created by the `credentials()` method (above) will always be appended with `_USR` and `_PSW` (i.e. in the format of an underscore followed by three capital letters).

The following code snippet shows the example Pipeline in its entirety:

```
// Declarative //
pipeline {
    agent {
        // Define agent details here
    }
    stages {
        stage('Example stage 1') {
            environment {
                BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-creds')
            }
            steps {
                // ①
            }
        }
        stage('Example stage 2') {
            steps {
                // ②
            }
        }
    }
}
// Script //
```

① The following credential environment variables (defined in this Pipeline's `environment` directive) are available within this stage's steps and can be referenced using the syntax:

- `$BITBUCKET_COMMON_CREDS`
- `$BITBUCKET_COMMON_CREDS_USR`
- `$BITBUCKET_COMMON_CREDS_PSW`

For example, here you can authenticate to Bitbucket with the username and password assigned to these credential variables.
To maintain the security and anonymity of these credentials, if you attempt to retrieve the value of these credential variables from within the Pipeline, the same behavior described in the Secret text example above applies to these username and password credential variable types too.

② In this Pipeline example, the credentials assigned to the three `COMMON_BITBUCKET_CREDS…` environment variables are scoped only to `Example stage 1`, so these credential variables are not available for use in this `Example stage 2` stage's steps. If, however, the `environment` directive in this Pipeline were moved immediately within the `pipeline` block (as is the case in the Secret text Pipeline example above), then these `COMMON_BITBUCKET_CREDS…` environment variables would be scoped globally and could be used in any stage's steps.

Secret files

As far as Pipelines are concerned, secret files are handled in exactly the same manner as secret text (above).

Essentially, the only difference between secret text and secret file credentials are that for secret text, the credential itself is entered directly into Jenkins whereas for a secret file, the credential is originally stored in a file which is then uploaded to Jenkins.

Unlike secret text, secret files cater for credentials that are:

- too unwieldy to enter directly into Jenkins, and/or
- in binary format, such as a GPG file.

For other credential types

If you need to set credentials in a Pipeline for anything other than secret text, usernames and passwords, or secret files (above) - i.e SSH keys or certificates, then use Jenkins' **Snippet Generator** feature, which you can access through Jenkins' classic UI.

To access the **Snippet Generator** for your Pipeline project/item:

1. From the Jenkins home page (i.e. the Dashboard of Jenkins' classic UI), click the name of your Pipeline project/item.
2. On the left, click **Pipeline Syntax** and ensure that the **Snippet Generator** link is in bold at the top-left. (If not, click its link.)
3. From the **Sample Step** field, choose **withCredentials: Bind credentials to variables**.
4. Under **Bindings**, click **Add** and choose from the dropdown:
    - **SSH User Private Key** - to handle SSH public/private key pair credentials, from which you can specify:
        - **Key File Variable** - the name of the environment variable that will be bound to these credentials. Jenkins actually assigns this temporary variable to the secure location of the private key file required in the SSH public/private key pair authentication process.
        - **Passphrase Variable** (*Optional*) - the name of the environment variable that will be bound to the passphrase associated with the SSH public/private key pair.
        - **Username Variable** (*Optional*) - the name of the environment variable that will be bound to username associated with the SSH public/private key pair.
        - **Credentials** - choose the SSH public/private key credentials stored in Jenkins. The value of this field is the credential ID, which Jenkins writes out to the generated snippet.
    - **Certificate** - to handle PKCS#12 certificates, from which you can specify:
        - **Keystore Variable** - the name of the environment variable that will be bound to these credentials. Jenkins actually assigns this temporary variable to the secure location of the certificate's keystore required in the certificate authentication process.
        - **Password Variable** (*Optional*) - the name of the environment variable that will be bound to the password associated with the certificate.
        - **Alias Variable** (*Optional*) - the name of the environment variable that will be bound to the unique alias associated with the certificate.
        - **Credentials** - choose the certificate credentials stored in Jenkins. The value of this field is the credential ID, which Jenkins writes out to the generated snippet.
    - **Docker client certificate** - to handle Docker Host Certificate Authentication.

5. Click **Generate Pipeline Script** and Jenkins generates a `withCredentials( … ) { … }` Pipeline step snippet for the credentials you specified, which you can then copy and paste into your Declarative or Scripted Pipeline code.
   Notes:
   - The **Credentials** fields (above) show the names of credentials configured in Jenkins. However, these values are converted to credential IDs after clicking **Generate Pipeline Script**.
   - To combine more than one credential in a single `withCredentials( … ) { … }` Pipeline step, see Combining credentials in one step (below) for details.

SSH User Private Key example

```
withCredentials(bindings: [sshUserPrivateKey(credentialsId: 'jenkins-ssh-key-for-abc',
\
                                             keyFileVariable: 'SSH_KEY_FOR_ABC', \
                                             passphraseVariable: '', \
                                             usernameVariable: '')]) {
  // some block
}
```

The optional `passphraseVariable` and `usernameVariable` definitions can be deleted in your final Pipeline code.

Certificate example

```
withCredentials(bindings: [certificate(aliasVariable: '', \
                                       credentialsId: 'jenkins-certificate-for-xyz', \
                                       keystoreVariable: 'CERTIFICATE_FOR_XYZ', \
                                       passwordVariable: 'XYZ-CERTIFICATE-PASSWORD')])
{
  // some block
}
```

The optional `aliasVariable` and `passwordVariable` variable definitions can be deleted in your final Pipeline code.

The following code snippet shows an example Pipeline in its entirety, which implements the **SSH User Private Key** and **Certificate** snippets above:

```
// Declarative //
pipeline {
    agent {
        // define agent details
    }
    stages {
        stage('Example stage 1') {
            steps {
                withCredentials(bindings: [sshUserPrivateKey(credentialsId: 'jenkins-
ssh-key-for-abc', \
                                                             keyFileVariable:
'SSH_KEY_FOR_ABC')]) {
                  // ①
                }
                withCredentials(bindings: [certificate(credentialsId: 'jenkins-
certificate-for-xyz', \
                                                       keystoreVariable:
'CERTIFICATE_FOR_XYZ', \
                                                       passwordVariable: 'XYZ-
CERTIFICATE-PASSWORD')]) {
                  // ②
                }
            }
        }
        stage('Example stage 2') {
            steps {
                // ③
            }
        }
    }
}
// Script //
```

① Within this step, you can reference the credential environment variable with the syntax `$SSH_KEY_FOR_ABC`. For example, here you can authenticate to the ABC application with its configured SSH public/private key pair credentials, whose **SSH User Private Key** file is assigned to `$SSH_KEY_FOR_ABC`.

② Within this step, you can reference the credential environment variable with the syntax `$CERTIFICATE_FOR_XYZ` and `$XYZ-CERTIFICATE-PASSWORD`. For example, here you can authenticate to the XYZ application with its configured certificate credentials, whose **Certificate**'s keystore file and password are assigned to the variables `$CERTIFICATE_FOR_XYZ` and `$XYZ-CERTIFICATE-PASSWORD`, respectively.

③ In this Pipeline example, the credentials assigned to the `$SSH_KEY_FOR_ABC`, `$CERTIFICATE_FOR_XYZ` and `$XYZ-CERTIFICATE-PASSWORD` environment variables are scoped only within their respective `withCredentials( … ) { … }` steps, so these credential variables are not available for use in this `Example stage 2` stage’s steps.

To maintain the security and anonymity of these credentials, if you attempt to retrieve the value of these credential variables from within these `withCredentials( … ) { … }` steps, the same behavior described in the Secret text example (above) applies to these SSH public/private key pair credential and certificate variable types too.

NOTE

- When using the **Sample Step** field's **withCredentials: Bind credentials to variables** option in the **Snippet Generator**, only credentials which your current Pipeline project/item has access to can be selected from any **Credentials** field's list. While you can manually write a `withCredentials( … ) { … }` step for your Pipeline (like the examples above), using the **Snippet Generator** is recommended to avoid specifying credentials that are out of scope for this Pipeline project/item, which when run, will make the step fail.
- You can also use the **Snippet Generator** to generate `withCredentials( … ) { … }` steps to handle secret text, usernames and passwords and secret files. However, if you only need to handle these types of credentials, it is recommended you use the relevant procedure described in the section above for improved Pipeline code readability.
- The use of **single-quotes** instead of **double-quotes** to define the `script` (the implicit parameter to `sh`) in Groovy above. The single-quotes will cause the secret to be expanded by the shell as an environment variable. The double-quotes are potentially less secure as the secret is interpolated by Groovy, and so typical operating system process listings (as well as Blue Ocean, and the pipeline steps tree in the classic UI) will accidentally disclose it :

```
node {
  withCredentials([string(credentialsId: 'mytoken', variable: 'TOKEN')])
{
    sh /* WRONG! */ """
      set +x
      curl -H 'Token: $TOKEN' https://some.api/
    """
    sh /* CORRECT */ '''
      set +x
      curl -H 'Token: $TOKEN' https://some.api/
    '''
  }
}
```

Combining credentials in one step

Using the **Snippet Generator**, you can make multiple credentials available within a single `withCredentials( … ) { … }` step by doing the following:

1. From the Jenkins home page (i.e. the Dashboard of Jenkins' classic UI), click the name of your Pipeline project/item.
2. On the left, click **Pipeline Syntax** and ensure that the **Snippet Generator** link is in bold at the top-left. (If not, click its link.)

3. From the **Sample Step** field, choose **withCredentials: Bind credentials to variables**.
4. Click **Add** under **Bindings**.
5. Choose the credential type to add to the `withCredentials( … ) { … }` step from the dropdown list.
6. Specify the credential **Bindings** details. Read more above these in the procedure under For other credential types (above).
7. Repeat from "Click **Add** …" (above) for each (set of) credential/s to add to the `withCredentials( … ) { … }` step.
8. Click **Generate Pipeline Script** to generate the final `withCredentials( … ) { … }` step snippet.

Handling 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 failure

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.

Parallel execution

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.

Running a Pipeline

Multibranch

See the Multibranch documentation for more information.

Parameters

See the Jenkinsfile documentation for more information

Restarting or Rerunning a Pipeline

There are a number of ways to rerun or restart a completed Pipeline.

Replay

See the Replay documentation for more information.

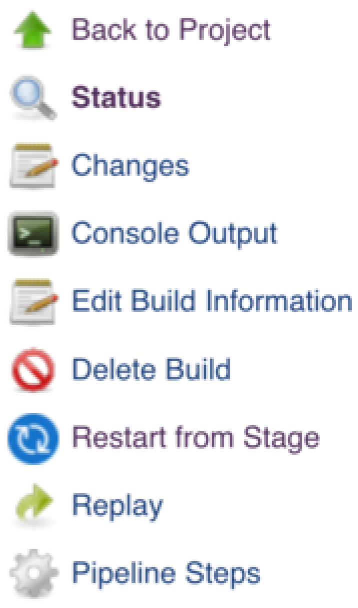
Restart from a Stage

You can restart any completed Declarative Pipeline from any top-level stage which ran in that Pipeline. This allows you to rerun a Pipeline from a stage which failed due to transient or environmental considerations, for example. All inputs to the Pipeline will be the same. This includes SCM information, build parameters, and the contents of any `stash` step calls in the original Pipeline, if specified.

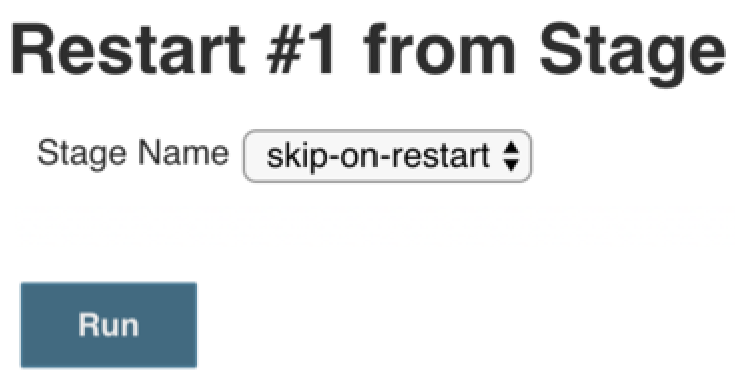
Currently, the UI for restarting a stage in a Declarative Pipeline is only available in the Classic Jenkins UI. Blue Ocean will be adding support for stage restarting in the near future.

How to Use

No additional configuration is needed in the Jenkinsfile to allow you to restart stages in your Declarative Pipelines. This is an inherent part of Declarative Pipelines and is available automatically. Once your Pipeline has completed, whether it succeeds or fails, you can go to the side panel for the run in the classic UI and click on "Restart from Stage".



You will be prompted to choose from a list of top-level stages that were executed in the original run, in the order they were executed. Stages which were skipped due to an earlier failure will not be available to be restarted, but stages which were skipped due to a `when` condition not being satisfied will be available. The parent stage for a group of `parallel` stages, or a group of nested `stages` to be run sequentially will also not be available - only top-level stages are allowed.



Once you choose a stage to restart from and click submit, a new build, with a new build number, will be started. All stages before the selected stage will be skipped, and the Pipeline will start executing at the selected stage. From that point on, the Pipeline will run as normal.

Preserving `stash`es for Use with Restarted Stages

Normally, when you run the `stash` step in your Pipeline, the resulting stash of artifacts is cleared when the Pipeline completes, regardless of the result of the Pipeline. Since `stash` artifacts aren't accessible outside of the Pipeline run that created them, this has not created any limitations on usage. But with Declarative stage restarting, you may want to be able to `unstash` artifacts from a stage which ran before the stage you're restarting from.

To enable this, there is a job property that allows you to configure a maximum number of completed runs whose `stash` artifacts should be preserved for reuse in a restarted run. You can specify anywhere from 1 to 50 as the number of runs to preserve.

This job property can be configured in your Declarative Pipeline's `options` section, as below:

```
options {
    preserveStashes() ①
    // or
    preserveStashes(buildCount: 5) ②
}
```

① The default number of runs to preserve is 1, just the most recent completed build.

② If a number for `buildCount` outside of the range of 1 to 50 is specified, the Pipeline will fail with a validation error.

When a Pipeline completes, it will check to see if any previously completed runs should have their `stash` artifacts cleared.

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.

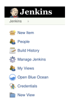
Creating a Multibranch Pipeline

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.

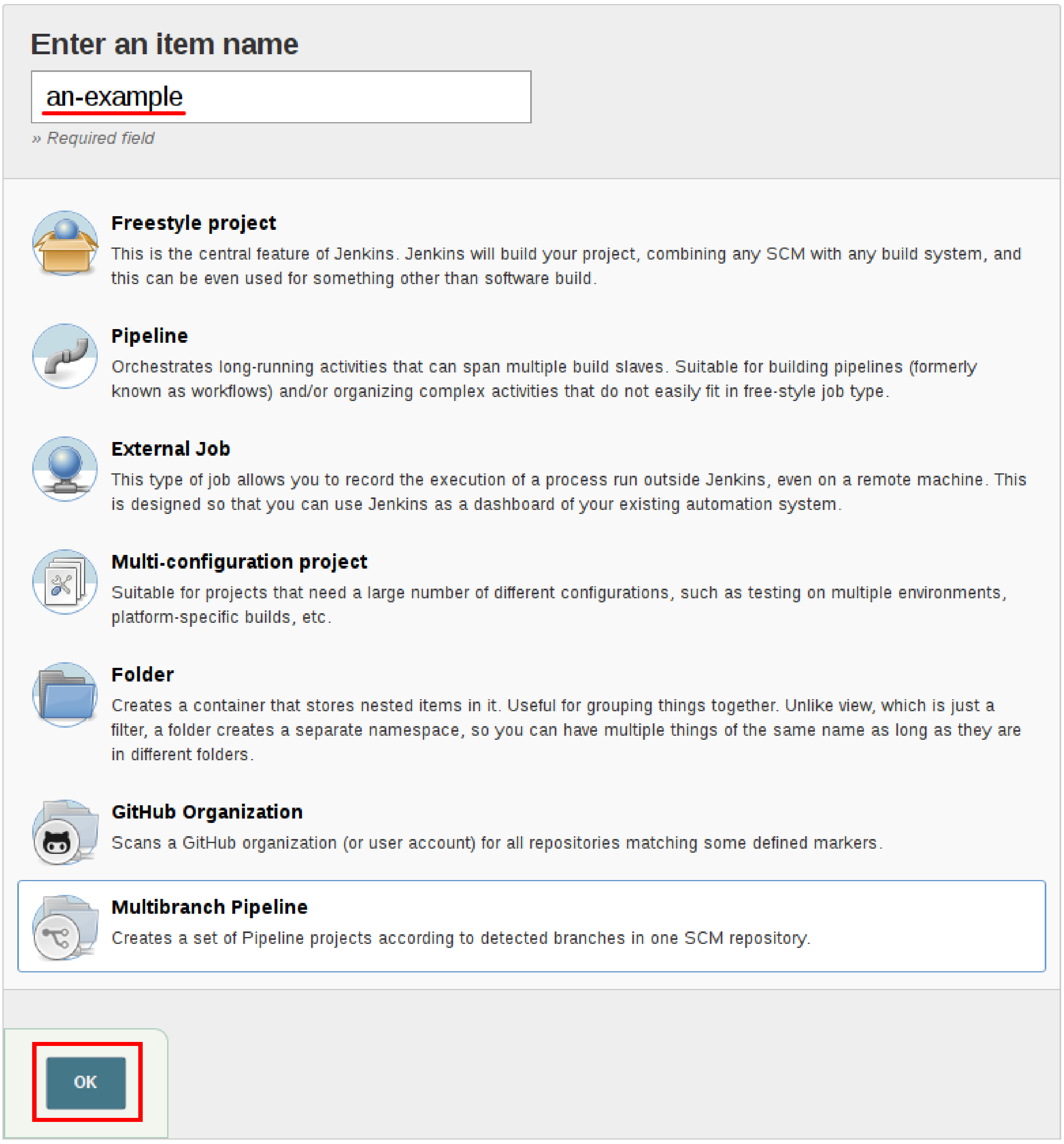
To create a Multibranch Pipeline:

- 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.



Enter an item name

an-example

» Required field

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

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

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

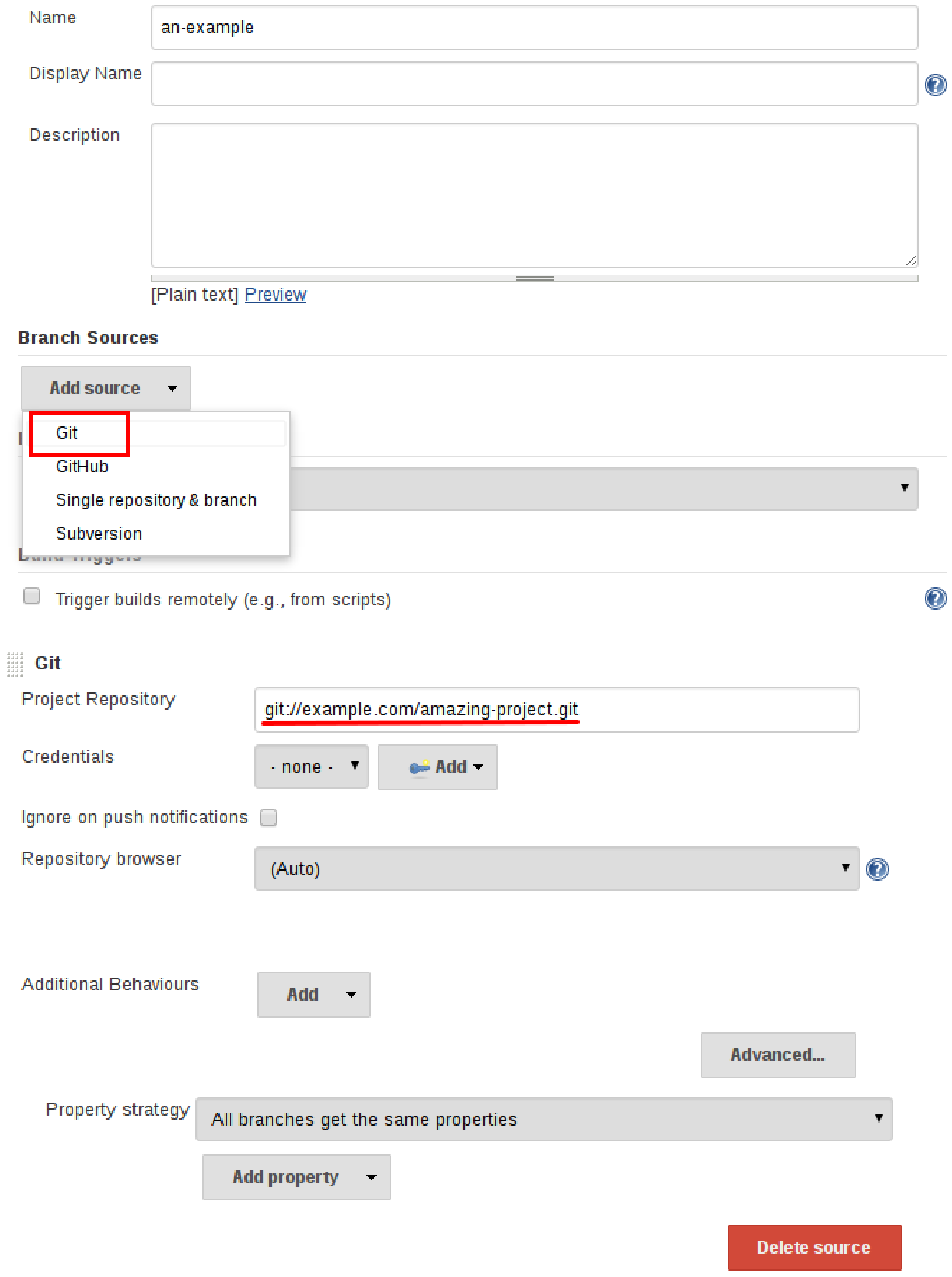
Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

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

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

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

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


Name: an-example

Display Name:

Description:

[Plain text] Preview

Branch Sources

Add source: Git, GitHub, Single repository & branch, Subversion

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

Git

Project Repository: git://example.com/amazing-project.git

Credentials: - none -

Ignore on push notifications: ☐

Repository browser: (Auto)

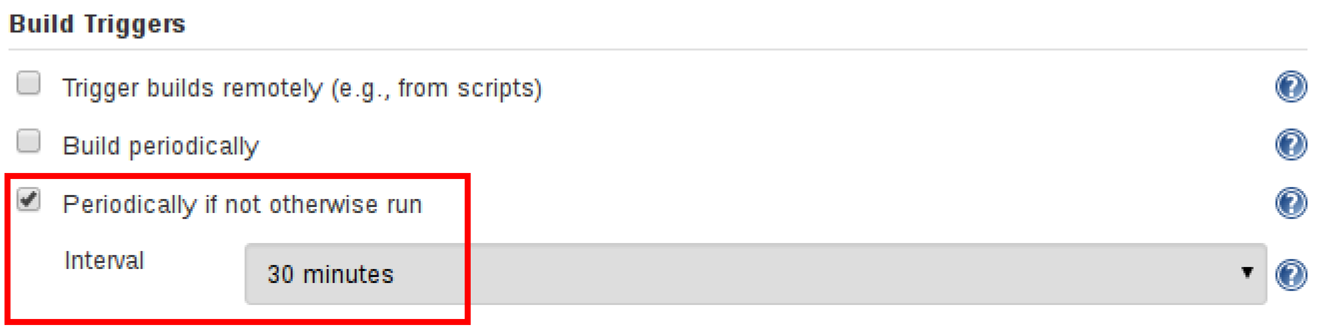
Additional Behaviours

Property strategy: All branches get the same properties

- **Save** the Multibranch Pipeline project.

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

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


Build Triggers

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

☐ Build periodically

☑ Periodically if not otherwise run

Interval: 30 minutes

Additional Environment Variables

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

BRANCH_NAME

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

CHANGE_ID

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

Additional environment variables are listed in the Global Variable Reference.

Supporting Pull Requests

With the "GitHub" or "Bitbucket" Branch Sources, Multibranch Pipelines can be used for validating pull/change requests. This functionality is provided, respectively, by the 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.

Using Docker with Pipeline

Many organizations use Docker to unify their build and test environments across machines, and to provide an efficient mechanism for deploying applications. Starting with Pipeline versions 2.5 and higher, Pipeline has built-in support for interacting with Docker from within a `Jenkinsfile`.

While this section will cover the basics of utilizing Docker from with a `Jenkinsfile`, it will not cover the fundamentals of Docker, which can be read about in the Docker Getting Started Guide.

Customizing the execution environment

Pipeline is designed to easily use Docker images as the execution environment for a single Stage or the entire Pipeline. Meaning that a user can define the tools required for their Pipeline, without having to manually configure agents. Practically any tool which can be packaged in a Docker container. can be used with ease by making only minor edits to a `Jenkinsfile`.

```
// Declarative //
pipeline {
    agent {
        docker { image 'node:7-alpine' }
    }
    stages {
        stage('Test') {
            steps {
                sh 'node --version'
            }
        }
    }
}
// Script //
node {
    /* Requires the Docker Pipeline plugin to be installed */
    docker.image('node:7-alpine').inside {
        stage('Test') {
            sh 'node --version'
        }
    }
}
```

When the Pipeline executes, Jenkins will automatically start the specified container and execute the defined steps within it:

```
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] sh
[guided-tour] Running shell script
+ node --version
v7.4.0
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
```

Caching data for containers

Many build tools will download external dependencies and cache them locally for future re-use. Since containers are initially created with "clean" file systems, this can result in slower Pipelines, as they may not take advantage of on-disk caches between subsequent Pipeline runs.

Pipeline supports adding custom arguments which are passed to Docker, allowing users to specify custom Docker Volumes to mount, which can be used for caching data on the agent between Pipeline runs. The following example will cache `~/.m2` between Pipeline runs utilizing the `maven` container, thereby avoiding the need to re-download dependencies for subsequent runs of the Pipeline.

```
// Declarative //
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v $HOME/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B'
            }
        }
    }
}
// Script //
node {
    /* Requires the Docker Pipeline plugin to be installed */
    docker.image('maven:3-alpine').inside('-v $HOME/.m2:/root/.m2') {
        stage('Build') {
            sh 'mvn -B'
        }
    }
}
```

Using multiple containers

It has become increasingly common for code bases to rely on multiple, different, technologies. For example, a repository might have both a Java-based back-end API implementation *and* a JavaScript-based front-end implementation. Combining Docker and Pipeline allows a `Jenkinsfile` to use **multiple** types of technologies by combining the `agent {}` directive, with different stages.

```
// Declarative //
pipeline {
    agent none
    stages {
        stage('Back-end') {
            agent {
                docker { image 'maven:3-alpine' }
            }
            steps {
                sh 'mvn --version'
            }
        }
        stage('Front-end') {
            agent {
                docker { image 'node:7-alpine' }
            }
            steps {
                sh 'node --version'
            }
        }
    }
}
// Script //
node {
    /* Requires the Docker Pipeline plugin to be installed */

    stage('Back-end') {
        docker.image('maven:3-alpine').inside {
            sh 'mvn --version'
        }
    }

    stage('Front-end') {
        docker.image('node:7-alpine').inside {
            sh 'node --version'
        }
    }
}
```

Using a Dockerfile

For projects which require a more customized execution environment, Pipeline also supports building and running a container from a `Dockerfile` in the source repository. In contrast to the previous approach of using an "off-the-shelf" container, using the `agent { dockerfile true }` syntax will build a new image from a `Dockerfile` rather than pulling one from Docker Hub.

Re-using an example from above, with a more custom `Dockerfile`:

Dockerfile

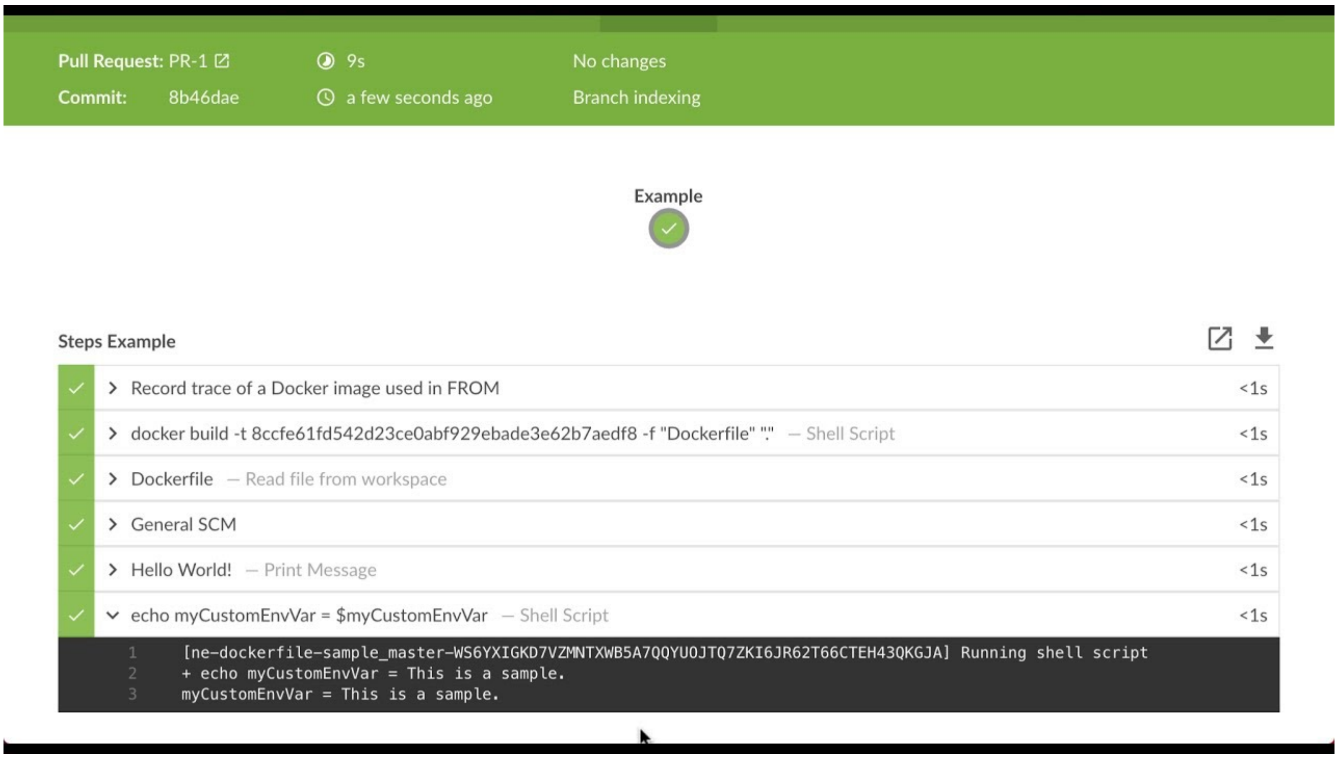
```
FROM node:7-alpine

RUN apk add -U subversion
```

By committing this to the root of the source repository, the `Jenkinsfile` can be changed to build a container based on this `Dockerfile` and then run the defined steps using that container:

```
// Declarative //
pipeline {
    agent { dockerfile true }
    stages {
        stage('Test') {
            steps {
                sh 'node --version'
                sh 'svn --version'
            }
        }
    }
}
// Script //
```

The `agent { dockerfile true }` syntax supports a number of other options which are described in more detail in the Pipeline Syntax section.



Pull Request: PR-1 | 9s | No changes
Commit: 8b46dae | a few seconds ago | Branch indexing

Example

Steps Example

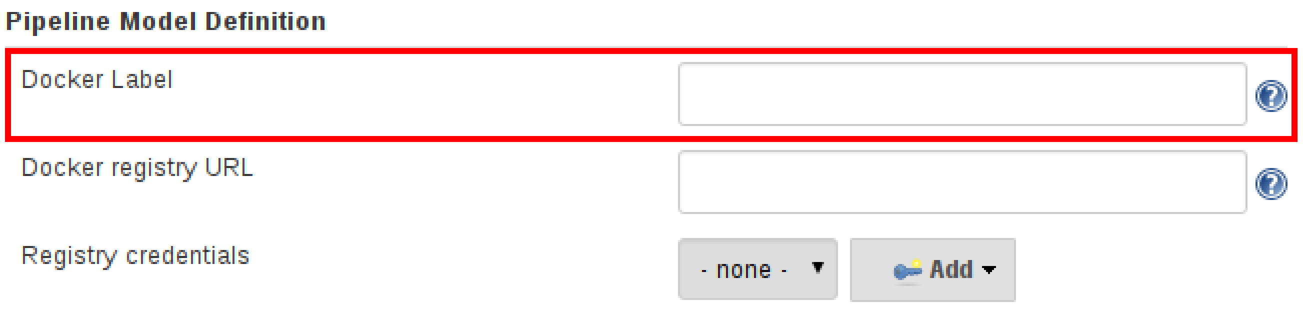
✓	Record trace of a Docker image used in FROM	<1s
✓	docker build -t 8ccfe61fd542d23ce0abf929ebade3e62b7aedf8 -f "Dockerfile" "." — Shell Script	<1s
✓	Dockerfile — Read file from workspace	<1s
✓	General SCM	<1s
✓	Hello World! — Print Message	<1s
✓	echo myCustomEnvVar = $myCustomEnvVar — Shell Script	<1s

```
[ne-dockerfile-sample_master-WS6YXIGKD7VZMNTXWB5A7QQYUOJTQ7ZKI6JR62T66CTEH43QKGJA] Running shell script
+ echo myCustomEnvVar = This is a sample.
myCustomEnvVar = This is a sample.
```

Using a Dockerfile with Jenkins Pipeline

Specifying a Docker Label

By default, Pipeline assumes that *any* configured agent is capable of running Docker-based Pipelines. For Jenkins environments which have macOS, Windows, or other agents, which are unable to run the Docker daemon, this default setting may be problematic. Pipeline provides a global option in the **Manage Jenkins** page, and on the Folder level, for specifying which agents (by Label) to use for running Docker-based Pipelines.



Pipeline Model Definition

Docker Label	
Docker registry URL	
Registry credentials	- none - ▾ · Add ▾

Advanced Usage with Scripted Pipeline

Running "sidecar" containers

Using Docker in Pipeline can be an effective way to run a service on which the build, or a set of tests, may rely. Similar to the sidecar pattern, Docker Pipeline can run one container "in the background", while performing work in another. Utilizing this sidecar approach, a Pipeline can have a "clean" container provisioned for each Pipeline run.

Consider a hypothetical integration test suite which relies on a local MySQL database to be running. Using the `withRun` method, implemented in the plugin:docker-workflow[Docker Pipeline] plugin's support for Scripted Pipeline, a `Jenkinsfile` can run MySQL as a sidecar:

```
node {
    checkout scm
    /*
     * In order to communicate with the MySQL server, this Pipeline explicitly
     * maps the port (`3306`) to a known port on the host machine.
     */
    docker.image('mysql:5').withRun('-e "MYSQL_ROOT_PASSWORD=my-secret-pw" -p
3306:3306') { c ->
        /* Wait until mysql service is up */
        sh 'while ! mysqladmin ping -h0.0.0.0 --silent; do sleep 1; done'
        /* Run some tests which require MySQL */
        sh 'make check'
    }
}
```

This example can be taken further, utilizing two containers simultaneously. One "sidecar" running MySQL, and another providing the execution environment, by using the Docker container links.

```
node {
    checkout scm
    docker.image('mysql:5').withRun('-e "MYSQL_ROOT_PASSWORD=my-secret-pw"') { c ->
        docker.image('mysql:5').inside("--link ${c.id}:db") {
            /* Wait until mysql service is up */
            sh 'while ! mysqladmin ping -hdb --silent; do sleep 1; done'
        }
        docker.image('centos:7').inside("--link ${c.id}:db") {
            /*
             * Run some tests which require MySQL, and assume that it is
             * available on the host name `db`
             */
            sh 'make check'
        }
    }
}
```

The above example uses the object exposed by `withRun`, which has the running container's ID available via the `id` property. Using the container's ID, the Pipeline can create a link by passing custom Docker arguments to the `inside()` method.

The `id` property can also be useful for inspecting logs from a running Docker container before the Pipeline exits:

```
sh "docker logs ${c.id}"
```

Building containers

In order to create a Docker image, the plugin:docker-workflow[Docker Pipeline] plugin also provides a `build()` method for creating a new image, from a `Dockerfile` in the repository, during a Pipeline run.

One major benefit of using the syntax `docker.build("my-image-name")` is that a Scripted Pipeline can use the return value for subsequent Docker Pipeline calls, for example:

```
node {
    checkout scm

    def customImage = docker.build("my-image:${env.BUILD_ID}")

    customImage.inside {
        sh 'make test'
    }
}
```

The return value can also be used to publish the Docker image to Docker Hub, or a custom Registry, via the `push()` method, for example:

```
node {
    checkout scm
    def customImage = docker.build("my-image:${env.BUILD_ID}")
    customImage.push()
}
```

One common usage of image "tags" is to specify a `latest` tag for the most recently, validated, version of a Docker image. The `push()` method accepts an optional `tag` parameter, allowing the Pipeline to push the `customImage` with different tags, for example:

```
node {
    checkout scm
    def customImage = docker.build("my-image:${env.BUILD_ID}")
    customImage.push()

    customImage.push('latest')
}
```

The `build()` method builds the `Dockerfile` in the current directory by default. This can be overridden by providing a directory path containing a `Dockerfile` as the second argument of the `build()` method, for example:

```
node {
    checkout scm
    def testImage = docker.build("test-image", "./dockerfiles/test") ①

    testImage.inside {
        sh 'make test'
    }
}
```

① Builds `test-image` from the Dockerfile found at `./dockerfiles/test/Dockerfile`.

It is possible to pass other arguments to docker build by adding them to the second argument of the `build()` method. When passing arguments this way, the last value in the that string must be the path to the docker file.

This example overrides the default `Dockerfile` by passing the `-f` flag:

```
node {
    checkout scm
    def dockerfile = 'Dockerfile.test'
    def customImage = docker.build("my-image:${env.BUILD_ID}", "-f ${dockerfile}
./dockerfiles") ①
}
```

① Builds `my-image:${env.BUILD_ID}` from the Dockerfile found at `./dockerfiles/Dockerfile.test`.

Using a remote Docker server

By default, the plugin:docker-workflow[Docker Pipeline] plugin will communicate with a local Docker daemon, typically accessed through `/var/run/docker.sock`.

To select a non-default Docker server, such as with Docker Swarm, the `withServer()` method should be used.

By passing a URI, and optionally the Credentials ID of a **Docker Server Certificate Authentication** pre-configured in Jenkins, to the method with:

```
node {
    checkout scm

    docker.withServer('tcp://swarm.example.com:2376', 'swarm-certs') {
        docker.image('mysql:5').withRun('-p 3306:3306') {
            /* do things */
        }
    }
}
```

CAUTION

`inside()` and `build()` will not work properly with a Docker Swarm server out of the box

For `inside()` to work, the Docker server and the Jenkins agent must use the same filesystem, so that the workspace can be mounted.

Currently neither the Jenkins plugin nor the Docker CLI will automatically detect the case that the server is running remotely; a typical symptom would be errors from nested `sh` commands such as

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

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

Additionally some versions of Docker Swarm do not support custom Registries.

Using a custom registry

By default the plugin:docker-workflow[Docker Pipeline] integrates assumes the default Docker Registry of Docker Hub.

In order to use a custom Docker Registry, users of Scripted Pipeline can wrap steps with the `withRegistry()` method, passing in the custom Registry URL, for example:

```
node {
    checkout scm

    docker.withRegistry('https://registry.example.com') {

        docker.image('my-custom-image').inside {
            sh 'make test'
        }
    }
}
```

For a Docker Registry which requires authentication, add a "Username/Password" Credentials item from the Jenkins home page and use the Credentials ID as a second argument to `withRegistry()`:

```
node {
    checkout scm

    docker.withRegistry('https://registry.example.com', 'credentials-id') {

        def customImage = docker.build("my-image:${env.BUILD_ID}")

        /* Push the container to the custom Registry */
        customImage.push()
    }
}
```

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

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

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

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

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

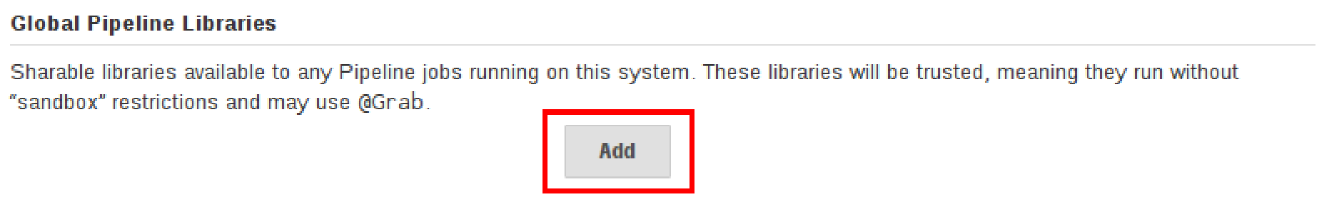
The Groovy source files in these directories get the same "CPS transformation" as in Scripted Pipeline.

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.



Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

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.

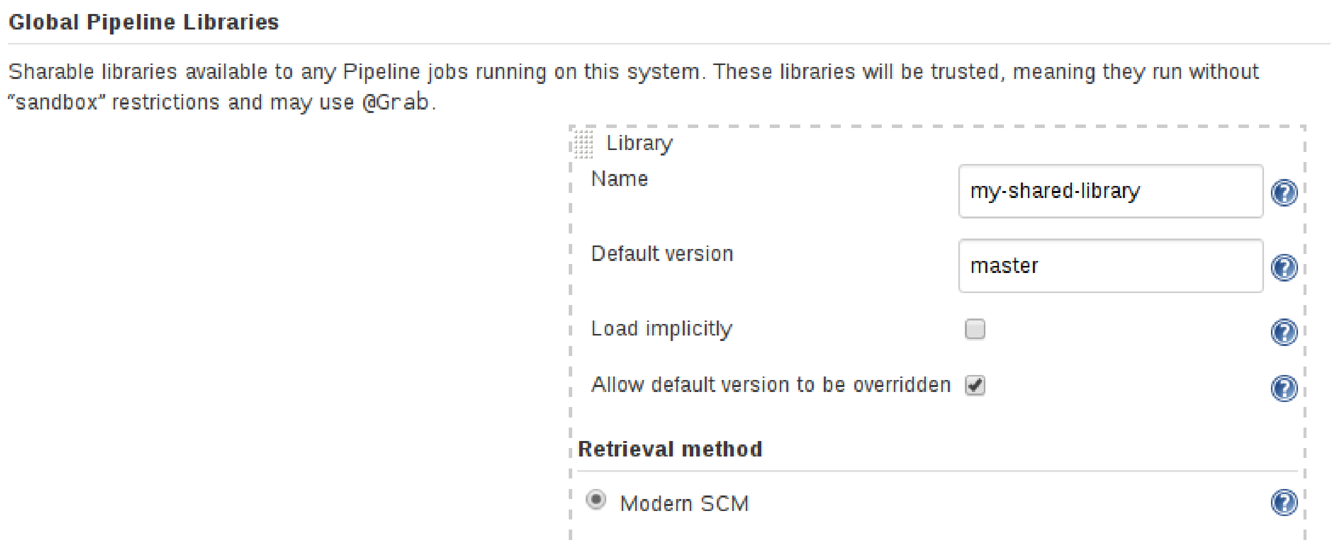
Folder-based libraries are not considered "trusted:" they run in the Groovy sandbox just like typical Pipelines.

Automatic Shared Libraries

Other plugins may add ways of defining libraries on the fly. For example, the plugin:github-branch-source[GitHub Branch Source] plugin provides a "GitHub Organization Folder" item which 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:


Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library

Name	my-shared-library
Default version	master
Load implicitly	☐
Allow default version to be overridden	☑

Retrieval method

● Modern SCM

```
@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 (`vars/`), or a `Jenkinsfile` which only needs a Global Variable, the annotation pattern `@Library('my-shared-library') _` may be useful for keeping code concise. In essence, instead of annotating an unnecessary `import` statement, the symbol `_` 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"
```

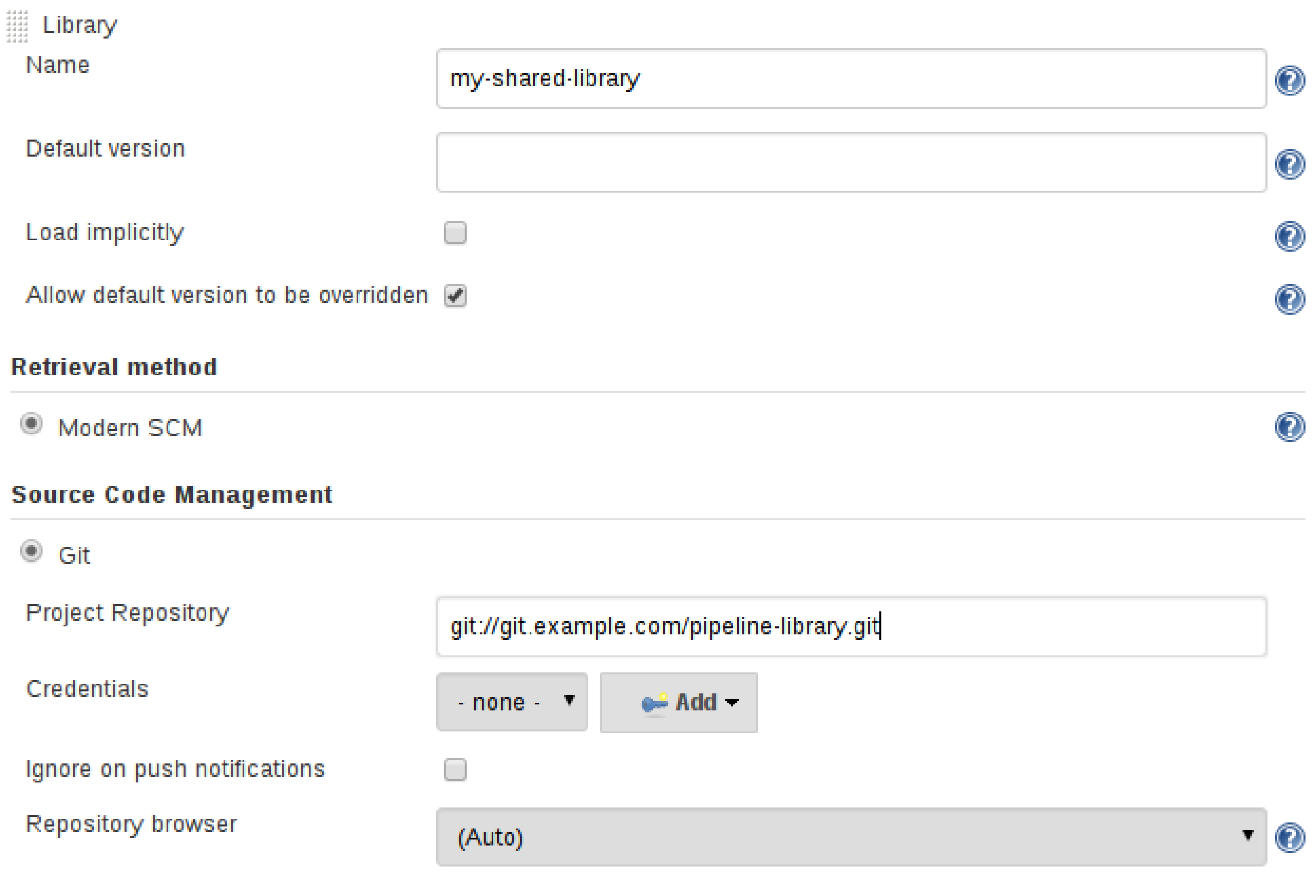
would load a library using the same SCM branch as the multibranch `Jenkinsfile`. As another example, you could pick a library by parameter:

```
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.


Library

Name	my-shared-library
Default version	
Load implicitly	☐
Allow default version to be overridden	✓

Retrieval method

● Modern SCM

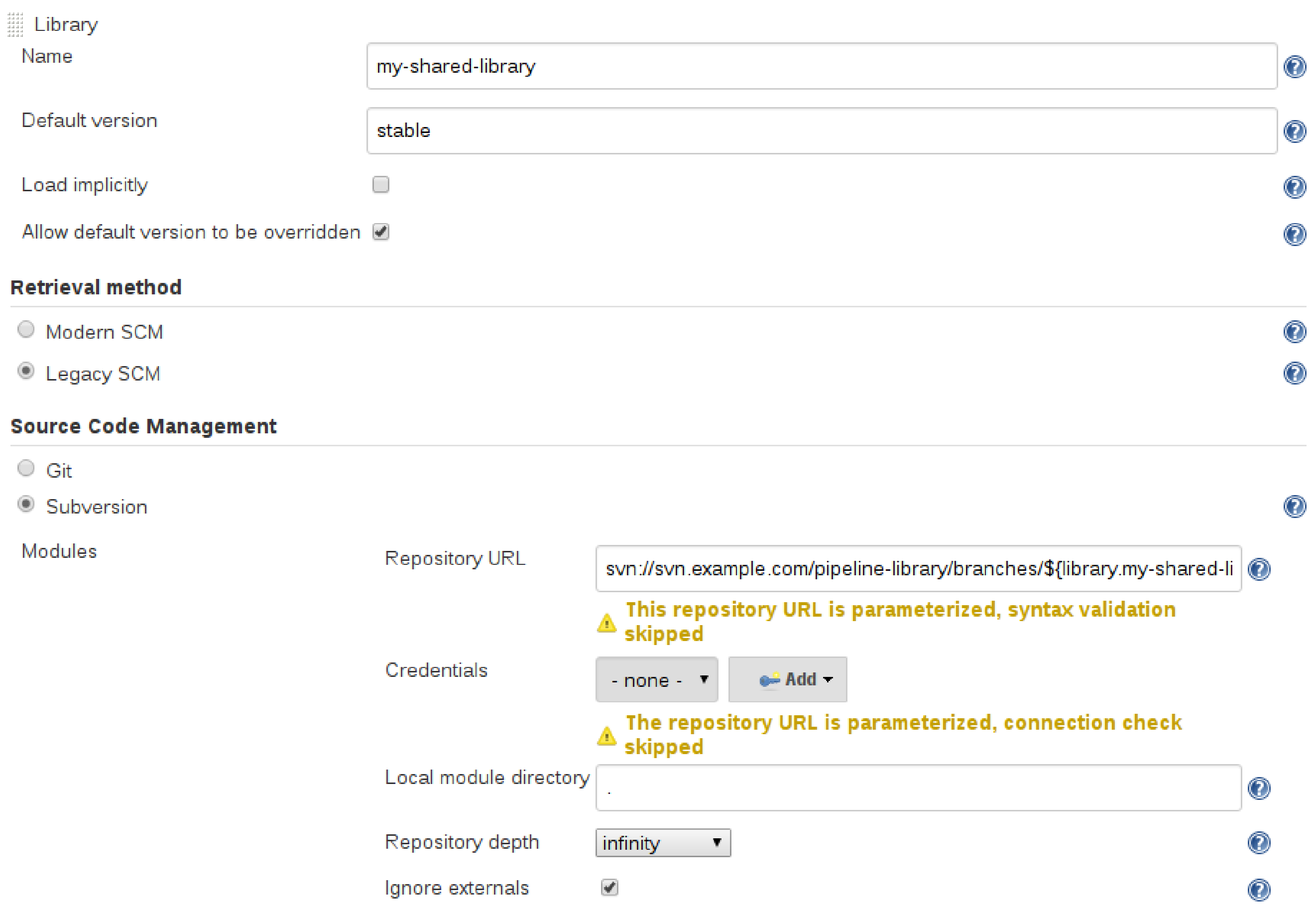
Source Code Management

● Git

Project Repository	git://git.example.com/pipeline-library.git
Credentials	- none -
Ignore on push notifications	☐
Repository browser	(Auto)

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.



Library Name	my-shared-library
Default version	stable
Load implicitly	☐
Allow default version to be overridden	✓

Retrieval method

Modern SCM

[✓ Legacy SCM]

Source Code Management

Git

[✓ Subversion]

Modules	Repository URL	svn://svn.example.com/pipeline-library/branches/${library.my-shared-li
		This repository URL is parameterized, syntax validation skipped
	Credentials	- none - Add
		The repository URL is parameterized, connection check skipped
	Local module directory	.
	Repository depth	infinity
	Ignore externals	✓

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}"
}

return this
```

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 using `this` 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(this)
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 to be defined in a single `.groovy` file for convenience. For example:

vars/log.groovy

```
def info(message) {
    echo "INFO: ${message}"
}

def warning(message) {
    echo "WARNING: ${message}"
}
```

Jenkinsfile

```
@Library('utils') _

log.info 'Starting'
log.warning 'Nothing to do!'
```

Declarative Pipeline does not allow global variable usage outside of a `script` directive (JENKINS-42360).

Jenkinsfile

```
@Library('utils') _

pipeline {
    agent none
    stage ('Example') {
        steps {
             script { ①
                 log.info 'Starting'
                 log.warning 'Nothing to do!'
             }
        }
    }
}
```

① `script` directive required to access global variables in Declarative Pipeline.

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.

WARNING *Avoid preserving state in global variables*

Avoid defining global variables with methods that interact or preserve state. Use a static class or instantiate a local variable of a class instead.

Defining custom 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(Map config) {
    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 //
```

There is also a “builder pattern” trick using Groovy’s `Closure.DELEGATE_FIRST`, which permits `Jenkinsfile` to look slightly more like a configuration file than a program, but this is more complex and error-prone and is not recommended.

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*.

Defining Declarative Pipelines

Starting with Declarative 1.2, released in late September, 2017, you can define Declarative Pipelines in your shared libraries as well. Here's an example, which will execute a different Declarative Pipeline depending on whether the build number is odd or even:

```
// vars/evenOrOdd.groovy
def call(int buildNumber) {
  if (buildNumber % 2 == 0) {
    pipeline {
      agent any
      stages {
        stage('Even Stage') {
          steps {
            echo "The build number is even"
          }
        }
      }
    }
  } else {
    pipeline {
      agent any
      stages {
        stage('Odd Stage') {
          steps {
            echo "The build number is odd"
          }
        }
      }
    }
  }
}
```

```
// Jenkinsfile
@Library('my-shared-library') _

evenOrOdd(currentBuild.getNumber())
```

Only entire `pipeline`s can be defined in shared libraries as of this time. This can only be done in `vars/*.groovy`, and only in a `call` method. Only one Declarative Pipeline can be executed in a single build, and if you attempt to execute a second one, your build will fail as a result.

Pipeline Development Tools

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

Blue Ocean Editor

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

Command-line Pipeline Linter

Jenkins can validate, or "lint", a Declarative Pipeline from the command line before actually running it. This can be done using a Jenkins CLI command or by making an HTTP POST request with appropriate parameters. We recommended using the SSH interface to run the linter. See the Jenkins CLI documentation for details on how to properly configure Jenkins for secure command-line access.

Linting via the CLI with SSH

```
# ssh (Jenkins CLI)
# JENKINS_SSHD_PORT=[sshd port on master]
# JENKINS_HOSTNAME=[Jenkins master hostname]
ssh -p $JENKINS_SSHD_PORT $JENKINS_HOSTNAME declarative-linter < Jenkinsfile
```

Linting via HTTP POST using `curl`

```
# curl (REST API)
# Assuming "anonymous read access" has been enabled on your Jenkins instance.
# JENKINS_URL=[root URL of Jenkins master]
# JENKINS_CRUMB is needed if your Jenkins master has CRSF protection enabled as it
should
JENKINS_CRUMB=`curl
"$JENKINS_URL/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,\":\",//crumb)"`
curl -X POST -H $JENKINS_CRUMB -F "jenkinsfile=<Jenkinsfile" $JENKINS_URL/pipeline-
model-converter/validate
```

Examples

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

Jenkinsfile

```
pipeline {
  agent
  stages {
    stage ('Initialize') {
      steps {
        echo 'Placeholder.'
      }
    }
  }
}
```

Linter output for invalid Jenkinsfile

```
# pass a Jenkinsfile that does not contain an "agent" section
ssh -p 8675 localhost declarative-linter < ./Jenkinsfile
Errors encountered validating Jenkinsfile:
WorkflowScript: 2: Not a valid section definition: "agent". Some extra configuration
is required. @ line 2, column 3.
     agent
     ^

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

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

Jenkinsfile

```
pipeline {
  agent any
  stages {
    stage ('Initialize') {
      steps {
        echo 'Placeholder.'
      }
    }
  }
}
```

Linter output for valid Jenkinsfile

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

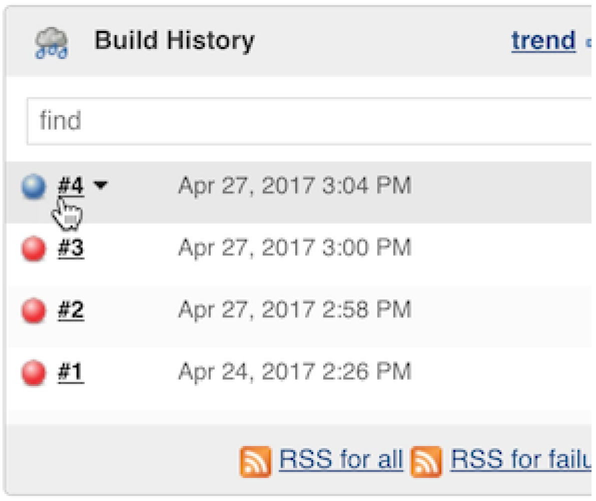
"Replay" Pipeline Runs with Modifications

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

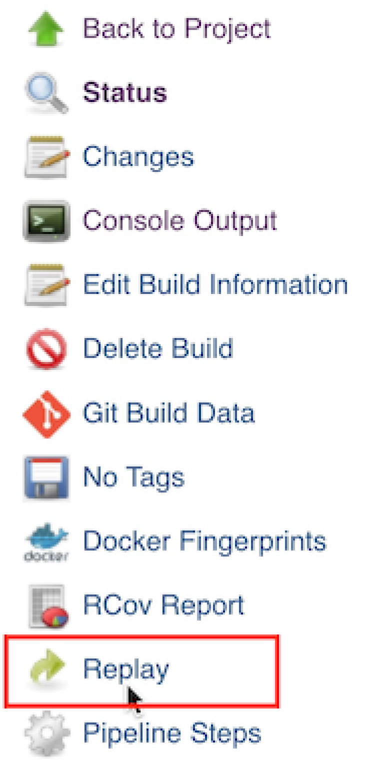
Usage

To use the "Replay" feature:

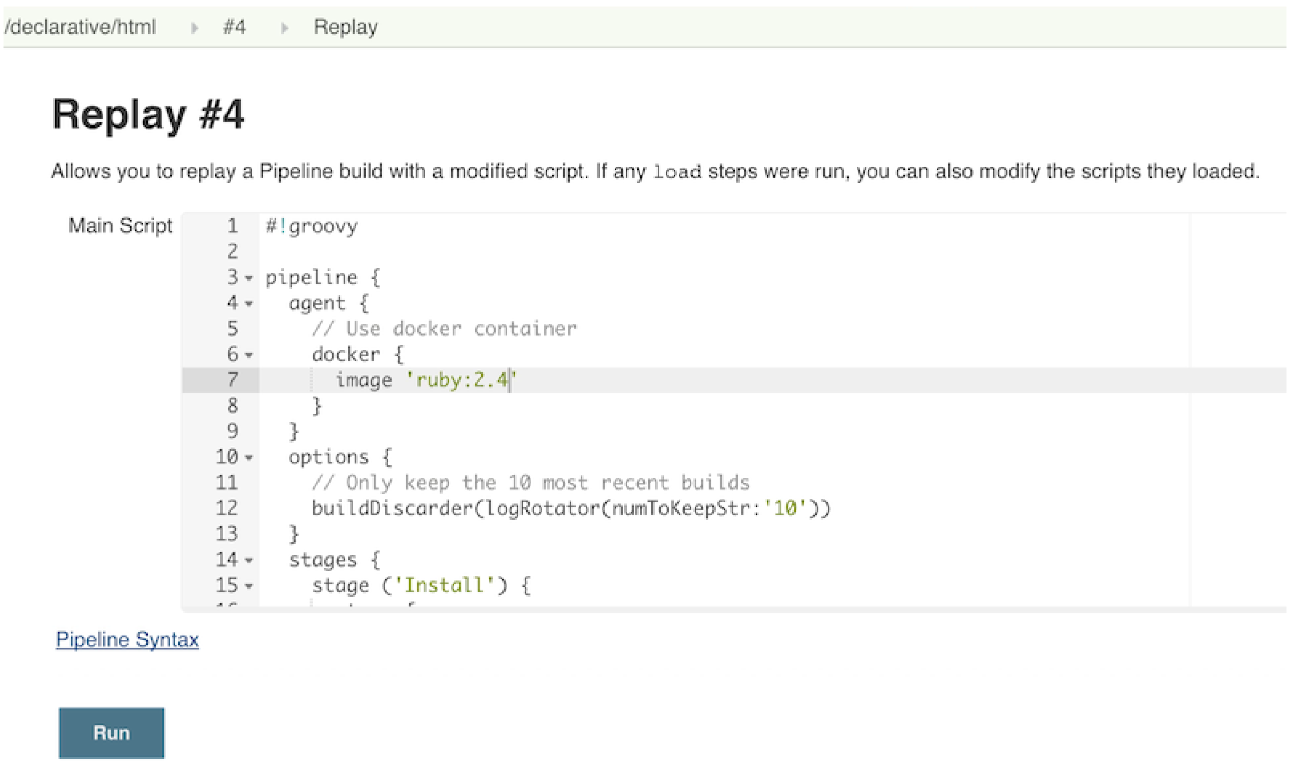
1. Select a previously completed run in the build history.



2. Click "Replay" in the left menu



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



Replay #4

Allows you to replay a Pipeline build with a modified script. If any `load` steps were run, you can also modify the scripts they loaded.

Main Script

```
#!groovy

pipeline {
  agent {
    // Use docker container
    docker {
      image 'ruby:2.4'
    }
  }
  options {
    // Only keep the 10 most recent builds
    buildDiscarder(logRotator(numToKeepStr:'10'))
  }
  stages {
    stage ('Install') {
```

4. Check the results of changes

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

Features

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

Limitations

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

IDE Integrations

Eclipse Jenkins Editor

The `Jenkins Editor` Eclipse plugin can be found on Eclipse Marketplace. This special text editor provides some features for defining pipelines e.g:

- Validate pipeline scripts by Jenkins Linter Validation. Failures are shown as eclipse markers
- An Outline with dedicated icons (for declarative Jenkins pipelines)
- Syntax / keyword highlighting
- Groovy validation

NOTE The Jenkins Editor Plugin is a third-party tool that is not supported by the Jenkins Project.

VisualStudio Code Jenkins Pipeline Linter Connector

The `Jenkins Pipeline Linter Connector` extension for VisualStudio Code takes the file that you have currently opened, pushes it to your Jenkins Server and displays the validation result in VS Code.

You can find the extension from within the VS Code extension browser or at the following url: marketplace.visualstudio.com/items?itemName=janjoerke.jenkins-pipeline-linter-connector

The extension adds four settings entries to VS Code which select the Jenkins server you want to use for validation.

- `jenkins.pipeline.linter.connector.url` is the endpoint at which your Jenkins Server expects the POST request, containing your Jenkinsfile which you want to validate. Typically this points to *<your_jenkins_server:port>/pipeline-model-converter/validate*.
- `jenkins.pipeline.linter.connector.user` allows you to specify your Jenkins username.
- `jenkins.pipeline.linter.connector.pass` allows you to specify your Jenkins password.
- `jenkins.pipeline.linter.connector.crumbUrl` has to be specified if your Jenkins Server has CRSF protection enabled. Typically this points to *<your_jenkins_server:port>/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,%22:%22,//crumb)*.

Atom linter-jenkins package

The linter-jenkins Atom package allows you to validate a Jenkins file by using the Pipeline Linter API of a running Jenkins. You can install it directly from the Atom package manager. It needs also to install Jenkinsfile language support in Atom

Pipeline Unit Testing Framework

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

The Pipeline Unit Testing Framework allows you to unit test Pipelines and Shared Libraries before running them in full. It provides a mock execution environment where real Pipeline steps are replaced with mock objects that you can use to check for expected behavior. New and rough around the edges, but promising. The README for that project contains examples and usage instructions.

Pipeline Syntax

This section builds on the information introduced in Getting started with Pipeline and should be treated solely as a reference. For more information on how to use Pipeline syntax in practical examples, refer to the Using a 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 at the start of this chapter, 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()

You can use the Declarative Directive Generator to help you get started with configuring the directives and sections in your Declarative Pipeline.

Sections

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

agent

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

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` section supports a few different types of parameters. These parameters can be applied at the top-level of the `pipeline` block, or within each `stage` directive.

any

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

none

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

label

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

node

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

docker

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

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

`docker` also optionally accepts a `registryUrl` and `registryCredentialsId` parameters which will help to specify the Docker Registry to use and its credentials. For example:

```
agent {
    docker {
        image 'myregistry.com/node'
        label 'my-defined-label'
        registryUrl 'https://myregistry.com/'
        registryCredentialsId 'myPredefinedCredentialsInJenkins'
    }
}
```

dockerfile

Execute the Pipeline, or stage, with a container built from a `Dockerfile` contained in the source repository. In order to use this option, the `Jenkinsfile` must be loaded from either a Multibranch Pipeline, or a "Pipeline from SCM." 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' } }`. If your `Dockerfile` has another name, you can specify the file name with the `filename` option. You can pass additional arguments to the `docker build …` command with the `additionalBuildArgs` option, like `agent { dockerfile { additionalBuildArgs '--build-arg foo=bar' } }`. For example, a repository with the file `build/Dockerfile.build`, expecting a build argument `version`:

```
agent {
    // Equivalent to "docker build -f Dockerfile.build --build-arg version=1.0.2
./build/
    dockerfile {
        filename 'Dockerfile.build'
        dir 'build'
        label 'my-defined-label'
        additionalBuildArgs  '--build-arg version=1.0.2'
        args '-v /tmp:/tmp'
    }
}
```

`dockerfile` also optionally accepts a `registryUrl` and `registryCredentialsId` parameters which will help to specify the Docker Registry to use and its credentials. For example:

```
agent {
    dockerfile {
        filename 'Dockerfile.build'
        dir 'build'
        label 'my-defined-label'
        registryUrl 'https://myregistry.com/'
        registryCredentialsId 'myPredefinedCredentialsInJenkins'
    }
}
```

Common Options

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

label

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

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

customWorkspace

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

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

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

reuseNode

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

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

args

A string. Runtime arguments to pass to `docker run`.

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

Example

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

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

Stage-level `agent` section

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

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

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

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

post

The `post` section defines one or more additional steps that are run upon the completion of a Pipeline's or stage's run (depending on the location of the `post` section within the Pipeline). `post` can support any of of the following post-condition blocks: `always`, `changed`, `fixed`, `regression`, `aborted`, `failure`, `success`, `unstable`, and `cleanup`. These condition blocks allow the execution of steps inside each condition depending on the completion status of the Pipeline or stage. The condition blocks are executed in the order shown below.

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

Conditions

`always`
: Run the steps in the `post` section regardless of the completion status of the Pipeline's or stage's run.

`changed`
: Only run the steps in `post` if the current Pipeline's or stage's run has a different completion status from its previous run.

`fixed`
: Only run the steps in `post` if the current Pipeline's or stage's run is successful and the previous run failed or was unstable.

`regression`
: Only run the steps in `post` if the current Pipeline's or stage's run's status is failure, unstable, or aborted and the previous run was successful.

`aborted`
: Only run the steps in `post` if the current Pipeline's or stage's run has an "aborted" status, usually due to the Pipeline being manually aborted. This is typically denoted by gray in the web UI.

`failure`
: Only run the steps in `post` if the current Pipeline's or stage's run has a "failed" status, typically denoted by red in the web UI. Note that if you manually set `currentBuild.result = 'FAILURE'` in a stage and have a `failure` `post` condition on that stage, the `failure` will not fire for that stage.

`success`
: Only run the steps in `post` if the current Pipeline's or stage's run has a "success" status, typically denoted by blue or green in the web UI.

`unstable`
: Only run the steps in `post` if the current Pipeline's or stage's run has an "unstable" status, usually caused by test failures, code violations, etc. This is typically denoted by yellow in the web UI.

`cleanup`
: Run the steps in this `post` condition after every other `post` condition has been evaluated, regardless of the Pipeline or stage's status.

Example

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

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

② Post-condition blocks contain steps the same as the [steps] section.

stages

Containing 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

The `steps` section defines a series of one or more 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

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` respectively.

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')) }`

checkoutToSubdirectory

Perform the automatic source control checkout in a subdirectory of the workspace. For example: `options { checkoutToSubdirectory('foo') }`

disableConcurrentBuilds

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

newContainerPerStage

Used with `docker` or `dockerfile` top-level agent. When specified, each stage will run in a new container instance on the same node, rather than all stages running in the same container instance.

overrideIndexTriggers

Allows overriding default treatment of branch indexing triggers. If branch indexing triggers are disabled at the multibranch or organization label, `options { overrideIndexTriggers(true) }` will enable them for this job only. Otherwise, `options { overrideIndexTriggers(false) }` will disable branch indexing triggers for this job only.

preserveStashes

Preserve stashes from completed builds, for use with stage restarting. For example: `options { preserveStashes() }` to preserve the stashes from the most recent completed build, or `options { preserveStashes(5) }` to preserve the stashes from the five most recent completed builds.

quietPeriod

Set the quiet period, in seconds, for the Pipeline, overriding the global default. For example: `options { quietPeriod(30) }`

retry

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

`retry(3) }`

skipDefaultCheckout

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

skipStagesAfterUnstable

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

timeout

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

timestamps

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

parallelsAlwaysFailFast

Set failfast true for all subsequent parallel stages in the pipeline. For example: `options { parallelsAlwaysFailFast() }`

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.

stage options

The `options` directive for a `stage` is similar to the `options` directive at the root of the Pipeline. However, the `stage`-level `options` can only contain steps like `retry`, `timeout`, or `timestamps`, or Declarative options that are relevant to a `stage`, like `skipDefaultCheckout`.

Inside a `stage`, the steps in the `options` directive are invoked before entering the `agent` or checking any `when` conditions.

Available Stage Options

skipDefaultCheckout

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

timeout

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

retry

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

timestamps

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

Example

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

① Specifying a execution timeout of one hour for the `Example` stage, after which Jenkins will abort the Pipeline run.

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: '') }`

text

A text parameter, which can contain multiple lines, for example: `parameters { text(name: 'DEPLOY_TEXT', defaultValue: 'One\nTwo\nThree\n', description: '') }`

booleanParam

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

choice

A choice parameter, for example: `parameters { choice(name: 'CHOICES', choices: ['one', 'two', 'three'], description: '') }`

file

A file parameter, which specifies a file to be submitted by the user when scheduling a build, for example: `parameters { file(name: 'FILE', description: 'Some file to upload') }`

password

A password parameter, for example: `parameters { password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password') }`

Example

```
// Declarative //
pipeline {
    agent any
    parameters {
        string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I
say hello to?')

        text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter some information
about the person')

        booleanParam(name: 'TOGGLE', defaultValue: true, description: 'Toggle this
value')

        choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'], description: 'Pick
something')

        password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'Enter a
password')

        file(name: "FILE", description: "Choose a file to upload")
    }
    stages {
        stage('Example') {
            steps {
                echo "Hello ${params.PERSON}"

                echo "Biography: ${params.BIOGRAPHY}"

                echo "Toggle: ${params.TOGGLE}"

                echo "Choice: ${params.CHOICE}"

                echo "Password: ${params.PASSWORD}"
            }
        }
    }
}
// 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. The triggers currently available are `cron`, `pollSCM` and `upstream`.

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 * * 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 * * 1-5') }`

upstream

Accepts a comma separated string of jobs and a threshold. When any job in the string finishes with the minimum threshold, the Pipeline will be re-triggered. For example: `triggers { upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS) }`

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

Example

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

Jenkins cron syntax

The Jenkins cron syntax follows the syntax of the cron utility (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

MINUTE	HOUR	DOM	MONTH	DOW
Minutes within the hour (0–59)	The hour of the day (0–23)	The day of the month (1–31)	The month (1–12)	The day of the week (0–7) where 0 and 7 are Sunday.

To specify multiple values for one field, the following operators are available. In the order of precedence,

- `*` specifies all valid values
- `M-N` specifies a range of values
- `M-N/X` or `*/X` steps by intervals of `X` through the specified range or whole valid range
- `A,B,⋯,Z` enumerates multiple values

To allow periodically scheduled tasks to produce even load on the system, the symbol `H` (for “hash”) should be used wherever possible. For example, using `0 0 * * *` for a dozen daily jobs will cause a large spike at midnight. In contrast, using `H H * * *` would still execute each job once a day, but not all at the same time, better using limited resources.

The `H` symbol can be used with a range. For example, `H H(0-7) * * *` means some time between 12:00 AM (midnight) to 7:59 AM. You can also use step intervals with `H`, with or without ranges.

The `H` symbol can be thought of as a random value over a range, but it actually is a hash of the job name, not a random function, so that the value remains stable for any given project.

Beware that for the day of month field, short cycles such as `/3 or H/3 will not work consistently near the end of most months, due to variable month lengths. For example, /3`j will run on the 1st, 4th, ⋯31st days of a long month, then again the next day of the next month. Hashes are always chosen in the 1-28 range, so `H/3` will produce a gap between runs of between 3 and 6 days at the end of a month. (Longer cycles will also have inconsistent lengths but the effect may be relatively less noticeable.)

Empty lines and lines that start with `#` will be ignored as comments.

In addition, `@yearly`, `@annually`, `@monthly`, `@weekly`, `@daily`, `@midnight`, and `@hourly` are supported as convenient aliases. These use the hash system for automatic balancing. For example, `@hourly` is the same as `H * * * *` and could mean at any time during the hour. `@midnight` actually means some time between 12:00 AM and 2:59 AM.

Table 1. Jenkins cron syntax examples

every fifteen minutes (perhaps at :07, :22, :37, :52)
`triggers{ cron('H/15 * * * *') }`
every ten minutes in the first half of every hour (three times, perhaps at :04, :14, :24)
`triggers{ H(0-29)/10 * * * *) }`
once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing at 3:45 PM every weekday.
`triggers{ 45 9-16/2 * * 1-5) }`

once in every two hours slot between 9 AM and 5 PM every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)
`triggers{ H H(9-16)/2 * * 1-5) }`
once a day on the 1st and 15th of every month except December
`triggers{ H H 1,15 1-11 *) }`

stage

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

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**.

input

The `input` directive on a `stage` allows you to prompt for input, using the `input` step. The `stage` will pause after any `options` have been applied, and before entering the `stage`s `agent` or evaluating its `when` condition. If the `input` is approved, the `stage` will then continue. Any parameters provided as part of the `input` submission will be available in the environment for the rest of the `stage`.

Configuration options

message

Required. This will be presented to the user when they go to submit the `input`.

id

An optional identifier for this `input`. Defaults to the `stage` name.

ok

Optional text for the "ok" button on the `input` form.

submitter

An optional comma-separated list of users or external group names who are allowed to submit this `input`. Defaults to allowing any user.

submitterParameter

An optional name of an environment variable to set with the `submitter` name, if present.

parameters

An optional list of parameters to prompt the submitter to provide. See [parameters] for more information.

Example

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example') {
            input {
                message "Should we continue?"
                ok "Yes, we should."
                submitter "alice,bob"
                parameters {
                    string(name: 'PERSON', defaultValue: 'Mr Jenkins', description:
'Who should I say hello to?')
                }
            }
            steps {
                echo "Hello, ${PERSON}, nice to meet you."
            }
        }
    }
}
// Script //
```

when

The `when` directive allows the Pipeline to determine whether the stage should be executed depending on the given condition. The `when` directive must contain at least one condition. If the `when` directive contains more than one condition, all the child conditions must return true for the stage to execute. This is the same as if the child conditions were nested in an `allOf` condition (see the examples below). If an `anyOf` condition is used, note that the condition skips remaining tests as soon as the first "true" condition is found.

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

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' }`. Note that this only works on a multibranch Pipeline.

buildingTag

Execute the stage when the build is building a tag. Example: `when { buildingTag() }`

changelog

Execute the stage if the build's SCM changelog contains a given regular expression pattern, for example: `when { changelog '.*^\\[DEPENDENCY\\] .+$' }`

changeset

Execute the stage if the build's SCM changeset contains one or more files matching the given string or glob. Example: `when { changeset "**/*.js" }`

By default the path matching will be case insensitive, this can be turned off with the `caseSensitive` parameter, for example: `when { changeset glob: "ReadMe.*", caseSensitive: true }`

changeRequest

Executes the stage if the current build is for a "change request" (a.k.a. Pull Request on GitHub and Bitbucket, Merge Request on GitLab or Change in Gerrit etc.). When no parameters are passed the stage runs on every change request, for example: `when { changeRequest() }`.

By adding a filter attribute with parameter to the change request, the stage can be made to run only on matching change requests. Possible attributes are `id`, `target`, `branch`, `fork`, `url`, `title`, `author`, `authorDisplayName`, and `authorEmail`. Each of these corresponds to a `CHANGE_*` environment variable, for example: `when { changeRequest target: 'master' }`.

The optional parameter `comparator` may be added after an attribute to specify how any patterns are evaluated for a match: `EQUALS` for a simple string comparison (the default), `GLOB` for an ANT style path glob (same as for example `changeset`), or `REGEXP` for regular expression matching. Example: `when { changeRequest authorEmail: "[\\w_-.]+@example.com", comparator: 'REGEXP' }`

environment

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

equals

Execute the stage when the expected value is equal to the actual value, for example: `when { equals expected: 2, actual: currentBuild.number }`

expression

Execute the stage when the specified Groovy expression evaluates to true, for example: `when {`

`expression { return params.DEBUG_BUILD } }` Note that when returning strings from your expressions they must be converted to booleans or return `null` to evaluate to false. Simply returning "0" or "false" will still evaluate to "true".

tag

Execute the stage if the `TAG_NAME` variable matches the given pattern. Example: `when { tag "release-*" }`. If an empty pattern is provided the stage will execute if the `TAG_NAME` variable exists (same as `buildingTag()`).

The optional parameter `comparator` may be added after an attribute to specify how any patterns are evaluated for a match: `EQUALS` for a simple string comparison, `GLOB` (the default) for an ANT style path glob (same as for example `changeset`), or `REGEXP` for regular expression matching. For example: `when { tag pattern: "release-\\d+", comparator: "REGEXP"}`

not

Execute the stage when the nested condition is false. Must contain one condition. For example: `when { not { branch 'master' } }`

allOf

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

anyOf

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

Evaluating `when` before entering the `stage’s `agent`

By default, the `when` condition for a `stage` will be evaluated after entering the `agent` for that `stage`, if one is defined. However, this can be changed by specifying the `beforeAgent` option within the `when` block. If `beforeAgent` is set to `true`, the `when` condition will be evaluated first, and the `agent` will only be entered if the `when` condition evaluates to true.

Evaluating `when` before the `input` directive

By default, the when condition for a stage will be evaluated before the input, if one is defined. However, this can be changed by specifying the `beforeInput` option within the when block. If `beforeInput` is set to true, the when condition will be evaluated first, and the input will only be entered if the when condition evaluates to true.

Examples

Single condition

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

Multiple condition

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

Nested condition (same behavior as previous example)

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                allOf {
                    branch 'production'
                    environment name: 'DEPLOY_TO', value: 'production'
                }
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
// Script //
```

Multiple condition and nested condition

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                branch 'production'
                anyOf {
                    environment name: 'DEPLOY_TO', value: 'production'
                    environment name: 'DEPLOY_TO', value: 'staging'
                }
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
// Script //
```

Expression condition and nested condition

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                expression { BRANCH_NAME ==~ /(production|staging)/ }
                anyOf {
                    environment name: 'DEPLOY_TO', value: 'production'
                    environment name: 'DEPLOY_TO', value: 'staging'
                }
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
// Script //
```

beforeAgent

```
// Declarative //
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            agent {
                label "some-label"
            }
            when {
                beforeAgent true
                branch 'production'
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
// Script //
```

`beforeInput`

```
// Declarative //
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                beforeInput true
                branch 'production'
            }
            input {
                message "Deploy to production?"
                id "simple-input"
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
// Script //
```

Sequential Stages

Stages in Declarative Pipeline may declare a list of nested stages to be run within them in sequential order. Note that a stage must have one and only one of `steps`, `parallel`, or `stages`, the last for sequential stages. The stages within `stages` in a stage cannot contain further `parallel` or `stages` themselves, but they do allow use of all other functionality of a stage, including `agent`, `tools, `when`, etc.

Example

```
// Declarative //
pipeline {
    agent none
    stages {
        stage('Non-Sequential Stage') {
            agent {
                label 'for-non-sequential'
            }
            steps {
                echo "On Non-Sequential Stage"
            }
        }
        stage('Sequential') {
            agent {
                label 'for-sequential'
            }
            environment {
                FOR_SEQUENTIAL = "some-value"
            }
            stages {
               stage('In Sequential 1') {
                   steps {
                       echo "In Sequential 1"
                   }
               }
               stage('In Sequential 2') {
                   steps {
                       echo "In Sequential 2"
                   }
               }
            }
        }
    }
}
// Script //
```

Parallel

Stages in Declarative Pipeline may declare a number of nested stages within a `parallel` block, which will be executed in parallel. Note that a stage must have one and only one of `steps`, `stages`, or `parallel`. The nested stages cannot contain further `parallel` stages themselves, but otherwise behave the same as any other `stage`, including a list of sequential stages within `stages`. Any stage containing `parallel` cannot contain `agent` or `tools`, since those are not relevant without `steps`.

In addition, you can force your `parallel` stages to all be aborted when one of them fails, by adding `failFast true` to the `stage` containing the `parallel`. Another option for adding `failfast` is adding an option to the pipeline definition: `parallelsAlwaysFailFast()`

Example

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Non-Parallel Stage') {
            steps {
                echo 'This stage will be executed first.'
            }
        }
        stage('Parallel Stage') {
            when {
                branch 'master'
            }
            failFast true
            parallel {
                stage('Branch A') {
                    agent {
                        label "for-branch-a"
                    }
                    steps {
                        echo "On Branch A"
                    }
                }
                stage('Branch B') {
                    agent {
                        label "for-branch-b"
                    }
                    steps {
                        echo "On Branch B"
                    }
                }
                stage('Branch C') {
                    agent {
                        label "for-branch-c"
                    }
                    stages {
                        stage('Nested 1') {
                            steps {
                                echo "In stage Nested 1 within Branch C"
                            }
                        }
                        stage('Nested 2') {
                            steps {
                                echo "In stage Nested 2 within Branch C"
                            }
                        }
                    }
                }
            }
        }
    }
}

// Script //
```

`parallelsAlwaysFailFast`

```
// Declarative //
pipeline {
    agent any
    options {
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Non-Parallel Stage') {
            steps {
                echo 'This stage will be executed first.'
            }
        }
        stage('Parallel Stage') {
            when {
                branch 'master'
            }
            parallel {
                stage('Branch A') {
                    agent {
                        label "for-branch-a"
                    }
                    steps {
                        echo "On Branch A"
                    }
                }
                stage('Branch B') {
                    agent {
                        label "for-branch-b"
                    }
                    steps {
                        echo "On Branch B"
                    }
                }
                stage('Branch C') {
                    agent {
                        label "for-branch-c"
                    }
                    stages {
                        stage('Nested 1') {
                            steps {
                                echo "In stage Nested 1 within Branch C"
                            }
                        }
                        stage('Nested 2') {
                            steps {
                                echo "In stage Nested 2 within Branch C"
                            }
                        }
                    }
                }
            }
        }
    }
}// 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 at the start of this chapter, 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 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. [1: 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.

Both are 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 to 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 encourages a declarative programming model. [21: Declarative Programming] Whereas Scripted Pipelines follow a more imperative programming model. [22: Imperative Programming]

Using Speed/Durability Settings To Reduce Disk I/O Needs

One of the main bottlenecks in Pipeline is that it writes transient data to disk **FREQUENTLY** so that running pipelines can handle an unexpected Jenkins restart or system crash. This durability is useful for many users but its performance cost can be a problem.

Pipeline now includes features to let users improve performance by reducing how much data is written to disk and how often it is written — at a small cost to durability. In some special cases, users may not be able to resume or visualize running Pipelines if Jenkins shuts down suddenly without getting a chance to write data.

Because these settings include a trade-off of speed vs. durability, they are initially opt-in. To enable performance-optimized modes, users need to explicity set a *Speed/Durability Setting* for Pipelines. If no explicit choice is made, pipelines currently default to the "maximum durability" setting and write to disk as they have in the past. There are some I/O optimizations to this mode included in the same plugin releases, but the benefits are much smaller.

How Do I Set Speed/Durability Settings?

There are 3 ways to configure the durability setting:

1. **Globally**, you can choose a global default durability setting under "Manage Jenkins" > "Configure System", labelled "Pipeline Speed/Durability Settings". You can override these with the more specific settings below.
2. **Per pipeline job:** at the top of the job configuration, labelled "Custom Pipeline Speed/Durability Level" - this overrides the global setting. Or, use a "properties" step - the setting will apply to the NEXT run after the step is executed (same result).
3. **Per-branch for a multibranch project:** configure a custom Branch Property Strategy (under the SCM) and add a property for Custom Pipeline Speed/Durability Level. This overrides the global setting. You can also use a "properties" step to override the setting, but remember that you may have to run the step again to undo this.

Durability settings will take effect with the next applicable Pipeline run, not immediately. The setting will be displayed in the log.

Will Higher-Performance Durability Settings Help Me?

- Yes, if your Jenkins instance uses NFS, magnetic storage, runs many Pipelines at once, or shows high iowait.
- Yes, if you are running Pipelines with many steps (more than several hundred).
- Yes, if your Pipeline stores large files or complex data to variables in the script, keeps that variable in scope for future use, and then runs steps. This sounds oddly specific but happens more than you'd expect.
    - For example: `readFile` step with a large XML/JSON file, or using configuration information

from parsing such a file with One of the Utility Steps.

  - Another common pattern is a "summary" object containing data from many branches (logs, results, or statistics). Often this is visible because you'll be adding to it often via an add/append or `Map.put()` operations.
  - Large arrays of data or `Map`s of configuration information are another common example of this situation.

- No, if your Pipelines spend almost all their time waiting for a few shell/batch scripts to finish. This ISN'T a magic "go fast" button for everything!
- No, if Pipelines are writing massive amounts of data to logs (logging is unchanged).
- No, if you are not using Pipelines, or your system is loaded down by other factors.
- No, if you don't enable higher-performance modes for pipelines.

What Am I Giving Up With This Durability Setting "Trade-Off?"

Stability of Jenkins ITSELF is not changed regardless of this setting - it only applies to Pipelines. The worst-case behavior for Pipelines reverts to something like Freestyle builds — running Pipelines that cannot persist transient data may not be able to resume or be displayed in Blue Ocean/Stage View/etc, but will show logs. This impacts *only* running Pipelines and only when Jenkins is shut down abruptly and not gracefully before they get to complete.

A **"graceful" shutdown** is where Jenkins goes through a full shutdown process, such as visiting http://[jenkins-server]/exit, or using normal service shutdown scripts (if Jenkins is healthy). Sending a SIGTERM/SIGINT to Jenkins will trigger a graceful shutdown. Note that running Pipelines do not need to complete (you do not need to use /safeExit to shut down).

A **"dirty" shutdown** is when Jenkins does not get to do normal shutdown processes. This can occur if the process is forcibly terminated. The most common causes are using SIGKILL to terminate the Jenkins process or killing the container/VM running Jenkins. Simply stopping or pausing the container/VM will not cause this, as long as the Jenkins process is able to resume. A dirty shutdown can also happen due to catastrophic operating system failures, including the Linux OOMKiller attacking the Jenkins java process to free memory.

Atomic writes: All settings **except** "maximum durability" currently avoid atomic writes — what this means is that if the operating system running Jenkins fails, data that is buffered for writing to disk will not be flushed, it will be lost. This is quite rare, but can happen as a result of container or virtualization operations that halt the operating system or disconnect storage. Usually this data is flushed pretty quickly to disk, so the window for data loss is brief. On Linux this flush-to-disk can be forced by running 'sync'. In some rare cases this can also result in a build that cannot be loaded.

Requirements To Use Durability Settings

- Jenkins LTS 2.73+ or higher (or a weekly 2.62+)
- For **all** the Pipeline plugins below, at least the specified minimum version must be installed

    - Pipeline: API (workflow-api) v2.25
    - Pipeline: Groovy (workflow-cps) v2.43
    - Pipeline: Job (workflow-job) v2.17
    - Pipeline: Supporting APIs (workflow-support) v2.17
    - Pipeline: Multibranch (workflow-multibranch) v2.17 - optional, only needed to enable this setting for multibranch pipelines.
- Restart the master to use the updated plugins - note: you need all of them to take advantage.

What Are The Durability Settings?

- Performance-optimized mode ("PERFORMANCE_OPTIMIZED") - **Greatly** reduces disk I/O. If Pipelines do not finish AND Jenkins is not shut down gracefully, they may lose data and behave like Freestyle projects — see details above.
- Maximum durability ("MAX_SURVIVABILITY") - behaves just like Pipeline did before, slowest option. Use this for running your most critical Pipelines.
- Less durable, a bit faster ("SURVIVABLE_NONATOMIC") - Writes data with every step but avoids atomic writes. This is faster than maximum durability mode, especially on networked filesystems. It carries a small extra risk (details above under "What Am I Giving Up: Atomic Writes").

Suggested Best Practices And Tips for Durability Settings

- Use the "performance-optimized" mode for most pipelines and especially basic build-test Pipelines or anything that can simply be run again if needed.
- Use either the "maximum durability" or "less durable" mode for pipelines when you need a guaranteed record of their execution (auditing). These two modes record every step run. For example, use one of these two modes when:
    - you have a pipeline that modifies the state of critical infrastructure
    - you do a production deployment
- Set a global default (see above) of "performance-optimized" for the Durability Setting, and then where needed set "maximum durability" on specific Pipeline jobs or Multibranch Pipeline branches ("master" or release branches).
- You can force a Pipeline to persist data by pausing it.

Other Scaling Suggestions

- Use @NonCPS-annotated functions for more complex work. This means more involved processing, logic, and transformations. This lets you leverage additional Groovy & functional features for more powerful, concise, and performant code.
    - This still runs on masters so be aware of complexity of the work, but is much faster than

native Pipeline code because it doesn't provide durability and uses a faster execution model. Still, be mindful of the CPU cost and offload to executors when the cost becomes too high.

  - @NonCPS functions can use a much broader subset of the Groovy language, such as iterators and functional features, which makes them more terse and fast to write.
  - @NonCPS functions **should not use** Pipeline steps internally, however you can store the result of a Pipeline step to a variable and use it that as the input to a @NonCPS function.
    - **Gotcha**: It's not guaranteed that use of a step will generate an error (there is an open RFE to implement that), but you should not rely on that behavior. You may see improper handling of exceptions.
  - While normal Pipeline is restricted to serializable local variables, @NonCPS functions can use more complex, nonserializable types internally (for example regex matchers, etc). Parameters and return types should still be Serializable, however.
    - **Gotcha**: improper usages are not guaranteed to raise an error with normal Pipeline (optimizations may mask the issue), but it is unsafe to rely on this behavior.
  - **General Gotcha**: when using running @NonCPS functions, the actual error can sometimes be swallowed by pipeline creating a confusing error message. Combat this by using a `try/catch` block and potentially using an `echo` to plain text print the error message in the `catch`

- **Whenever possible, run Jenkins with fast SSD-backed storage and not hard drives. This can make a *huge* difference.**
- In general try to fit the tool to the job. Consider writing short Shell/Batch/Groovy/Python scripts when running a complex process using a build agent. Good examples include processing data, communicating interactively with REST APIs, and parsing/templating larger XML or JSON files. The `sh` and `bat` steps are helpful to invoke these, especially with `returnStdout: true` to return the output from this script and save it as a variable (Scripted Pipeline).
  - The Pipeline DSL is not designed for arbitrary networking and computation tasks - it is intended for CI/CD scripting.
- Use the latest versions of the Pipeline plugins and Script Security, if applicable. These include regular performance improvements.
- Try to simplify Pipeline code by reducing the number of steps run and using simpler Groovy code for Scripted Pipelines.
- Consolidate sequential steps of the same type if you can, for example by using one Shell step to invoke a helper script rather than running many steps.
- Try to limit the amount of data written to logs by Pipelines. If you are writing several MB of log data, such as from a build tool, consider instead writing this to an external file, compressing it, and archiving it as a build artifact.
- When using Jenkins with more than 6 GB of heap use the suggested garbage collection tuning options to minimize garbage collection pause times and overhead.

Blue Ocean

This chapter covers all aspects of Blue Ocean's functionality, including how to:

- get started with Blue Ocean - covers how to set up Blue Ocean in Jenkins and access the Blue Ocean interface,
- create a new Pipeline project,
- use Blue Ocean's Dashboard,
- use the Activity view - where you can access lists of your current and previously completed Pipeline/item runs, as well as your Pipeline project's branches and any opened Pull Requests,
- use the Pipeline run details view - where you can access details (i.e. the console output) for a particular Pipeline/item run, and
- use the Pipeline Editor to modify Pipelines as code, which are committed to source control.

This chapter is intended for Jenkins users of all skill levels, but beginners may need to refer to some sections of the Pipeline chapter to understand some topics covered in this Blue Ocean chapter.

For an overview of content in the Jenkins User Handbook, see User Handbook overview.

What is Blue Ocean?

Blue Ocean rethinks the user experience of Jenkins. Designed from the ground up for Jenkins Pipeline, but still compatible with freestyle jobs, Blue Ocean reduces clutter and increases clarity for every member of the team. Blue Ocean's main features include:

- **Sophisticated visualizations** of continuous delivery (CD) Pipelines, allowing for fast and intuitive comprehension of your Pipeline's status.
- **Pipeline editor** - makes creation of Pipelines approachable by guiding the user through an intuitive and visual process to create a Pipeline.
- **Personalization** to suit the role-based needs of each member of the team.
- **Pinpoint precision** when intervention is needed and/or issues arise. Blue Ocean 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.

To start using Blue Ocean, see Getting started with Blue Ocean.

Frequently asked questions

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 is 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 is 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 have become 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. Jenkins needs to rise to meet this higher standard.

Creating and visualising CD pipelines is something valuable for many Jenkins users and this is demonstrated in the 5+ plugins that the Jenkins community has created to meet their needs. 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 is not just CD concepts but the tools that developers use every day – Github, Bitbucket, Slack, HipChat, Puppet or Docker. It is about more than Jenkins – it is the developer workflow which surrounds Jenkins that spans multiple tools.

New teams have little time to learn how 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 changes and the mission of Blue Ocean enables 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".

Does Blue Ocean support freestyle jobs?

Blue Ocean aims to deliver a great experience around Pipeline and be compatible with any freestyle jobs you already have configured on your Jenkins instance. However, you will not benefit from any of the features built for Pipelines – for example, Pipeline visualization.

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

What does this mean for the Jenkins classic UI?

The intention is that as Blue Ocean matures, there will be fewer reasons for users to go back to the existing "classic UI". Read more about the classic UI in Getting started with Pipeline.

For example, early versions of Blue Ocean are mainly targeted at 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 Blue Ocean UI for some time. This means users will have to jump back to the classic UI to configure items/projects/jobs other than Pipeline ones.

There are likely going to be more examples of this, which is why the classic UI will remain important in the long term.

What does this mean for my plugins?

Extensibility is a core feature of Jenkins. Therefore, being able to extend the Blue Ocean UI is important. The
`<ExtensionPoint name=..>` can be used in the markup of Blue Ocean, leaving places for plugins to contribute to the Blue Ocean UI - i.e. plugins can have their own Blue Ocean extension points, just like they can in the Jenkins classic UI. So far, Blue Ocean itself is implemented using these extension points.

Extensions are delivered by plugins as usual. However, plugin developers will need to include some additional JavaScript to hook into Blue Ocean’s extension points and contribute to the Blue Ocean user experience.

What technologies are currently in use?

Blue Ocean is built as a collection of Jenkins plugins itself. There is one key difference - Blue Ocean 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 (read more about this in the Building Plugins for React Apps blog post), an `<ExtensionPoint>` pattern was established that allows extensions to come from any Jenkins plugin (only with JavaScript) and should they fail to load, have their failures isolated.

Where can I find the source code?

The source code can be found on Github:

- Blue Ocean
- Jenkins Design Language

Join the community

There a few ways you can join the community:

1. Chat with the community and development team on Gitter chat on gitter
2. Request features or report bugs against the `blueocean-plugin` component in JIRA.
3. Subscribe and ask questions on the Jenkins Users mailing list.
4. Developer? We’ve labeled a few issues that are great for anyone wanting to get started developing Blue Ocean. Don’t forget to drop by the Gitter chat and introduce yourself!

Getting started with Blue Ocean

This section describes how to get started with Blue Ocean in Jenkins. It includes instructions for setting up Blue Ocean on your Jenkins instance as well as how to access the Blue Ocean UI and return to the Jenkins classic UI.

Installing Blue Ocean

Blue Ocean can be installed using the following methods:

- As a suite of plugins on an existing Jenkins instance, or
- As part of Jenkins in Docker.

On an existing Jenkins instance

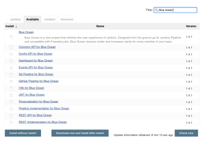
When Jenkins is installed on most platforms, the plugin:blueocean[Blue Ocean plugin] and all its other dependent plugins (which form the Blue Ocean "suite of plugins") are not installed by default.

To install the Blue Ocean suite of plugins on an existing Jenkins instance, your Jenkins instance must be running Jenkins 2.7.x or later.

Plugins can be installed on a Jenkins instance by any Jenkins user who has the **Administer** permission (set through **Matrix-based security**). Jenkins users with this permission can also configure the permissions of other users on their system. Read more about this in the Authorization section of Managing Security.

To install the Blue Ocean suite of plugins to your Jenkins instance:

1. If required, ensure you are logged in to Jenkins (as a user with the **Administer** permission).
2. From the Jenkins home page (i.e. the Dashboard of the Jenkins classic UI), click **Manage Jenkins** on the left and then **Manage Plugins** in the center.
3. Click the **Available** tab and type `blue ocean` into the **Filter** text box, which filters the list of plugins to those whose name and/or description contains the words "blue" and "ocean".



Install ↓	Name	Version
☐	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.	1.4.1
☐	Common API for Blue Ocean	1.4.1
☐	Config API for Blue Ocean	1.4.1
☐	Dashboard for Blue Ocean	1.4.1
☐	Events API for Blue Ocean	1.4.1
☐	Git Pipeline for Blue Ocean	1.4.1
☐	GitHub Pipeline for Blue Ocean	1.4.1
☐	i18n for Blue Ocean	1.4.1
☐	JWT for Blue Ocean	1.4.1
☐	Personalization for Blue Ocean	1.4.1
☐	Pipeline implementation for Blue Ocean	1.4.1
☐	REST API for Blue Ocean	1.4.1
☐	REST Implementation for Blue Ocean	1.4.1

Update information obtained: 8 min 13 sec ago

4. Select the **Blue Ocean** plugin's check box near the top of the the **Install** column and then click either the **Download now and install after restart** button (recommended) or the **Install without restart** button at the the end of the page.
   Notes:
   - There is no need to select the check boxes of the other plugins in this filtered list because the main **Blue Ocean** plugin has other plugin dependencies (constituting the Blue Ocean suite of plugins) which will automatically be selected and installed when you click one of these "Install" buttons.

    ◦ If you chose the **Install without restart** button, you may need to restart Jenkins in order to gain full Blue Ocean functionality.

Read more about how to install and manage plugins in the Managing Plugins page.

Blue Ocean requires no additional configuration after installation, and existing Pipelines projects and other items such as freestyle projects will continue to work as usual.

Be aware, however, that the first time a Pipeline is created in Blue Ocean for a specific Git server (i.e. GitHub, Bitbucket or an ordinary Git server), Blue Ocean prompts you for credentials to access your repositories on the Git server in order to create Pipelines based on those repositories. This is required since Blue Ocean can write `Jenkinsfile`s to your repositories.

As part of Jenkins in Docker

The Blue Ocean suite of plugins are also bundled with Jenkins as part of a Jenkins Docker image (`jenkinsci/blueocean`), which is available from the Docker Hub repository.

Read more about running Jenkins and Blue Ocean this way in the Docker section of the Installing Jenkins page.

Accessing Blue Ocean

Once a Jenkins environment has Blue Ocean installed, after logging in to the Jenkins classic UI, you can access the Blue Ocean UI by clicking **Open Blue Ocean** on the left.

Alternatively, you can access Blue Ocean directly by appending `/blue` to the end of your Jenkins server's URL - e.g. `http://jenkins-server-url/blue`.

If your Jenkins instance:

- already has existing Pipeline projects or other items present, then the Blue Ocean Dashboard is displayed.
- is new or has no Pipeline projects or other items configured, then Blue Ocean displays a **Welcome to Jenkins** box with a **Create a new Pipeline** button you can use to begin creating a new Pipeline project. Read more about this in Creating a Pipeline.

Navigation bar

The Blue Ocean UI has a navigation bar along the top of its interface, which allows you to access the different views and other features of Blue Ocean.

Th navigation bar is divided into two sections - a common section along the top of most Blue Ocean views and a contextual section below. The contextual section is specific to the current Blue Ocean page you are viewing.

The navigation bar’s common section includes the following buttons:

- **Jenkins** logo - takes you to the Dashboard, or reloads this page if you are already viewing it.
- **Pipelines** - also takes you to the Dashboard, or does nothing if you are already viewing the Dashboard. This button serves a different purpose when you are viewing a Pipeline run details page.
- **Administration** - takes you to the **Manage Jenkins** page of the Jenkins classic UI.
  Note: This button is not available if your Jenkins user does not have the **Administer** permission (set through **Matrix-based security**). Read more about this in the Authorization section of Managing Security.
- **Go to classic** icon - takes you back to the Jenkins classic UI. Read more about this in [switching-to-the-classic-ui].
- **Logout** - Logs out your current Jenkins user and returns to the Jenkins login page.

Views that use the standard navigation bar will add another bar below it with options specific to that view. Some views replace the common navigation bar with one specifically suited to that view.

Switching to the classic UI

Blue Ocean does not support some legacy or administrative features of Jenkins that are necessary to some users.

If you need to leave the Blue Ocean user experience to access these features, click the **Go to classic** icon at the top of common section of Blue Ocean’s navigation bar.

Clicking this button takes you to the equivalent page in the Jenkins classic UI, or the most relevant classic UI page that parallels the current page in Blue Ocean.

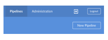
Creating a Pipeline

Blue Ocean makes it easy to create a Pipeline project in Jenkins.

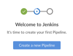
A Pipeline can be generated from an existing `Jenkinsfile` in source control, or you can use the Blue Ocean Pipeline editor to create a new Pipeline for you (as a `Jenkinsfile` that will be committed to source control).

Setting up your Pipeline project

To start setting up your Pipeline project in Blue Ocean, at the top-right of the Blue Ocean Dashboard, click the **New Pipeline** button.



If your Jenkins instance is new or has no Pipeline projects or other items configured (and the Dashboard is empty), Blue Ocean displays a **Welcome to Jenkins** message box on which you can click the **Create a new Pipeline** button to start setting up your Pipeline project.

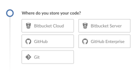


You now have a choice of creating your new Pipeline project from a:

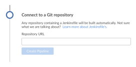
- standard Git repository
- repository on GitHub or GitHub Enterprise
- repository on Bitbucket Cloud or Bitbucket Server

For a Git repository

To create your Pipeline project for a Git repository, click the **Git** button under **Where do you store your code?**



In the **Connect to a Git repository** section, enter the URL for your Git repository in the **Repository URL** field.


Connect to a Git repository

Any repository containing a Jenkinsfile will be built automatically. Not sure what we are talking about? Learn more about Jenkinsfile's.

Repository URL

You now need to specify a local or a remote repository from which to build your Pipeline project.

Local repository

If your URL is a local directory path (e.g. beginning with a forward slash `/` such as `/home/cloned-git-repos/my-git-repo.git`), you can proceed to click the **Create Pipeline** button.

Blue Ocean will then scan your local repository's branches for a `Jenkinsfile` and will commence a Pipeline run for each branch containing a `Jenkinsfile`. If Blue Ocean cannot find any `Jenkinsfile`, you will be prompted to begin creating one through the Pipeline editor.

Remote repository

Since the Pipeline editor saves edited Pipelines to Git repositories as `Jenkinsfile`s, Blue Ocean only supports connections to remote Git repositories over the SSH protocol.

If your URL is for a remote Git repository, then as soon as you begin typing the URL, starting with either:

- `ssh://` - e.g. `ssh://gituser@git-server-url/git-server-repos-group/my-git-repo.git` or
- `user@host:path/to/git/repo.git` - e.g. `gituser@git-server-url:git-server-repos-group/my-git-repo.git`,

Blue Ocean automatically generates an SSH public/private key pair (or presents you with an existing one) for your current/logged in Jenkins user. This credential is automatically registered in Jenkins with the following details for this Jenkins user:

- **Domain**: `blueocean-private-key-domain`
- **ID**: `jenkins-generated-ssh-key`
- **Name**: `<jenkins-username> (jenkins-generated-ssh-key)`

You need to ensure that this SSH public/private key pair has been registered with your Git server before continuing. If you have not already done this, follow these 2 steps. Otherwise, continue on.

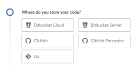
1. Configure the SSH public key component of this key pair (which you can copy and paste from the Blue Ocean interface) for the remote Git server's user account (e.g. within the `authorized_keys` file of the machine's `gituser/.ssh` directory).
   Note: This process allows your Jenkins user to access the repositories that your Git server's user account (e.g. `gituser`) has access to. Read more about this in Setting Up the Server of the Pro Git documentation.
2. When done, return to the Blue Ocean interface.

Click the **Create Pipeline** button.

Blue Ocean will then scan your local repository's branches for a `Jenkinsfile` and will commence a Pipeline run for each branch containing a `Jenkinsfile`. If Blue Ocean cannot find any `Jenkinsfile`, you will be prompted to begin creating one through the Pipeline editor.

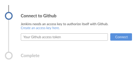
For a repository on GitHub

To create your Pipeline project directly for a repository on GitHub, click the **GitHub** button under **Where do you store your code?**



In the **Connect to GitHub** section, enter your GitHub access token into the **Your GitHub access**

token field.
If you previously configured Blue Ocean to connect to GitHub using a personal access token, Blue Ocean takes you directly to the choosing your GitHub account/organization and repository steps below.


Connect to Github
Jenkins needs an access key to authorize itself with Github.

If you do not have a GitHub access token, click the **Create an access key here** link to open GitHub to the **New personal access token** page.

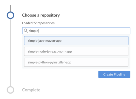
Create your access token

1. In the new tab, sign in to your GitHub account (if necessary) and on the GitHub **New Personal Access Token** page, specify a brief **Token description** for your GitHub access token (e.g. `Blue Ocean`).
   Note: An access token is usually an alphanumeric string that respresents your GitHub account along with permissions to access various GitHub features and areas through your GitHub account. The new access token process (triggered through the **Create an access key here** link above) has the appropriate permissions pre-selected, which Blue Ocean requires to access and interact with your GitHub account.
2. Scroll down to the end of the page and click **Generate token**.
3. On the resulting **Personal access tokens** page, copy your newly generated access token.
4. Back in Blue Ocean, paste the access token into the **Your GitHub access token** field and click **Connect**.
   Your current/logged in Jenkins user now has access to your GitHub account (provided by your access token), so you can now choose your GitHub account/organization and repository.
   Jenkins registers this credential with the following details for this Jenkins user:
   - **Domain**: `blueocean-github-domain`
   - **ID**: `github`
   - **Name**: `<jenkins-username>/****** (GitHub Access Token)`

Choose your GitHub account/organization and repository

At this point, Blue Ocean prompts you to choose your GitHub account or an organization you are a member of, as well as the repository it contains from which to build your Pipeline project.

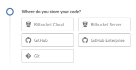
1. In the **Which organization does the repository belong to?** section, click either:
   - Your GitHub account to create a Pipeline project for one of your own GitHub repositories or one which you have forked from elsewhere on GitHub.
   - An organization you are a member of to create a Pipeline project for a GitHub repository located within this organization.
2. In the **Choose a repository** section, click the repository (within your GitHub account or organization) from which to build your Pipeline project.
   Tip: If your list of repositories is long, you can filter this list using the **Search** option.



3. Click **Create Pipeline**.
   Blue Ocean will then scan your local repository's branches for a `Jenkinsfile` and will commence a Pipeline run for each branch containing a `Jenkinsfile`. If Blue Ocean cannot find any `Jenkinsfile`, you will be prompted to begin creating one through the Pipeline editor (by clicking **Create Pipeline** again).
   Note: Under the hood, a Pipeline project created through Blue Ocean is actually "multibranch Pipeline". Therefore, Jenkins looks for the presence of at least one Jenkinsfile in any branch of your repository.

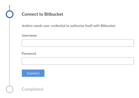
For a repository on Bitbucket Cloud

To create your Pipeline project directly for a Git or Mercurial repository on Bitbucket Cloud, click the **Bitbucket Cloud** button under **Where do you store your code?**



In the **Connect to Bitbucket** section, enter your Bitbucket email address and password into the **Username** and **Password** fields, respectively. Note that:

- If you previously configured Blue Ocean to connect to Bitbucket with your email address and password, Blue Ocean takes you directly to the choosing your Bitbucket account/team and repository steps below.
- If you entered these credentials, Jenkins registers them with the following details for this Jenkins user:
  - **Domain**: `blueocean-bitbucket-cloud-domain`
  - **ID**: `bitbucket-cloud`
  - **Name**: `<bitbucket-user@email.address>/****** (Bitbucket server credentials)`


Connect to Bitbucket
Jenkins needs user credential to authorize itself with Bitbucket.
Username
Password
Connect
Completed

Click **Connect** and your current/logged in Jenkins user will now have access to your Bitbucket account. You can now choose your Bitbucket account/team and repository.

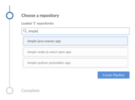
Choose your Bitbucket account/team and repository

At this point, Blue Ocean prompts you to choose your Bitbucket account or a team you are a member of, as well as the repository it contains from which to build your Pipeline project.

1. In the **Which team does the repository belong to?** section, click either:
   - Your Bitbucket account to create a Pipeline project for one of your own Bitbucket repositories or one which you have forked from elsewhere on Bitbucket.
   - A team you are a member of to create a Pipeline project for a Bitbucket repository located

within this team.

2. In the **Choose a repository** section, click the repository (within your Bitbucket account or team) from which to build your Pipeline project.
   Tip: If your list of repositories is long, you can filter this list using the **Search** option.

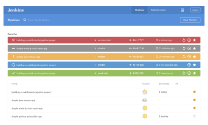
   

3. Click **Create Pipeline**.
   Blue Ocean will then scan your local repository's branches for a `Jenkinsfile` and will commence a Pipeline run for each branch containing a `Jenkinsfile`. If Blue Ocean cannot find any `Jenkinsfile`, you will be prompted to begin creating one through the Pipeline editor (by clicking **Create Pipeline** again).
   Note: Under the hood, a Pipeline project created through Blue Ocean is actually "multibranch Pipeline". Therefore, Jenkins looks for the presence of at least one Jenkinsfile in any branch of your repository.

Dashboard

Blue Ocean's "Dashboard" is the default view shown when you open Blue Ocean and shows an overview of all Pipeline projects configured on a Jenkins instance.

The Dashboard consists of a blue navigation bar at the top, the Pipelines list, as well as the Favorites list.



Navigation bar

The Dashboard includes the blue-colored navigation bar along the top of the interface.

This bar is divided into two sections - a common section along the top and a contextual section below. The contextual section changes depending on the current Blue Ocean page you are viewing.

When viewing the Dashboard, the navigation bar's contextual section includes the:

- **Search pipelines** field, to filter the Pipelines list to show items containing the text you enter into this field.
- **New Pipeline** button, which begins the create a Pipeline process.

Pipelines list

The "Pipelines" list is the Dashboard's default list and upon accessing Blue Ocean for the first time, this is the only list shown on the Dashboard.

This list shows the overall state of each Pipeline configured on the Jenkins instance (which can also include other Jenkins items). For a given item in this list, the following information is indicated:

- The item's **NAME**,
- The item's **HEALTH**,
- The numbers of **BRANCH**es and pull requests (**PR**s) of the Pipeline's source control repository which are passing or failing, and
- A star indicating whether or not the default/main branch of the item has been manually added to your current Jenkins user's [favorites-list].

Clicking on an item's star will toggle between:

- Adding the default branch of the item's repository to your current user's Favorites list (indicated by a solid " "), and
- Removing the item's default branch from this list (indicated by an outlined " ").

Clicking on an item in the Pipelines list will display that item's Activity View.

Favorites list

The **Favorites** list appears above the Dashboard's default [pipelines-list] when at least one Pipeline/item is present in your user's Favorites list.

This list provides key information and actions for a core subset of your user's accessible items in the [pipelines-list]. This key information includes the current run status for an item and its repository's branch, as well as other details about the item's run, including the name of the branch, the initial part of the commit hash and the time of the last run. Items in this list also include clickable icons to run or re-run the item on the repository branch indicated.

You should only add an item (or one of the repository's specific branches) to your Favorites list if you need to examine that item's branch on a regular basis. Adding an item's specific branch to your Favorites list can be done through the item's Activity View.

Blue Ocean automatically adds branches or PRs to this list when a they contain a run that has changes authored by the current user.

You can also manually remove items from your Favorites list by clicking on the solid " " in this list. When the last item is removed from this list, the list is removed from the interface.

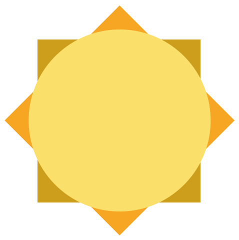
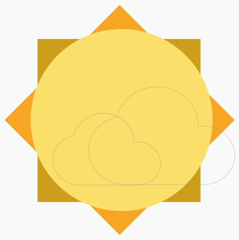
Clicking on an item in the Favorites list will open the Pipeline run details for latest run on the repository branch or PR indicated.

Health icons

Blue Ocean represents the overall health of a Pipeline/item or one of its repository's branches using weather icons, which change depending on the number of recent builds that have passed.

Health icons on the Dashboard represent overall Pipeline health, whereas the health icons in the Branches tab of the Activity View represent the overall health for each branch.

Table 2. Health icons (best to worst)

Icon	Health
	**Sunny**, more than 80% of Runs passing
	**Partially Sunny**, 61% to 80% of Runs passing

Icon	Health
☁	**Cloudy**, 41% to 60% of Runs passing
🌧	**Raining**, 21% to 40% of Runs passing
⛈	**Storm**, less than 21% of Runs passing

Run status

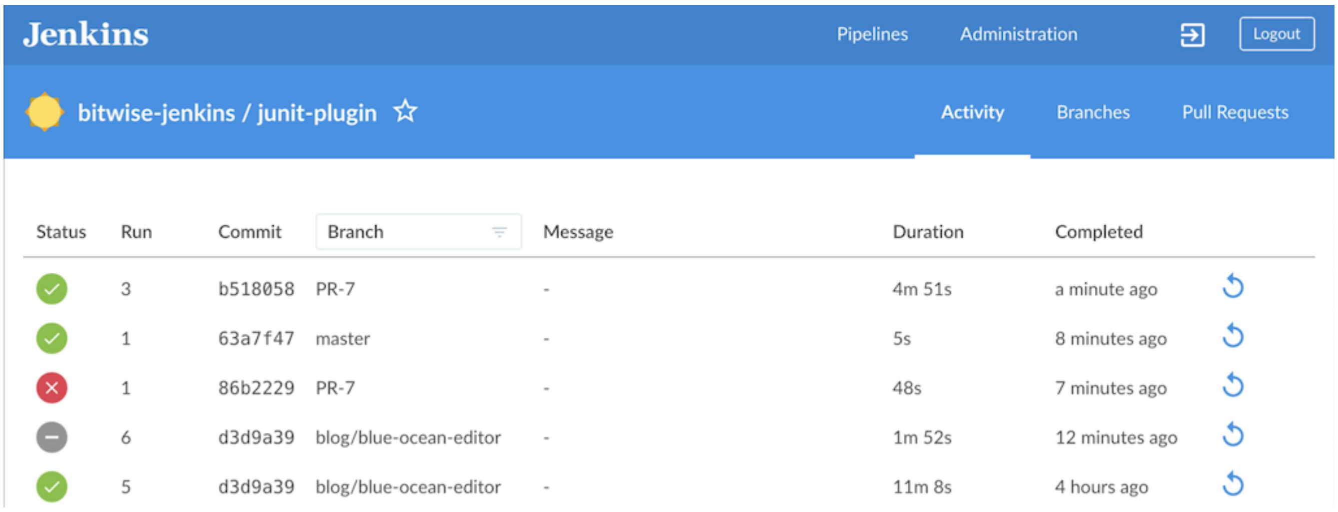
Blue Ocean represents the run status of a Pipeline/item or one of its repository's branches using a consistent set of icons throughout.

Table 3. Run status icons

Icon	Status
	In Progress
	Passed
	Unstable
	Failed
	Aborted

Activity View

The Blue Ocean Activity View shows the all activity related to one Pipeline.



Status	Run	Commit	Branch	Message	Duration	Completed
✓	3	b518058	PR-7	-	4m 51s	a minute ago
✓	1	63a7f47	master	-	5s	8 minutes ago
✗	1	86b2229	PR-7	-	48s	7 minutes ago
−	6	d3d9a39	blog/blue-ocean-editor	-	1m 52s	12 minutes ago
✓	5	d3d9a39	blog/blue-ocean-editor	-	11m 8s	4 hours ago

Navigation Bar

The Activity View includes the standard navigation bar at the top, with a local navigation bar below that. The local navigation bar includes:

- **Pipeline Name** - Clicking on this displays the default activity tab
- **Favorites Toggle** - Clicking the "Favorite" symbol (a star outline " ") adds a branch to the favorites list shown on the Dashboard's "Favorites" list for this user.
- **Tabs** (Activity, Branches, Pull Requests) - Clicking one of these will display that tab of the Activity View.

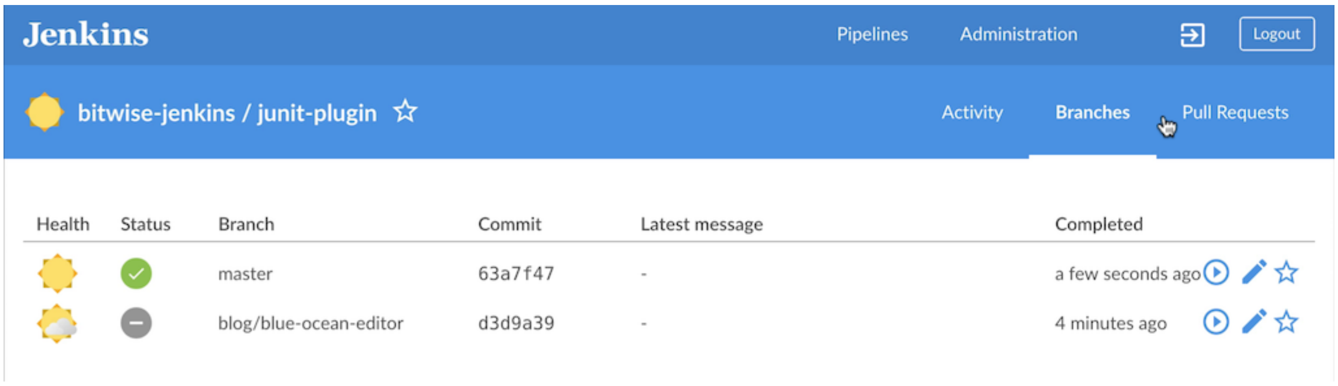
Activity

The default tab of the Activity View, the "Activity" tab, shows a list of the latest completed or in-progress Runs. Each line in the list shows the status of the Run, id number, commit information, duration, and when the run completed. Clicking on a Run will bring up the Pipeline Run Details for that Run. "In Progress" Runs can be aborted from this list by clicking on the "Stop" symbol (a square " " inside a circle). Runs that have completed can be re-run by clicking the "Re-run" symbol (a counter-clockwise arrow " "). The list can be filtered by branch or pull request by clicking on the "branch" drop-down in the list header.

This list does not allow runs to be edited or marked as favorites. Those actions can be done from the "branches" tab.

Branches

The "Branches" tab shows a list of all branches that have a completed or in-progress Run in the current Pipeline. Each line in the list corresponds to a branch in source control, [23: en.wikipedia.org/wiki/Source_control_management] showing overall health of the branch based on recent runs, status of the most recent run, id number, commit information, duration, and when the run completed.

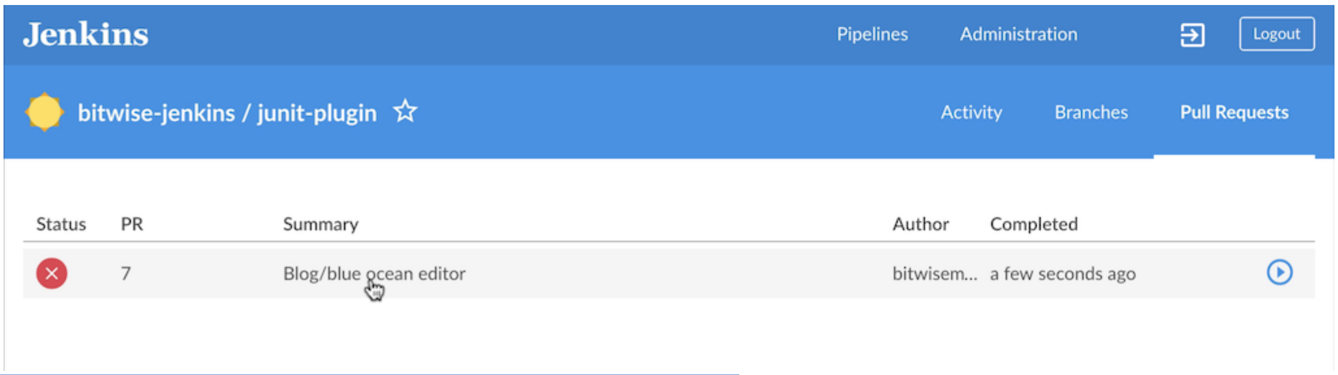


Health	Status	Branch	Commit	Latest message	Completed
	✓	master	63a7f47	-	a few seconds ago
	–	blog/blue-ocean-editor	d3d9a39	-	4 minutes ago

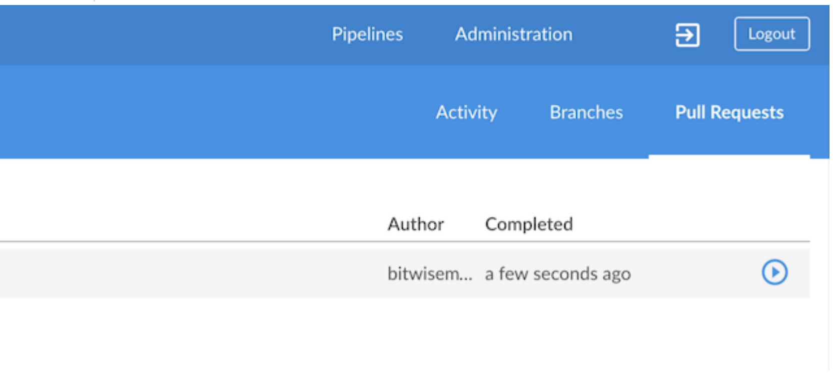
Clicking on a branch in this list will bring up the Pipeline Run Details for the latest completed or in-progress Run of that branch. "In Progress" runs can be aborted from this list by clicking on the "Stop" symbol (a square " " inside a circle). Pull requests whose latest run has completed can be run again by clicking the "Play" symbol (a triangle "▶" inside a circle). Clicking the "Edit" symbol (similar to a pencil " ") opens the pipeline editor on the Pipeline for that brach. Clicking the "Favorite" symbol (a star outline " ") adds a branch to the favorites list shown on the Dashboard's "Favorites" list for this user. A favorite branch will show a solid star " " and clicking it removes this branch from the favorites.

Pull Requests

The "Pull Requests" tab shows a list of all Pull Requests for the current Pipeline that have a completed or in-progress Run. (Some source control systems call these "Merge Requests", others do not support them at all.) Each line in the list corresponds to a pull request in source control, showing the status of the most recent run, id number, commit information, duration, and when the run completed.



Status	PR	Summary	Author	Completed
✗	7	Blog/blue ocean editor	bitwisem...	a few seconds ago



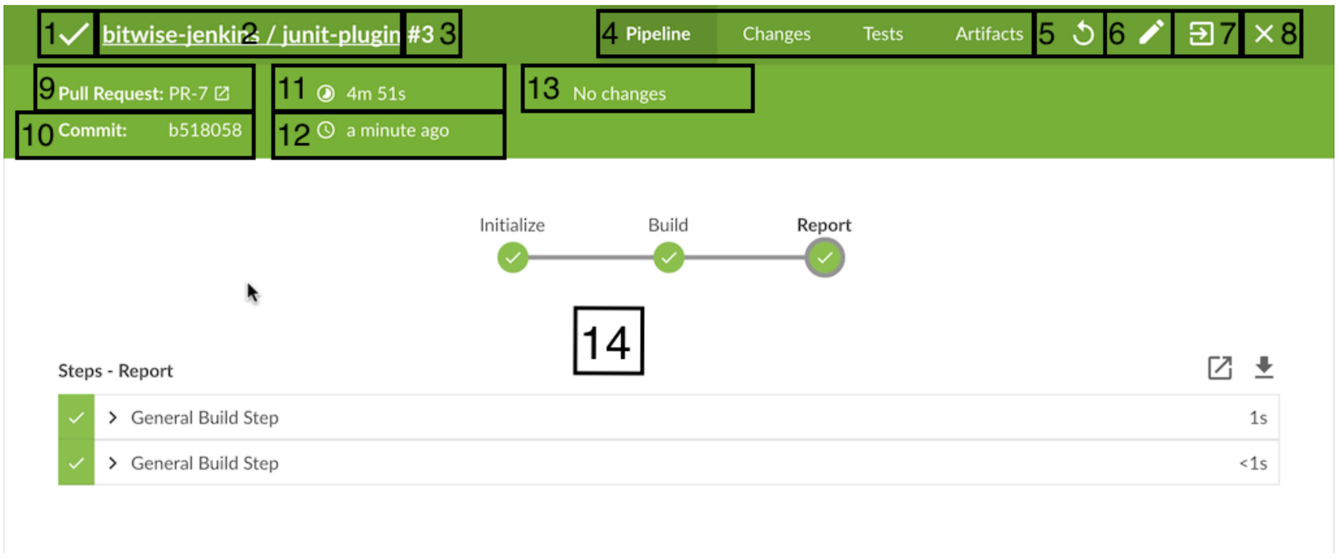
Author	Completed
bitwisem...	a few seconds ago

Blue Ocean displays pull requests separately from branches, but otherwise the Pull Requests list behaves similar to the Branches list. Clicking on a pull request in this list will bring up the Pipeline Run Details for the latest completed or in-progress Run of that pull request. "In Progress" runs can be aborted from this list by clicking on the "Stop" symbol (a square " " inside a circle). Pull requests whose latest run has completed can be run again by clicking the "Play" symbol (a triangle "▶" inside a circle). Pull request do not display "Heath Icons" and cannot be edited or marked as favorites.

NOTE By default, when a Pull Request is closed, Jenkins will remove the Pipeline from Jenkins (to be cleaned up at a later date), and runs for that Pull Request will not longer be accessible from Jenkins. That can be changed by changing the configuration of the underlying Multi-branch Pipeline job.

Pipeline Run Details View

The Blue Ocean Pipeline Run Details view shows the information related to a single Pipeline Run and allows users to edit or replay that run. Below is a detailed overview of the parts of the Run Details view.



1. **Run Status** - This icon, along with the background color of the top menu bar, indicates the status of this Pipeline run.
2. **Pipeline Name** - The name of this run's Pipeline.
3. **Run Number** - The id number for this Pipeline run. Id numbers unique for each Branch (and Pull Request) of a Pipeline.
4. **Tab Selector** - View one of the detail tabs for this run. The default is "Pipeline".
5. **Re-run Pipeline** - Execute this run's Pipeline again.
6. **Edit Pipeline** - Open this run's Pipeline in the Pipeline Editor.
7. **Go to Classic** - Switch to the "Classic" UI view of the details for this run.
8. **Close Details** - This closes the Details view and returns the user to the <<activity, Activity view> for this Pipeline.
9. **Branch** or **Pull Request** - the branch or pull request for this run.
10. **Commit Id** - Commit id for this run.
11. **Duration** - The duration of this run.
12. **Completed Time** - How long ago the this run completed.
13. **Change Author** - Names of the authors with changes in this run.
14. **Tab View** - Shows the information for the selected tab.

Pipeline Run Status

Blue Ocean makes it easy to see the status of the current Pipeline Run by changing the color of the top menu bar to match the status: blue for "In progress", green for "Passed", yellow for "Unstable", red for "Failed", and gray for "Aborted".

Special cases

Blue Ocean is optimized for working with Pipelines in Source Control, but it can display details for other kinds of projects as well. Blue Ocean offers the same tabs for all supported projects types, but those tabs may display different information.

Pipelines outside of Souce Control

For Pipelines that are not based on Source Control, Blue Ocean still shows the "Commit Id", "Branch", and "Changes", but those fields are left blank. In this case, the top menu bar does not include the "Edit" option.

Freestyle Projects

For Freestyle projects, Blue Ocean still offers the same tabs, but the Pipeline tab only shows the console log output. The "Rerun" or "Edit" options are also not shown in the top menu bar.

Matrix projects

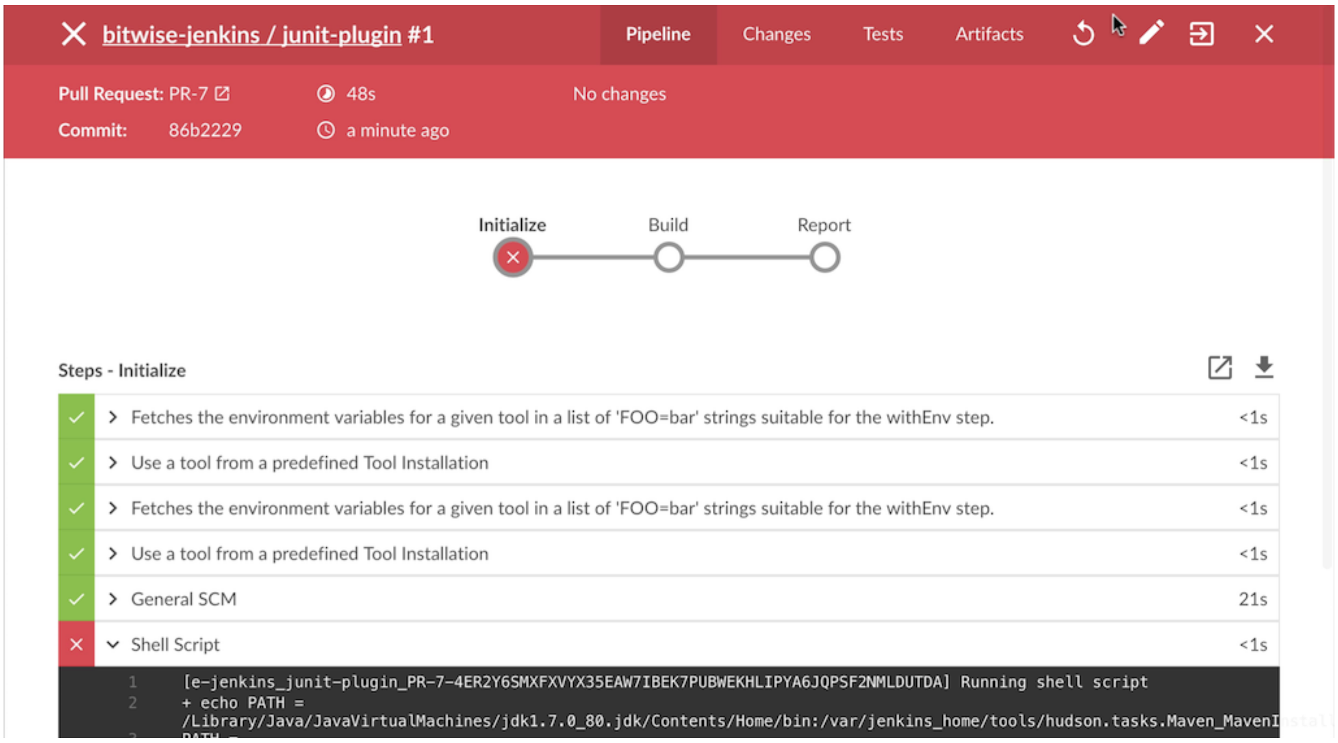
Matrix projects are not supported in Blue Ocean. Viewing a Matrix project will redirect to the "Classic UI" view for that project.

Tabs

Each of the tabs of the Run Detail view provides information on a specific aspect of a run.

Pipeline

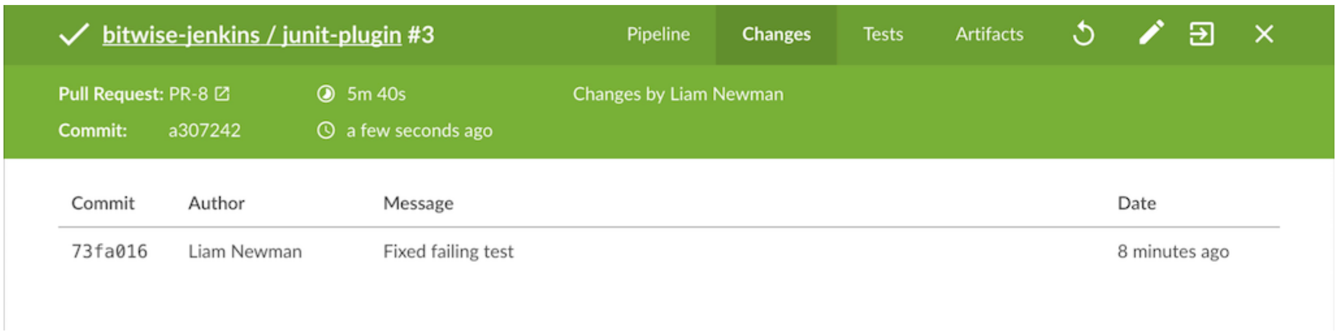
This is the default tab and gives an overall view of the flow of this Pipeline Run. It shows each stage and parallel branch, the steps in those stages, and the console output from those steps. The overview image above shows a successful Pipeline run. If a particular step during the run fails, this tab will automatically default to showing the console log from the failed step. The image below shows a failed Run.



✓	Fetches the environment variables for a given tool in a list of 'FOO=bar' strings suitable for the withEnv step.	<1s
✓	Use a tool from a predefined Tool Installation	<1s
✓	Fetches the environment variables for a given tool in a list of 'FOO=bar' strings suitable for the withEnv step.	<1s
✓	Use a tool from a predefined Tool Installation	<1s
✓	General SCM	21s
✕	Shell Script	<1s

```
[e-jenkins_junit-plugin_PR-7-4ER2Y6SMXFXVYX35EAW7IBEK7PUBWEKHLIPYA6JQPSF2NMLDUTDA] Running shell script
+ echo PATH =
/Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home/bin:/var/jenkins_home/tools/hudson.tasks.Maven_MavenInstall
```

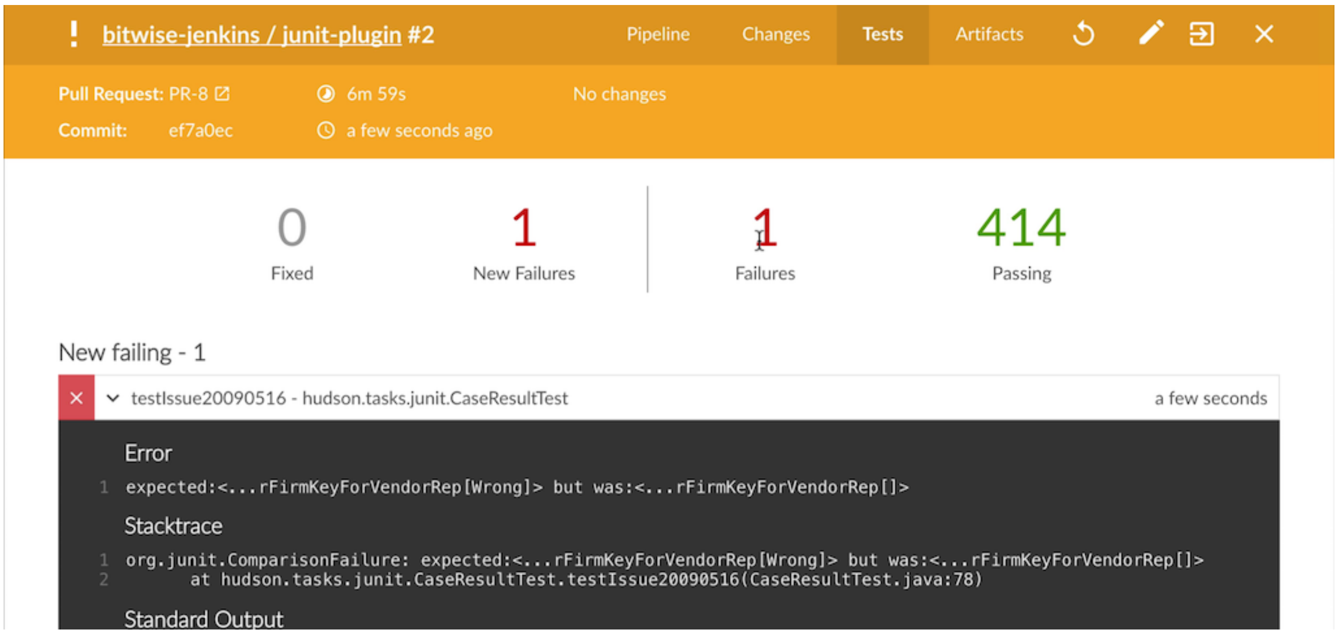
Changes



Commit	Author	Message	Date
73fa016	Liam Newman	Fixed failing test	8 minutes ago

Tests

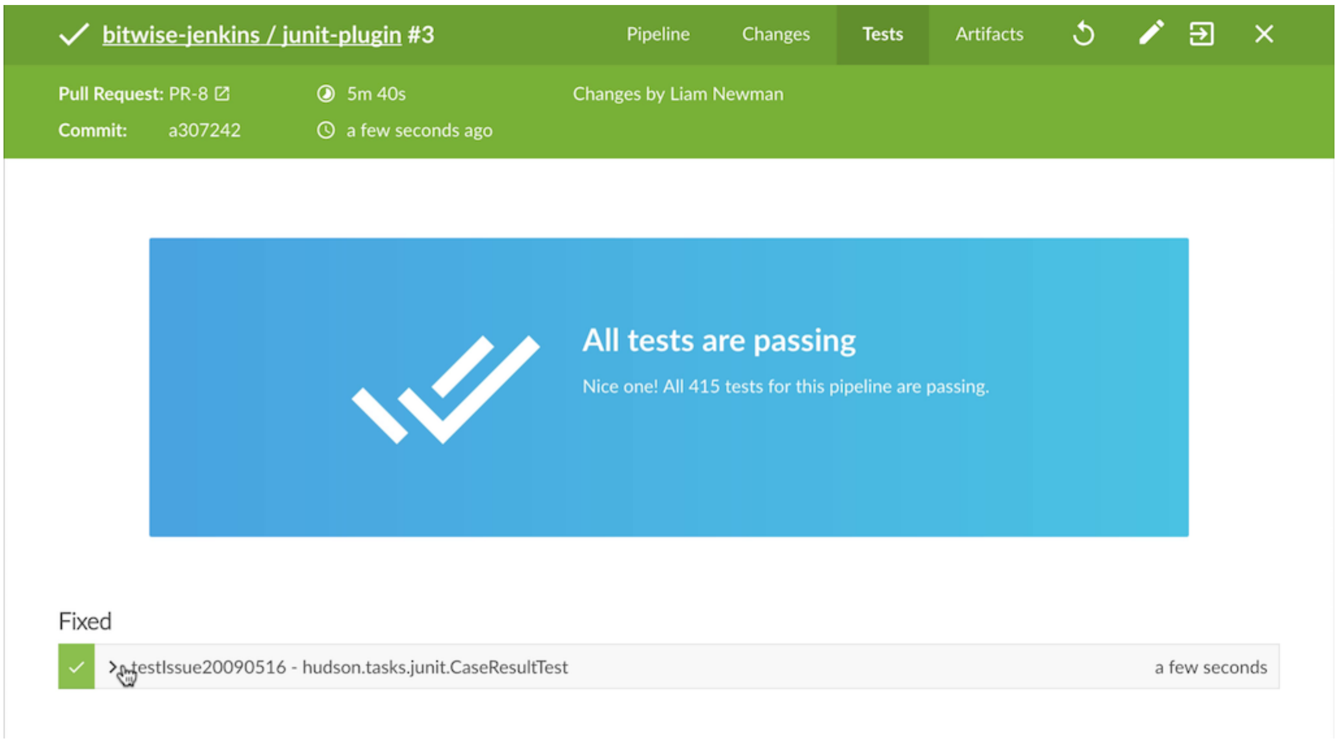
The "Tests" tab shows information about test results for this run. This tab will only contain information if a test result publishing step, such as the "Publish JUnit test results" (`junit`) step. If no results are recorded this table will say that, If all tests pass, this tab will report the total number of passing tests. In the case of failures, the tab will display logs details from the failures as shown below.



```
expected:<...rFirmKeyForVendorRep[Wrong]> but was:<...rFirmKeyForVendorRep[]>
```

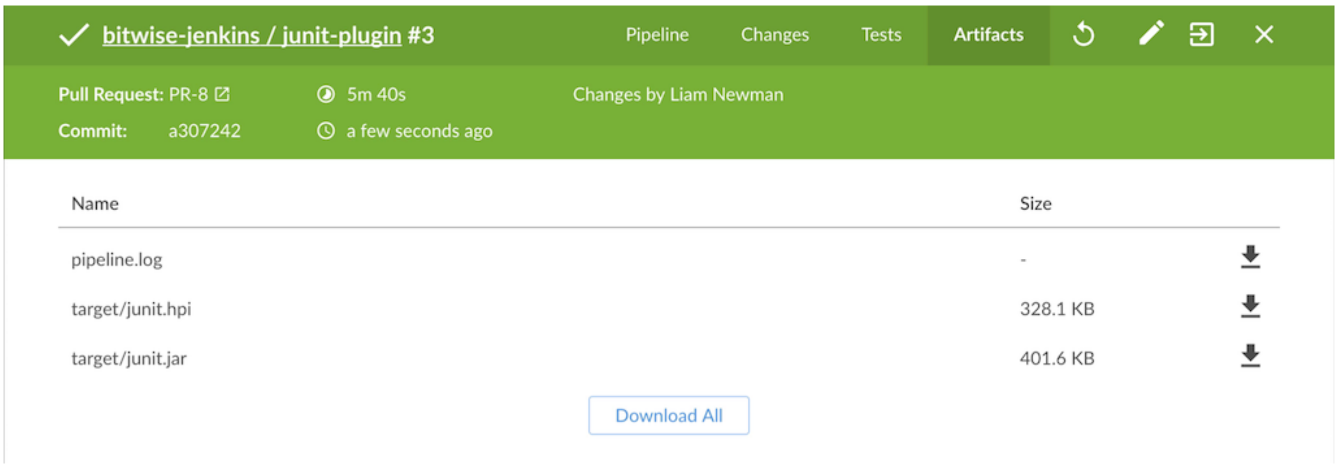
```
org.junit.ComparisonFailure: expected:<...rFirmKeyForVendorRep[Wrong]> but was:<...rFirmKeyForVendorRep[]>
      at hudson.tasks.junit.CaseResultTest.testIssue20090516(CaseResultTest.java:78)
```

When the previous Run had failures and the current run fixes those failures, this tab will note the fixed texts and display their logs as well.



Artifacts

The "Artifacts" tabs show a list of any artifacts saved using the "Archive Artifacts" (`archive`) step. Clicking on a item in the list will download it. The full output log from the Run can be downloaded from this list.

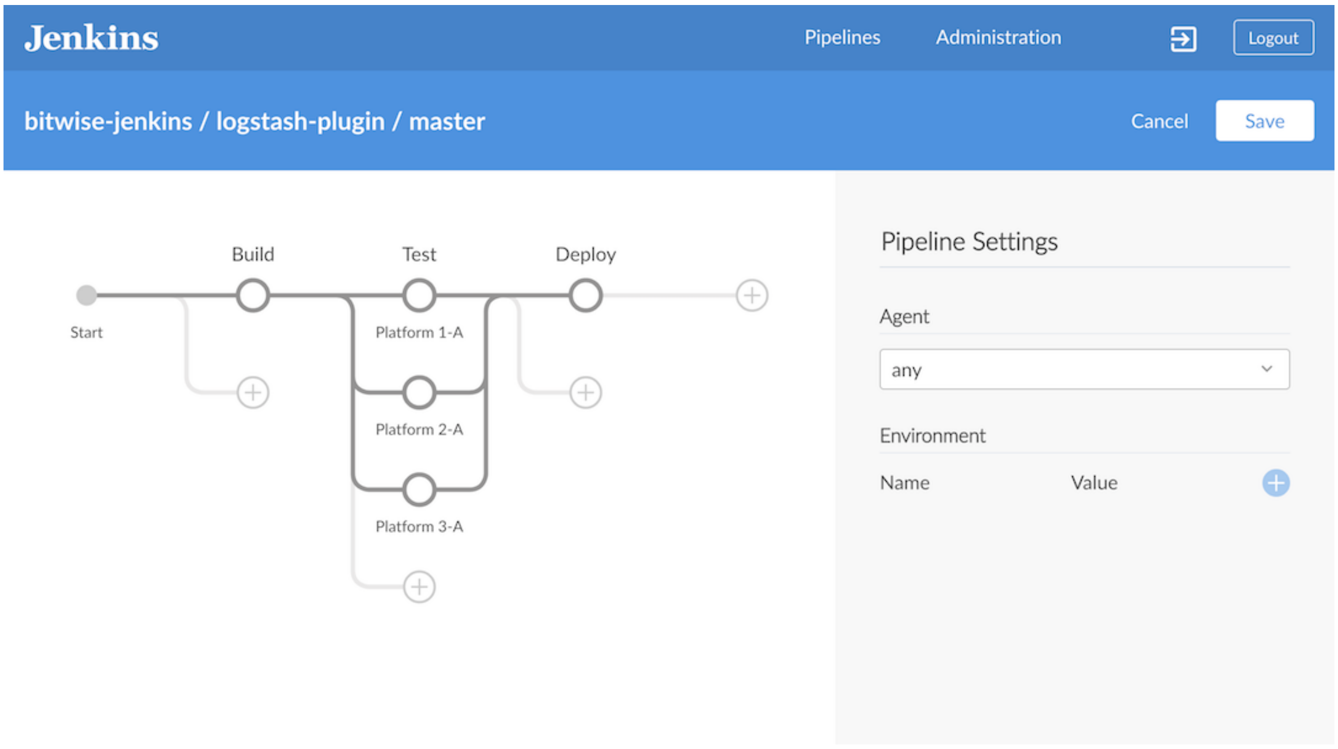


Name	Size
pipeline.log	-
target/junit.hpi	328.1 KB
target/junit.jar	401.6 KB

Pipeline Editor

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

The editor allows users to create and edit Declarative Pipelines, add stages and parallelized tasks that can run at the same time, depending on their needs. When finished, the editor saves the Pipeline to a source code repository as a `Jenkinsfile`. If the Pipeline needs to be changed again, Blue Ocean makes it easy to jump back in into the visual editor to modify the Pipeline at any time.



Starting the editor

To use the editor a user must first have created a pipeline in Blue Ocean or have one or more existing Pipelines already created in Jenkins. If editing an existing pipeline, the credentials for that pipeline must allow pushing of changes to the target repository.

The editor can be launched via:

- Dashboard "New Pipeline" button
- Activity View for Single Run
- Pipeline Run Details

Limitations

- SCM-based Declarative Pipelines only
- Credentials must have write permission
- Does not have full parity with Declarative Pipeline

- Pipeline re-ordered and comments removed

Navigation bar

The Pipeline Editor includes the standard navigation bar at the top, with a local navigation bar below that. The local navigation bar includes:

- **Pipeline Name** - This will include the branch depending or how
- **Cancel** - Discard changes made to the pipeline.
- **Save** - Open the Save Pipeline Dialog.

Pipeline settings

By default, the right side of editor shows the "Pipeline Settings". This sheet can be accessed by clicking anywhere in the Stage editor that is not a Stage or one of the "Add Stage" buttons.

Agent

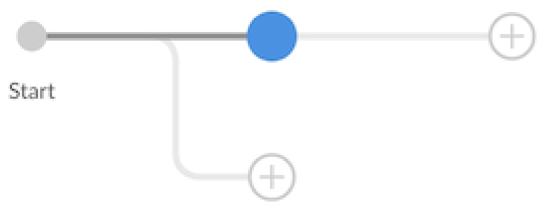
The "Agent" section controls what agent the Pipeline will use. This is the same as the "agent" directive.

Environment

The "Environment" sections lets us set environment variables for the Pipeline. This is the same as the "environment" directive.

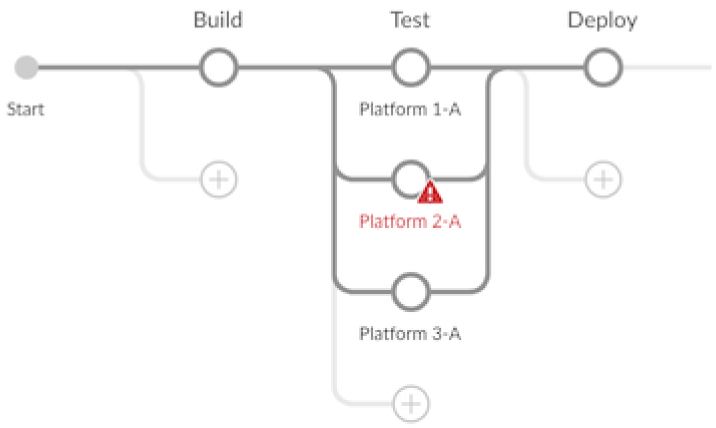
Stage editor

The left side editor screen contains the Stage editor, used for creating the stages of a Pipeline.



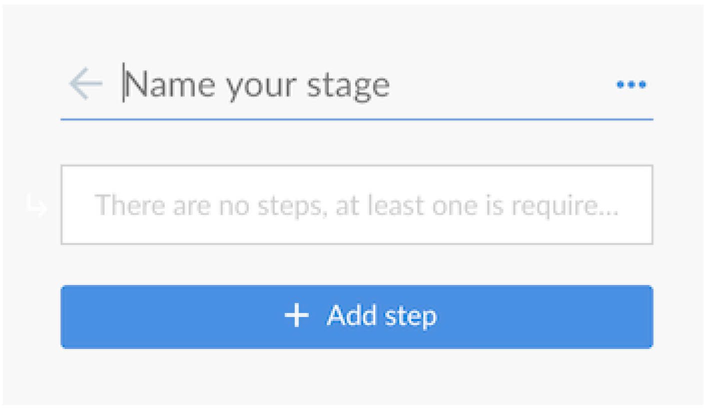
Stages can be added to the Pipeline by clicking the "+" button to the right of an existing stage. Parallel stages can be added by clicking the "\+" button below an existing Stage. Stages can be deleted using the context menu in the stage configuration sheet.

The Stage editor will display the name of each Stage once it has been set. Stages that contain incomplete or invalid information will display a warning symbol. Pipelines can have validation errors while they are being edited, but cannot be saved until the errors are fixed.

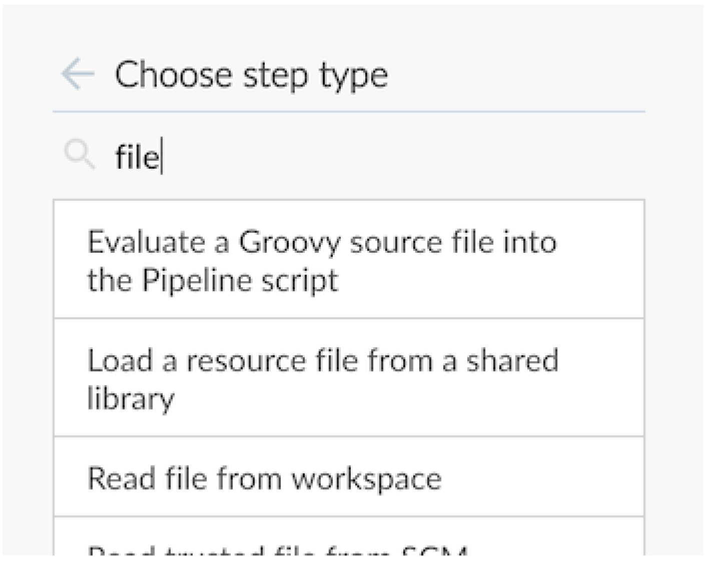


Stage configuration

Selecting a stage in the Stage editor will open the "Stage Configuration" sheet on the right side. Here we can can change the name of the Stage, delete the Stage, and add steps to the Stage.

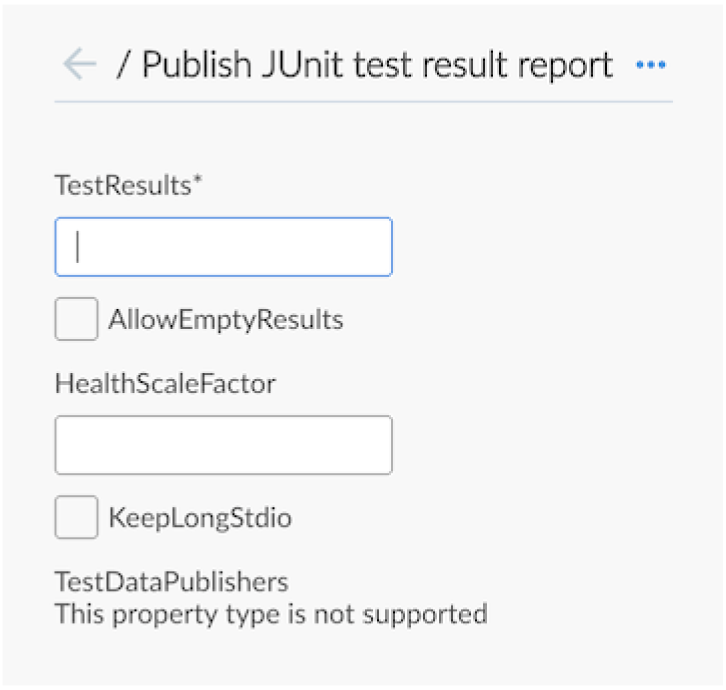


The name of the Stage can be set at the top of the Stage Configuration sheet. The context menu (three dots on the upper right), can be used to delete the current stage. Clicking "Add step" will display the list of available Steps types with a search bar at the top. Steps can be deleted using the context context menu in the step configuration sheet. Adding a step or selecting an existing step will open the step configuration sheet.


Choose step type
file
Evaluate a Groovy source file into the Pipeline script
Load a resource file from a shared library
Read file from workspace

Step configuration

Selecting a step from the Stage configuration sheet will open the Step Configuration sheet.


/ Publish JUnit test result report

TestResults*

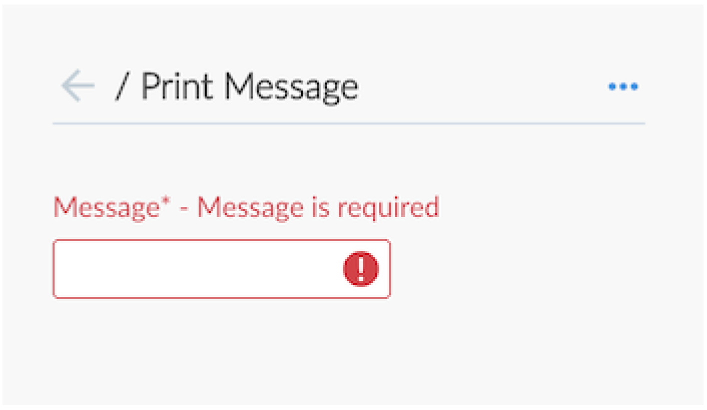
AllowEmptyResults

HealthScaleFactor

KeepLongStdio

TestDataPublishers
This property type is not supported

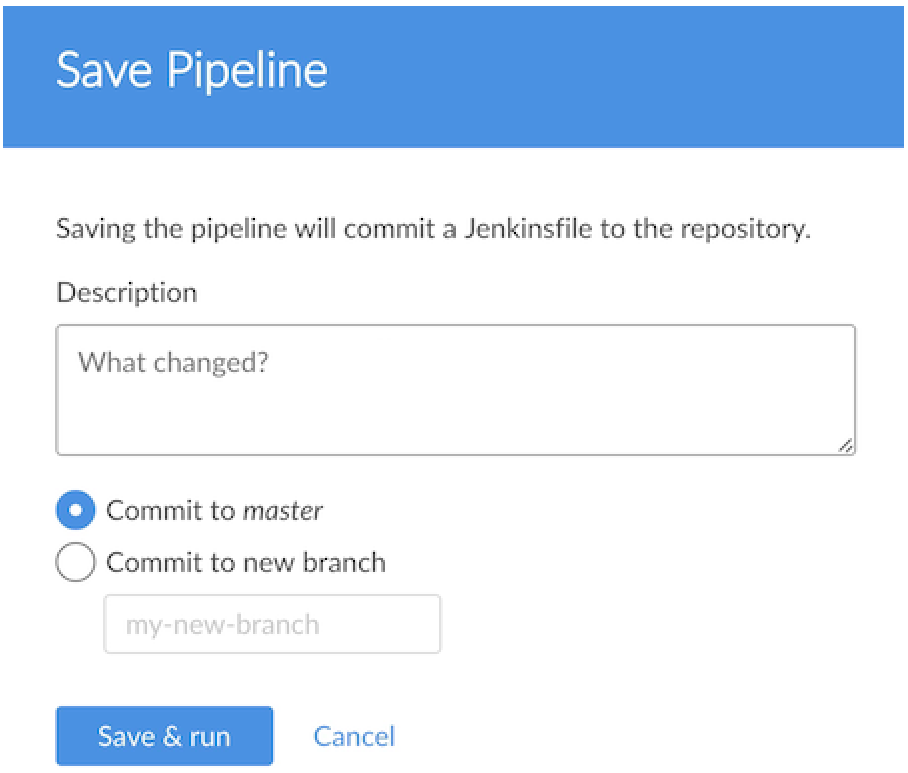
This sheet will differ depending on the step type, containing whatever fields or controls are needed. The name of the Step cannot be changed. The context menu (three dots on the upper right), can be used to delete the current step. Fields that contain incomplete or invalid information will display a warning symbol. Pipelines can have validation errors while they are being edited, but cannot be saved until the errors are fixed.


/ Print Message

Message* - Message is required

Save Pipeline dialog

In order to be run, changes to a Pipeline must be saved in source control The "Save Pipeline" dialog controls saving of changes to source control.


Save Pipeline

Saving the pipeline will commit a Jenkinsfile to the repository.

Description

What changed?

[✓ Commit to *master*]

Commit to new branch

my-new-branch

A helpful description of the changes can be added or left blank. The dialog also supports saving changes the same branch or entering a new branch to save to. Clicking on "Save & run" will save any changes to the Pipeline as a new commit, will start a new Pipeline Run based on those changes, and will navigate to the Activity View for this pipeline.

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 a system administrator and want learn how to back-up, restore, maintain as Jenkins servers and nodes, see Jenkins System Administration.

For an overview of content in the Jenkins User Handbook, see User Handbook overview.

Configuring the System

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.

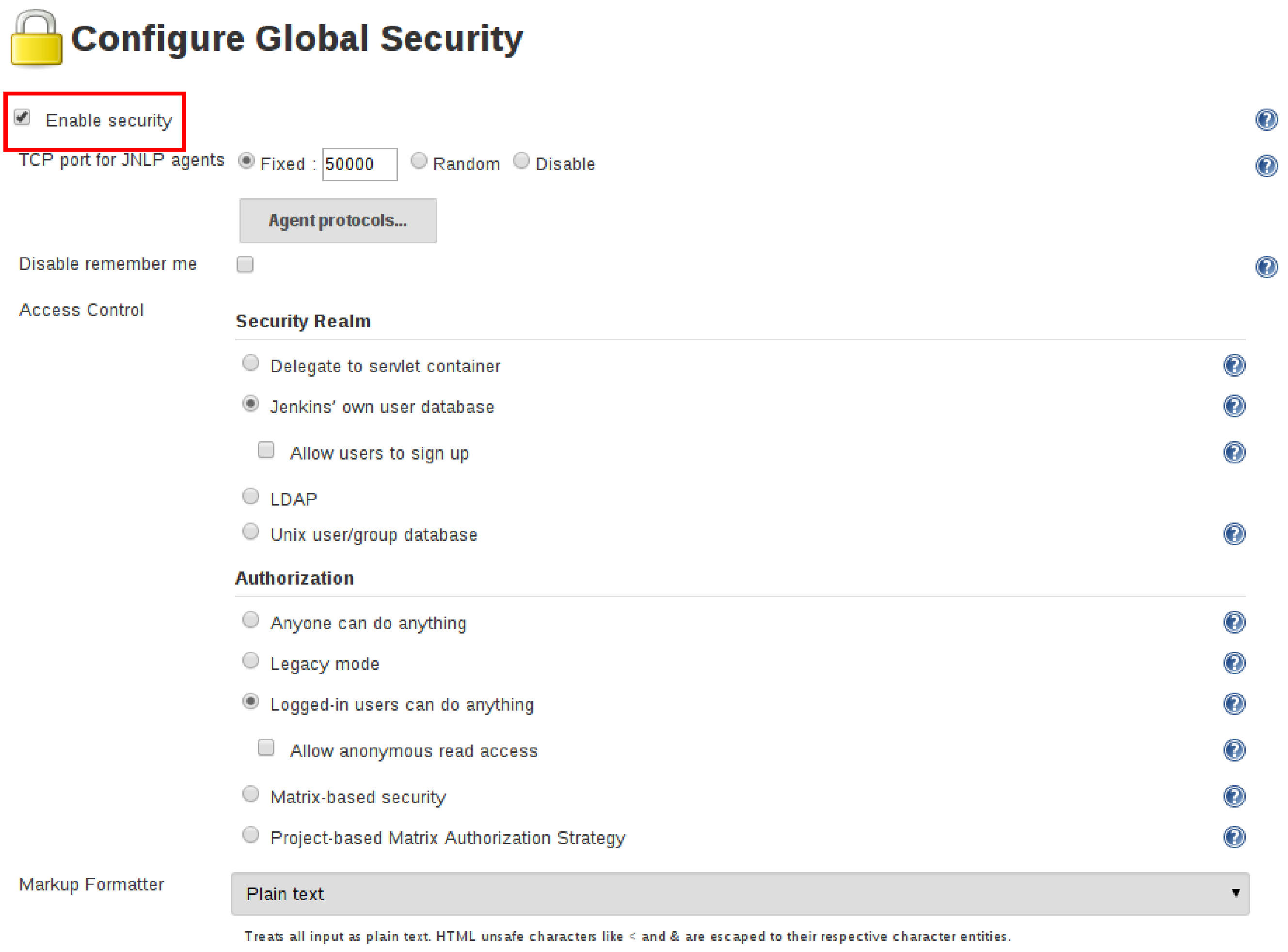
As of Jenkins 2.0, many of the security options were enabled by default to ensure that Jenkins environments remained secure unless an administrator explicitly disabled certain protections.

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.



Configure Global Security

[✓] Enable security

TCP port for JNLP agents: [● Fixed : 50000] ○ Random ○ Disable

Agent protocols...

Disable remember me: ☐

Access Control

Security Realm
- ○ Delegate to servlet container
- ● Jenkins' own user database
  - ☐ Allow users to sign up
- ○ LDAP
- ○ Unix user/group database

Authorization
- ○ Anyone can do anything
- ○ Legacy mode
- ● Logged-in users can do anything
  - ☐ Allow anonymous read access
- ○ Matrix-based security
- ○ Project-based Matrix Authorization Strategy

Markup Formatter: Plain text

Treats all input as plain text. HTML unsafe characters like < and & are escaped to their respective character entities.

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

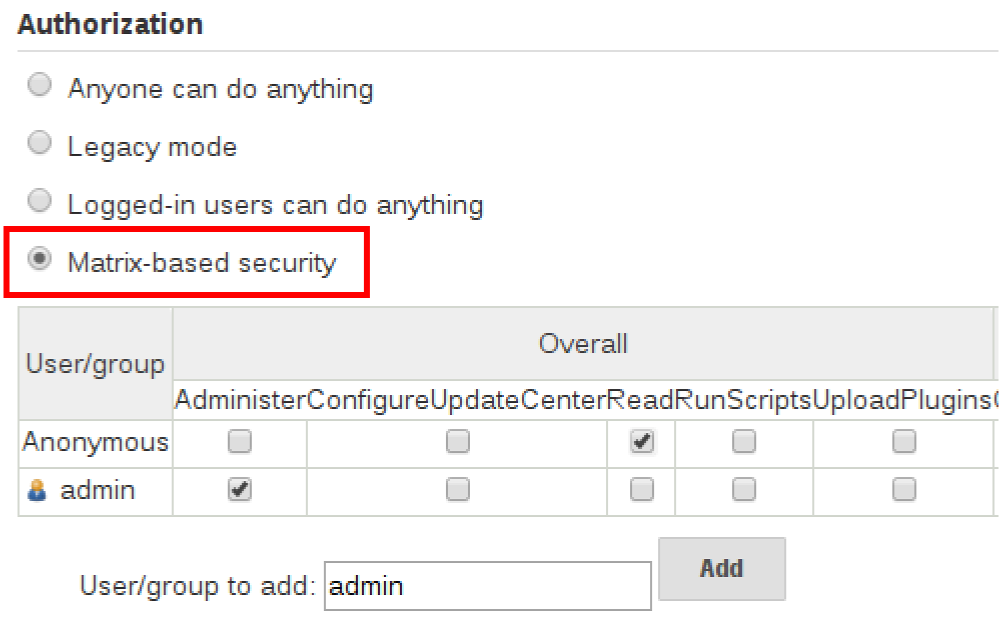
This authorization scheme allows for granular control over which users and groups are able to perform which actions in the Jenkins environment (see the screenshot below).

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.



Authorization

- Anyone can do anything
- Legacy mode
- Logged-in users can do anything
- [✓ Matrix-based security]

User/group	Overall				
	Administer	ConfigureUpdateCenter	Read	RunScripts	UploadPlugins
Anonymous	☐	☐	✓	☐	☐
admin	✓	☐	☐	☐	☐

User/group to add: admin

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.

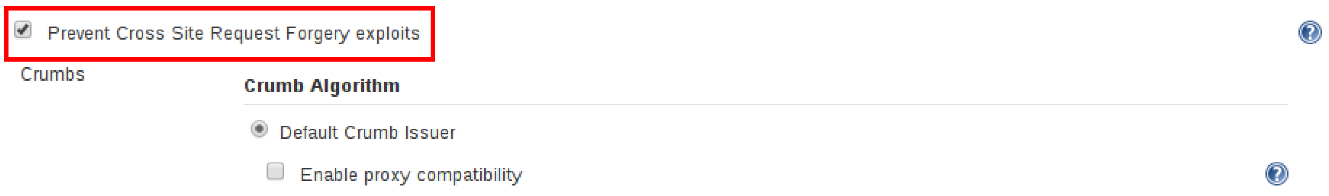
By default the **Markup Formatter** configuration is set to **Plain Text** which will escape unsafe characters such as `<` and `&` to their respective character entities.

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) [24: 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.


☑ Prevent Cross Site Request Forgery exploits

Crumbs

Crumb Algorithm

● Default Crumb Issuer

☐ Enable proxy compatibility

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 [25: 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.

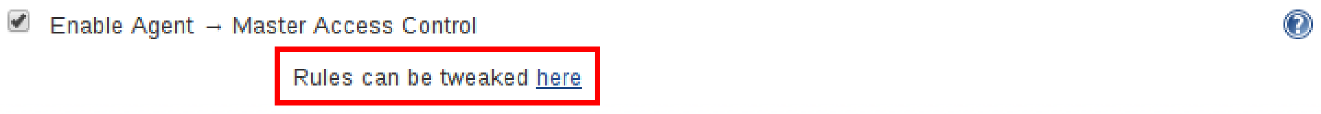
☑ Enable Agent → Master Access Control

Rules can be tweaked here

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.


Enable Agent → Master Access Control
Rules can be tweaked here

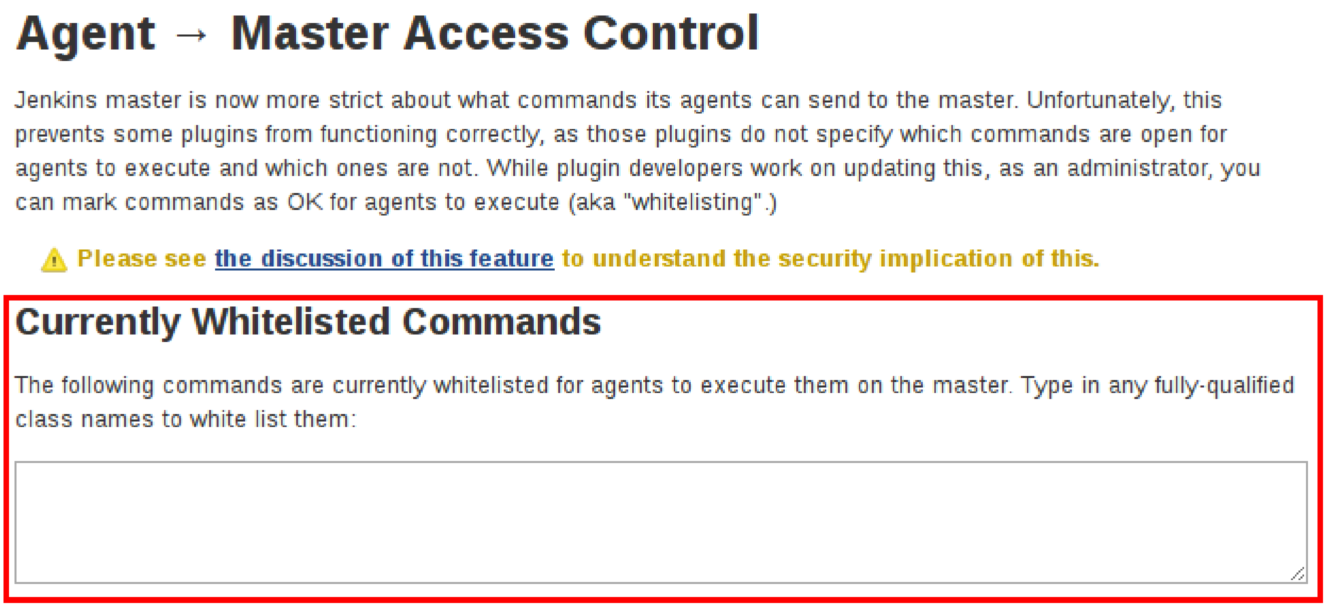
By following the link highlighted above, an administrator may edit **Commands** and **File Access** Agent/Master access control rules.

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" [26: en.wikipedia.org/wiki/Whitelist] certain commands as acceptable for execution on the master.



Agent → Master Access Control

Jenkins master is now more strict about what commands its agents can send to the master. Unfortunately, this prevents some plugins from functioning correctly, as those plugins do not specify which commands are open for agents to execute and which ones are not. While plugin developers work on updating this, as an administrator, you can mark commands as OK for agents to execute (aka "whitelisting".)

Please see the discussion of this feature to understand the security implication of this.

Currently Whitelisted Commands

The following commands are currently whitelisted for agents to execute them on the master. Type in any fully-qualified class names to white list them:

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

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.

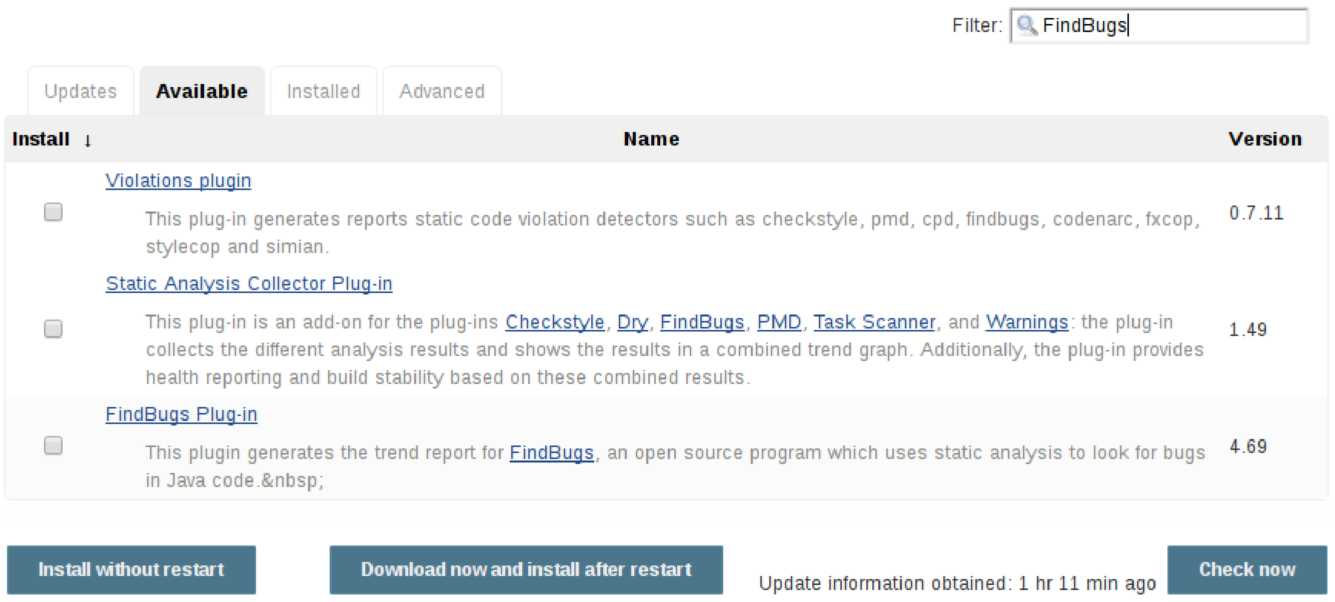
The two approaches require that the Jenkins master be able to download meta-data from an Update Center, whether the primary Update Center operated by the Jenkins project [27: updates.jenkins.io], or a custom Update Center.

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:


Filter: FindBugs

Updates | **Available** | Installed | Advanced

Install ↓	Name	Version
☐	Violations plugin This plug-in generates reports static code violation detectors such as checkstyle, pmd, cpd, findbugs, codenarc, fxcop, stylecop and simian.	0.7.11
☐	Static Analysis Collector Plug-in This plug-in is an add-on for the plug-ins Checkstyle, Dry, FindBugs, PMD, Task Scanner, and Warnings: the plug-in collects the different analysis results and shows the results in a combined trend graph. Additionally, the plug-in provides health reporting and build stability based on these combined results.	1.49
☐	FindBugs Plug-in This plugin generates the trend report for FindBugs, an open source program which uses static analysis to look for bugs in Java code.	4.69

Install without restart
Download now and install after restart
Update information obtained: 1 hr 11 min ago
Check now

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.
```

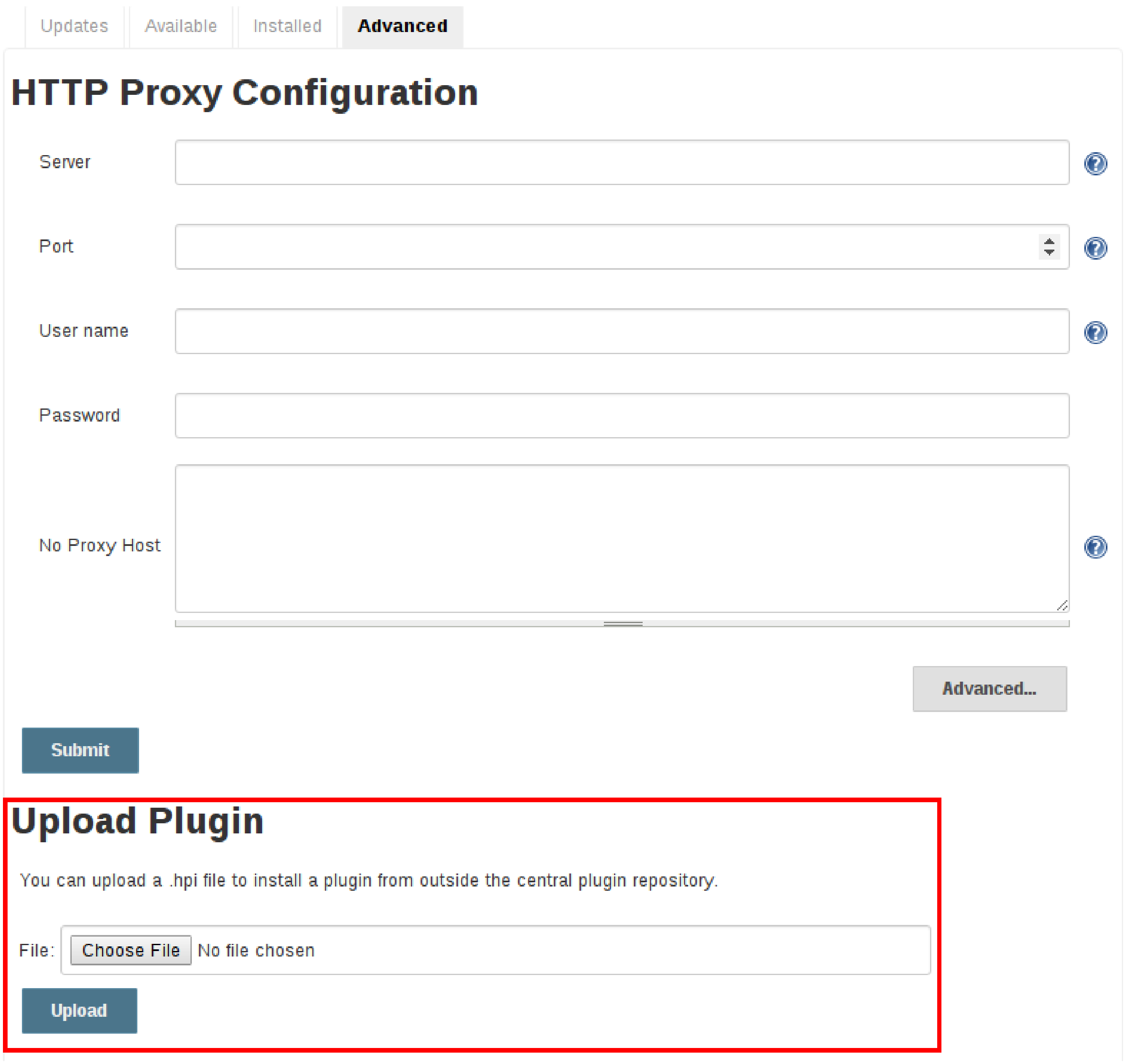
Advanced installation

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.



You can upload a .hpi file to install a plugin from outside the central plugin repository.

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.

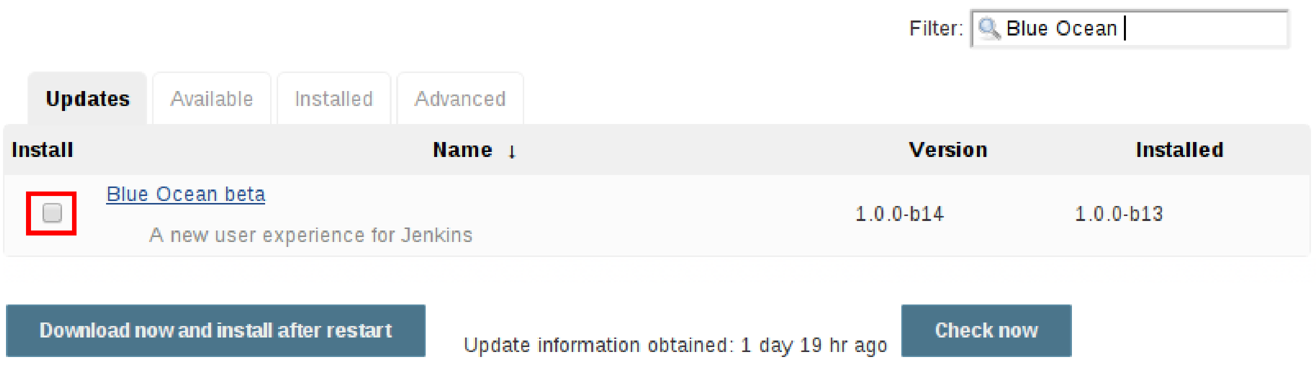
Copy the downloaded `.hpi`` file into the `JENKINS_HOME/plugins` directory on the Jenkins master (for example, on Debian systems `JENKINS_HOME` is generally `/var/lib/jenkins`).

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 [27: 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.



Install	Name ↓	Version	Installed
☐	Blue Ocean beta A new user experience for Jenkins	1.0.0-b14	1.0.0-b13

Update information obtained: 1 day 19 hr ago

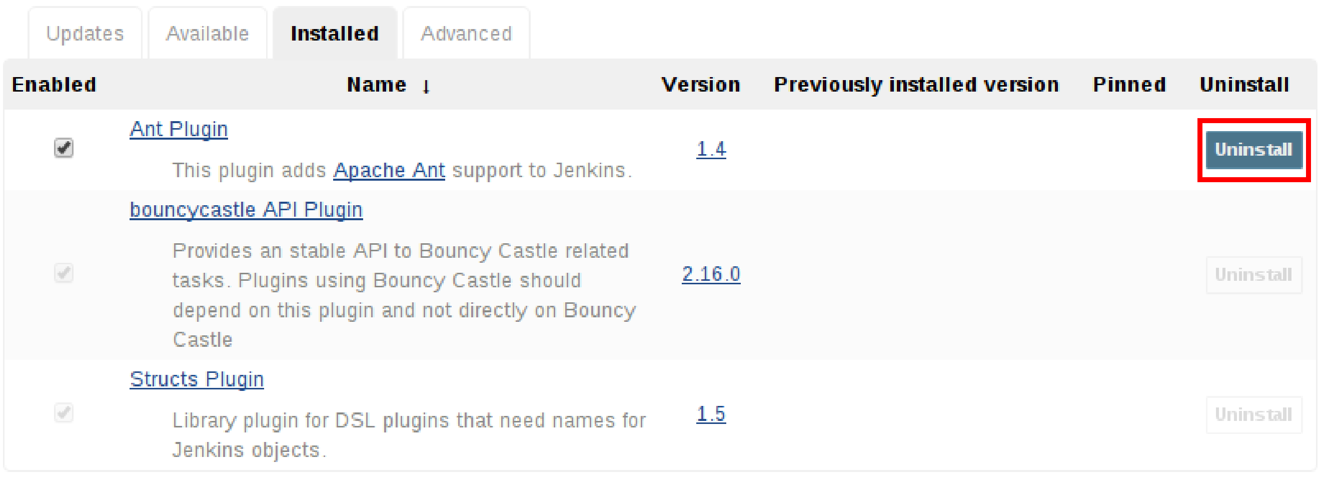
By default, the Jenkins master will check for updates from the Update Center once every 24 hours. To manually trigger a check for updates, simply click on the **Check now** button in the **Updates** tab.

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.



Enabled	Name ↓	Version	Previously installed version	Pinned	Uninstall
✓	Ant Plugin This plugin adds Apache Ant support to Jenkins.	1.4			Uninstall
✓	bouncycastle API Plugin Provides an stable API to Bouncy Castle related tasks. Plugins using Bouncy Castle should depend on this plugin and not directly on Bouncy Castle	2.16.0			Uninstall
✓	Structs Plugin Library plugin for DSL plugins that need names for Jenkins objects.	1.5			Uninstall

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.

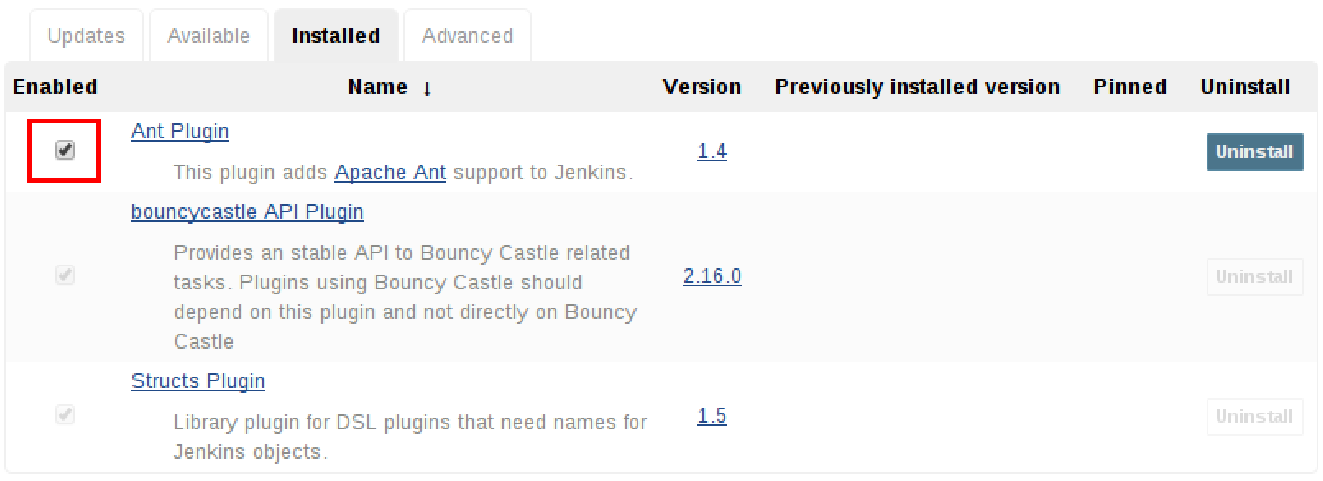
Removing old data

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).


Updates | Available | **Installed** | Advanced

Enabled	Name ↓	Version	Previously installed version	Pinned	Uninstall
✓	Ant Plugin This plugin adds Apache Ant support to Jenkins.	1.4			Uninstall
✓	bouncycastle API Plugin Provides an stable API to Bouncy Castle related tasks. Plugins using Bouncy Castle should depend on this plugin and not directly on Bouncy Castle	2.16.0			Uninstall
✓	Structs Plugin Library plugin for DSL plugins that need names for Jenkins objects.	1.5			Uninstall

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.

Using the Jenkins CLI

It is also possible to enable or disable plugins via the Jenkins CLI using the `enable-plugin` or `disable-plugin` commands.

NOTE The `enable-plugin` command was added to Jenkins in v2.136. The `disable-plugin` command was added to Jenkins in v2.151.

The `enable-plugin` command receives a list of plugins to be enabled. Any plugins which a selected plugin depends on will also be enabled by this command.

```
java -jar jenkins-cli.jar -s http://localhost:8080/ enable-plugin PLUGIN ... [-
restart]

Enables one or more installed plugins transitively.

 PLUGIN   : Enables the plugins with the given short names and their
            dependencies.
 -restart : Restart Jenkins after enabling plugins.
```

The `disable-plugin` command receives a list of plugins to be disabled. The output will display messages for both successful and failed operations. If you only want to see error messages, the `-quiet` option can be specified. The `-strategy` option controls what action will be taken when one of the specified plugins is listed as an optional or mandatory dependency of another enabled plugin.

```
java -jar jenkins-cli.jar -s http://localhost:8080/ disable-plugin PLUGIN ... [-quiet
(-q)]
[-restart (-r)] [-strategy (-s) strategy]

Disable one or more installed plugins.
Disable the plugins with the given short names. You can define how to proceed with the
dependant plugins and if a restart after should be done. You can also set the quiet
mode
to avoid extra info in the console.

 PLUGIN                    : Plugins to be disabled.
 -quiet (-q)               : Be quiet, print only the error messages
 -restart (-r)             : Restart Jenkins after disabling plugins.
 -strategy (-s) strategy : How to process the dependant plugins.
                             - none: if a mandatory dependant plugin exists and
                             it is enabled, the plugin cannot be disabled
                             (default value).
                             - mandatory: all mandatory dependant plugins are
                             also disabled, optional dependant plugins remain
                             enabled.
                             - all: all dependant plugins are also disabled, no
                             matter if its dependency is optional or mandatory.
```

CAUTION In the same way than enabling and disabling plugins from the UI requires a restart to complete the process, the changes made with the CLI commands will take effect once Jenkins is restarted. The `-restart` option forces a safe restart of the instance once the command has successfully finished, so the changes will be immediately applied.

Pinned plugins

CAUTION Pinned plugins feature was removed in Jenkins 2.0. Versions later than Jenkins 2.0 do not bundle plugins, instead providing a wizard to install the most useful plugins.

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

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

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

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

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

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

Jenkins CLI

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

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

WARNING

Use of the CLI client distributed with Jenkins 2.53 and older and Jenkins LTS 2.46.1 and older is **not recommended** for security reasons: while there are no currently known vulnerabilities, several have been reported and patched in the past, and the Jenkins Remoting protocol it uses is inherently vulnerable to remote code execution bugs, even “preauthentication” exploits (by anonymous users able to physically access the Jenkins network).

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

Using the CLI over SSH

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

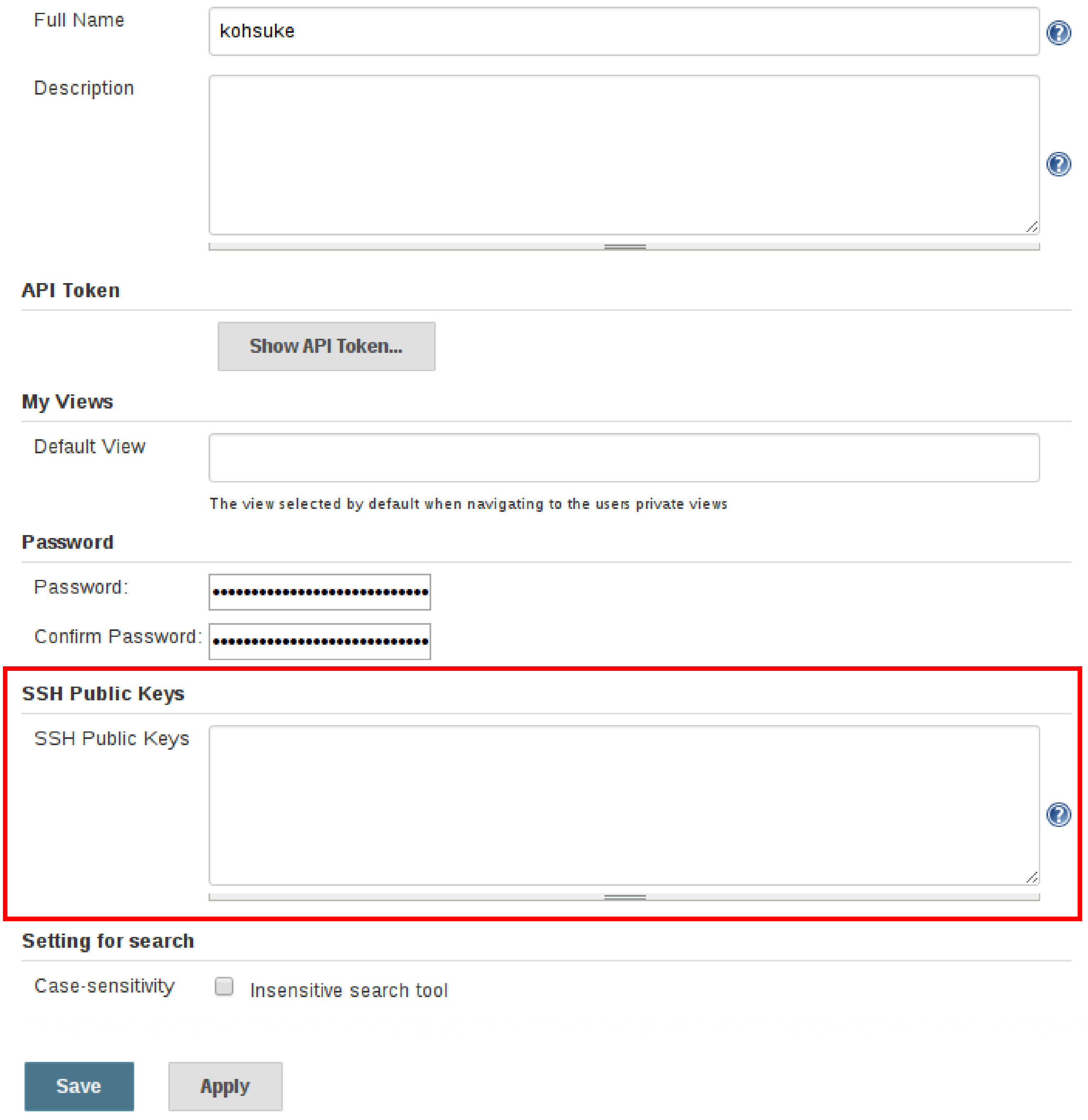
```
% curl -Lv https://JENKINS_URL/login 2>&1 | grep 'X-SSH-Endpoint'
< X-SSH-Endpoint: localhost:53801
%
```

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

Authentication

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

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


Full Name: kohsuke

Description:

API Token

My Views

Default View:

The view selected by default when navigating to the users private views

Password

Password: ••••••••••••••••••••••••••••

Confirm Password: ••••••••••••••••••••••••••••

SSH Public Keys

SSH Public Keys:

Setting for search

Case-sensitivity: ☐ Insensitive search tool

Common Commands

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

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

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

build

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

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

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

java -jar jenkins-cli.jar build JOB [-c] [-f] [-p] [-r N] [-s] [-v] [-w]
Starts a build, and optionally waits for a completion.  Aside from general
scripting use, this command can be used to invoke another job from within a
build of one job.  With the -s option, this command changes the exit code based
on the outcome of the build (exit code 0 indicates a success) and interrupting
the command will interrupt the job.  With the -f option, this command changes
the exit code based on the outcome of the build (exit code 0 indicates a
success) however, unlike -s, interrupting the command will not interrupt the
job (exit code 125 indicates the command was interrupted).  With the -c option,
a build will only run if there has been an SCM change.
 JOB : Name of the job to build
 -c  : Check for SCM changes before starting the build, and if there's no
       change, exit without doing a build
 -f  : Follow the build progress. Like -s only interrupts are not passed
       through to the build.
 -p  : Specify the build parameters in the key=value format.
 -s  : Wait until the completion/abortion of the command. Interrupts are passed
       through to the build.
 -v  : Prints out the console output of the build. Use with -s
 -w  : Wait until the start of the command
% ssh -l kohsuke -p 53801 localhost build build-all-software -f -v
Started build-all-software #1
Started from command line by admin
Building in workspace /tmp/jenkins/workspace/build-all-software
[build-all-software] $ /bin/sh -xe /tmp/hudson1100603797526301795.sh
+ echo hello world
hello world
Finished: SUCCESS
Completed build-all-software #1 : SUCCESS
%
```

console

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

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

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

who-am-i

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

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

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

Using the CLI client

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

Downloading the client

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

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

Using the client

The general syntax for invoking the client is as follows:

```
java -jar jenkins-cli.jar [-s JENKINS_URL] [global options...] command [command
options...] [arguments...]
```

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

Client connection modes

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

HTTP connection mode

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

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

```
java -jar jenkins-cli.jar [-s JENKINS_URL] -auth kohsuke:abc1234ffe4a command ...
```

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

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

```
java -jar jenkins-cli.jar [-s JENKINS_URL] -auth @/home/kohsuke/.jenkins-cli command
...
```

WARNING For security reasons the use of a file to load the authentication credentials is the recommended authentication way.

An alternative authentication method is to configure environment variables in a similar way as the `$JENKINS_URL` is used. The `username` can be specified via the environment variable `$JENKINS_USER_ID` while the `apitoken` can be specified via the variable `$JENKINS_API_TOKEN`. Both variables have to be set all at once.

```
export JENKINS_USER_ID=kohsuke
export JENKINS_API_TOKEN=abc1234ffe4a
java -jar jenkins-cli.jar [-s JENKINS_URL] command ...
```

In case these environment variables are configured you could still override the authentication method using different credentials with the `-auth` option, which takes preference over them.

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

WARNING The HTTP(S) connection mode of the CLI in Jenkins 2.54 and newer does not work correctly behind an Apache HTTP reverse proxy server using mod_proxy. Workarounds include using a different reverse proxy such as Nginx or HAProxy, or using the SSH connection mode where possible. See JENKINS-47279.

SSH connection mode

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

```
java -jar jenkins-cli.jar [-s JENKINS_URL] -ssh -user kohsuke command ...
```

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

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

Remoting connection mode

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

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

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

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

Common Problems with the Remoting-based client

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

Operation timed out

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

```
% java -jar jenkins-cli.jar -s JENKINS_URL help
Exception in thread "main" java.net.ConnectException: Operation timed out
    at java.net.PlainSocketImpl.socketConnect(Native Method)
    at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:351)
    at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:213)
    at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:200)
    at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:432)
    at java.net.Socket.connect(Socket.java:529)
    at java.net.Socket.connect(Socket.java:478)
    at java.net.Socket.<init>(Socket.java:375)
    at java.net.Socket.<init>(Socket.java:189)
    at hudson.cli.CLI.<init>(CLI.java:97)
    at hudson.cli.CLI.<init>(CLI.java:82)
    at hudson.cli.CLI._main(CLI.java:250)
    at hudson.cli.CLI.main(CLI.java:199)
```

No X-Jenkins-CLI2-Port

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

```
java.io.IOException: No X-Jenkins-CLI2-Port among [X-Jenkins, null, Server, X-Content-
Type-Options, Connection,
        X-You-Are-In-Group, X-Hudson, X-Permission-Implied-By, Date, X-Jenkins-
Session, X-You-Are-Authenticated-As,
        X-Required-Permission, Set-Cookie, Expires, Content-Length, Content-Type]
    at hudson.cli.CLI.getCliTcpPort(CLI.java:284)
    at hudson.cli.CLI.<init>(CLI.java:128)
    at hudson.cli.CLIConnectionFactory.connect(CLIConnectionFactory.java:72)
    at hudson.cli.CLI._main(CLI.java:473)
    at hudson.cli.CLI.main(CLI.java:384)
    Suppressed: java.io.IOException: Server returned HTTP response code: 403 for URL:
http://citest.gce.px/cli
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.java:184
0)
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1441
)
        at hudson.cli.FullDuplexHttpStream.<init>(FullDuplexHttpStream.java:78)
        at hudson.cli.CLI.connectViaHttp(CLI.java:152)
        at hudson.cli.CLI.<init>(CLI.java:132)
        ... 3 more
```

Server key did not validate

You may get the error below and find a log entry just below that concerning `mismatched keys`:

```
org.apache.sshd.common.SshException: Server key did not validate
    at
org.apache.sshd.client.session.AbstractClientSession.checkKeys(AbstractClientSession.j
ava:523)
    at
org.apache.sshd.common.session.helpers.AbstractSession.handleKexMessage(AbstractSessio
n.java:616)
    ...
```

This means your SSH configuration does not recognize the public key presented by the server. It's often the case when you run Jenkins in dev mode and multiple instances of the application are run under the same SSH port over time.

In a development context, access your `~/.ssh/known_hosts` (or in `C:/Users/<your_name>/.ssh/known_hosts` for Windows) and remove the line corresponding to your current SSH port (e.g. `[localhost]:3485`). In a production context, check with the Jenkins administrator if the public key of the server changed recently. If so, ask the administrator to do the the steps described above.

UsernameNotFoundException

If your client displays a stacktrace that looks like:

```
org.acegisecurity.userdetails.UsernameNotFoundException: <name_you_used>
    ...
```

This means your SSH keys were recognized and validated against the stored users but the username is not valid for the security realm your application is using at the moment. This could occur when you were using the Jenkins database initially, configured your users, and then switched to another security realm (like LDAP, etc.) where the defined users do not exist yet.

To solve the problem, ensure your users exist in your configured security realm.

Troubleshooting logs

To get more information about the authentication process:

1. Go into **Manage Jenkins** > **System Log** > **Add new log recorder**.
2. Enter any name you want and click on **Ok**.
3. Click on **Add**
4. Type `org.jenkinsci.main.modules.sshd.PublicKeyAuthenticatorImpl` (or type `PublicKeyAuth` and then select the full name)
5. Set the level to **ALL**.
6. Repeat the previous three steps for `hudson.model.User`
7. Click on **Save**

When you try to authenticate, you can then refresh the page and see what happen internally.

Script Console

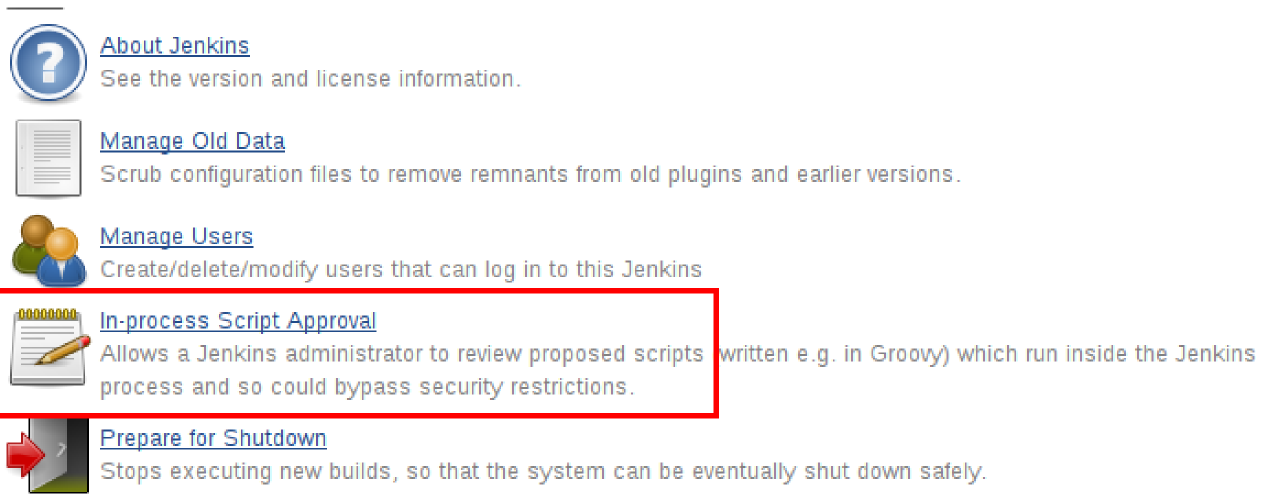
Managing Nodes

In-process Script Approval

Jenkins, and a number of plugins, allow users to execute Groovy scripts *in* Jenkins. These scripting capabilities are provided by:

- Script Console.
- Jenkins Pipeline.
- The plugin:email-ext[Extended Email plugin].
- The plugin:groovy[Groovy plugin] - when using the "Execute system Groovy script" step.
- The plugin:job-dsl[JobDSL plugin] as of version 1.60 and later.

To protect Jenkins from execution of malicious scripts, these plugins execute user-provided scripts in a Groovy Sandbox that limits what internal APIs are accessible. Administrators can then use the "In-process Script Approval" page, provided by the plugin:script-security[Script Security plugin], to manage which unsafe methods, if any, should be allowed in the Jenkins environment.


About Jenkins
See the version and license information.

Manage Old Data
Scrub configuration files to remove remnants from old plugins and earlier versions.

Manage Users
Create/delete/modify users that can log in to this Jenkins

In-process Script Approval
Allows a Jenkins administrator to review proposed scripts (written e.g. in Groovy) which run inside the Jenkins process and so could bypass security restrictions.

Prepare for Shutdown
Stops executing new builds, so that the system can be eventually shut down safely.

Getting Started

The plugin:script-security[Script Security plugin] is installed automatically by the Post-install Setup Wizard, although initially no additional scripts or operations are approved for use.

IMPORTANT	Older versions of this plugin may not be safe to use. Please review the security warnings listed on plugin:script-security[the Script Security plugin page] in order to ensure that the plugin:script-security[Script Security plugin] is up to date.

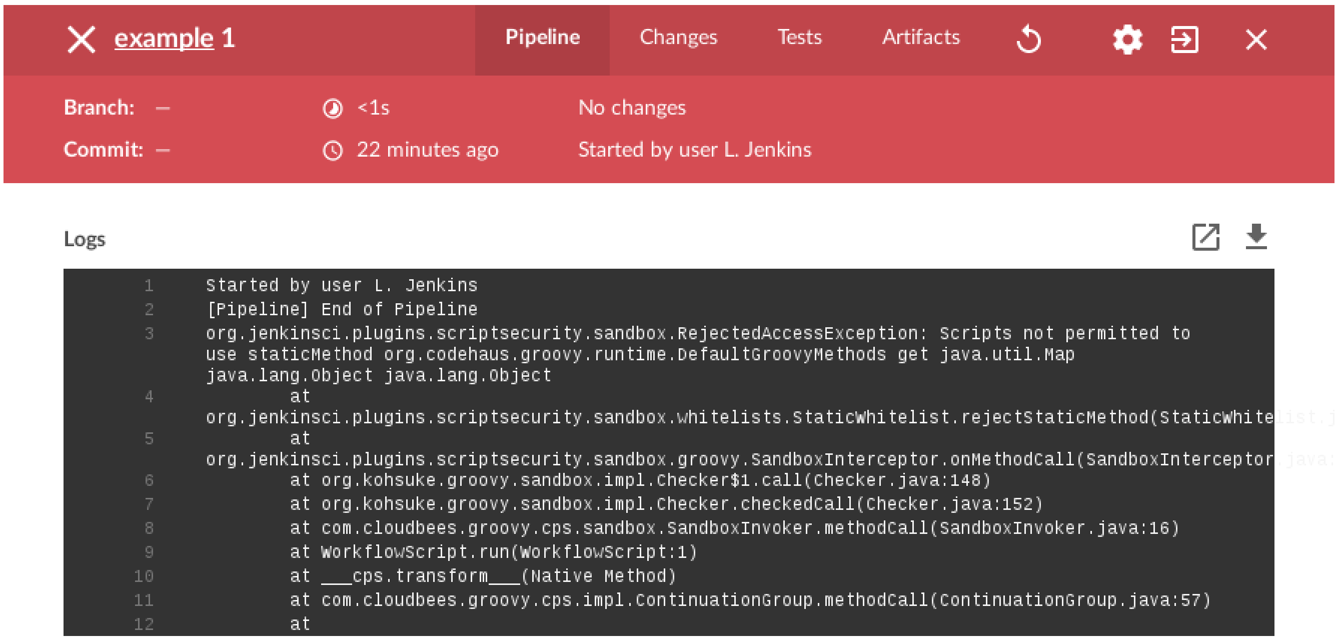
Security for in-process scripting is provided by two different mechanisms: the Groovy Sandbox and Script Approval. The first, the Groovy Sandbox, is enabled by default for Jenkins Pipeline allowing user-supplied Scripted and Declarative Pipeline to execute without prior Administrator intervention. The second, Script Approval, allows Administrators to approve or deny unsandboxed scripts, or allow sandboxed scripts to execute additional methods.

For most instances, the combination of the Groovy Sandbox and the Script Security's built-in list of approved method signatures, will be sufficient. It is strongly recommended that Administrators only deviate from these defaults if absolutely necessary.

Groovy Sandbox

To reduce manual interventions by Administrators, most scripts will run in a Groovy Sandbox by default, including all Jenkins Pipelines. The sandbox only allows a subset of Groovy's methods deemed sufficiently safe for "untrusted" access to be executed without prior approval. Scripts using the Groovy Sandbox are **all** subject to the same restrictions, therefore a Pipeline authored by an Administrator is subject to the restrictions as one authorized by a non-administrative user.

When a script attempts to use features or methods unauthorized by the sandbox, a script is halted immediately, as shown below with Jenkins Pipeline



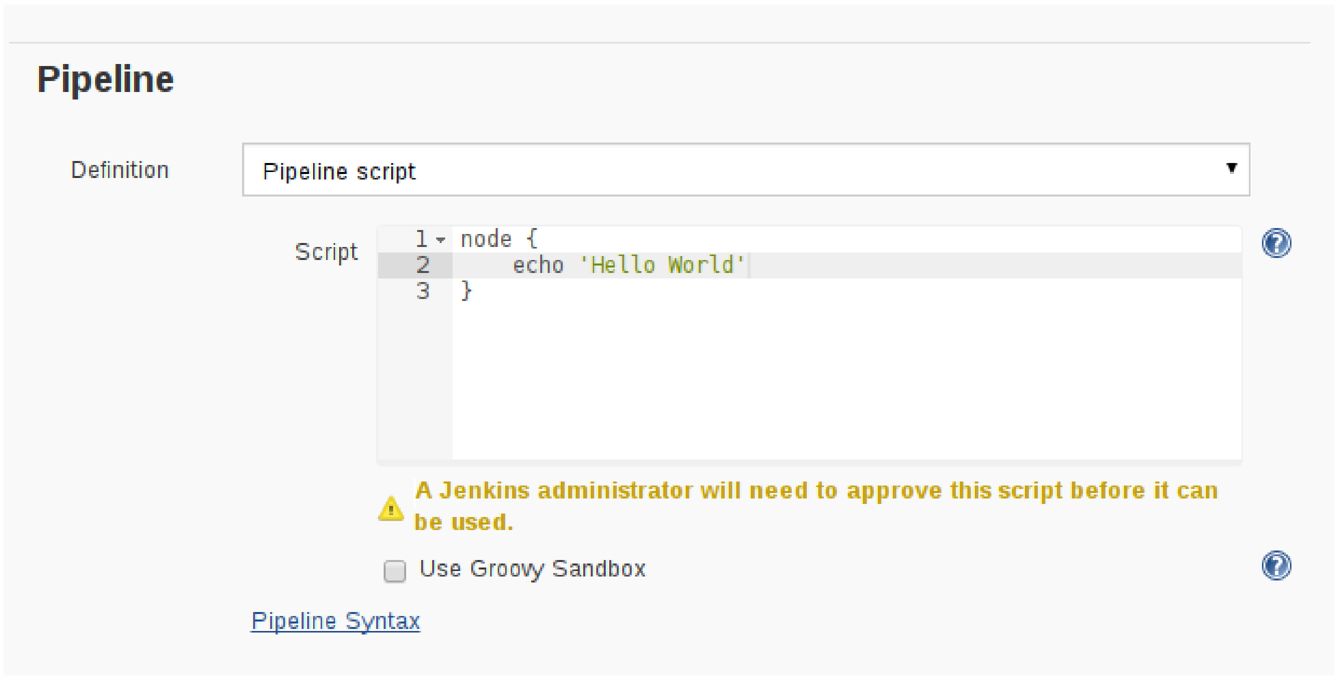
Branch: —
Commit: —
<1s
22 minutes ago
No changes
Started by user L. Jenkins

```
Started by user L. Jenkins
[Pipeline] End of Pipeline
org.jenkinsci.plugins.scriptsecurity.sandbox.RejectedAccessException: Scripts not permitted to
use staticMethod org.codehaus.groovy.runtime.DefaultGroovyMethods get java.util.Map
java.lang.Object java.lang.Object
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.whitelists.StaticWhitelist.rejectStaticMethod(StaticWhitelist.j
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.groovy.SandboxInterceptor.onMethodCall(SandboxInterceptor.java:
        at org.kohsuke.groovy.sandbox.impl.Checker$1.call(Checker.java:148)
        at org.kohsuke.groovy.sandbox.impl.Checker.checkedCall(Checker.java:152)
        at com.cloudbees.groovy.cps.sandbox.SandboxInvoker.methodCall(SandboxInvoker.java:16)
        at WorkflowScript.run(WorkflowScript:1)
        at ___cps.transform___(Native Method)
        at com.cloudbees.groovy.cps.impl.ContinuationGroup.methodCall(ContinuationGroup.java:57)
        at
```

Figure 2. Unauthorized method signature rejected at runtime via Blue Ocean

The Pipeline above will not execute until an Administrator approves the method signature via the **In-process Script Approval** page.

In addition to adding approved method signatures, users may also disable the Groovy Sandbox entirely as shown below. Disabling the Groovy Sandbox requires that the **entire** script must be reviewed and manually approved by an administrator.


Pipeline

Definition: Pipeline script

Script:
```
node {
    echo 'Hello World'
}
```

A Jenkins administrator will need to approve this script before it can be used.

Use Groovy Sandbox

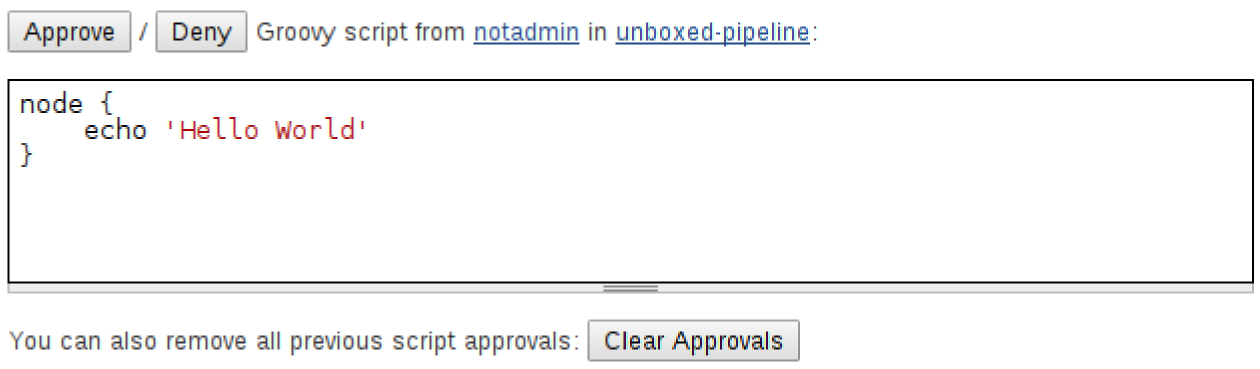
Pipeline Syntax

Figure 3. Disabling the Groovy Sandbox for a Pipeline

Script Approval

Manual approval of entire scripts, or method signatures, by an administrator provides Administrators with additional flexibility to support more advanced usages of in-process scripting. When the Groovy Sandbox is disabled, or a method outside of the built-in list is invoked, the Script Security plugin will check the Administrator-managed list of approved scripts and methods.

For scripts which wish to execute outside of the Groovy Sandbox, the Administrator must approve the **entire** script in the **In-process Script Approval** page:

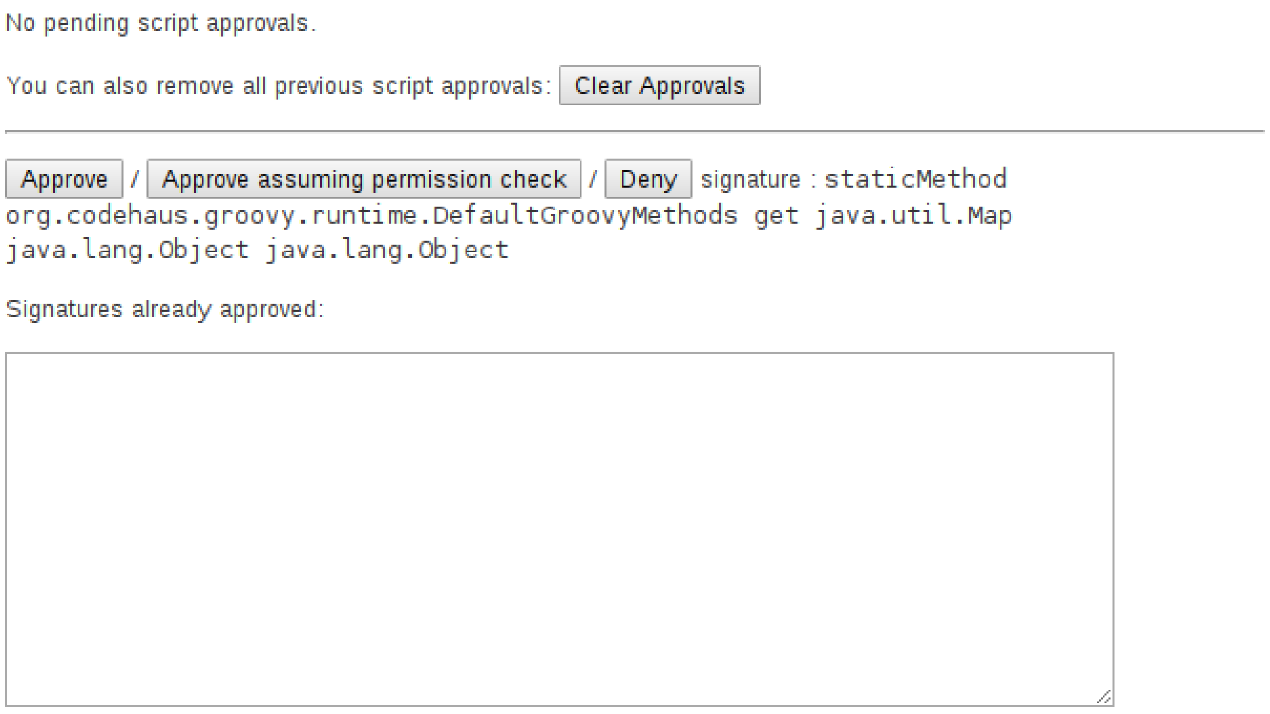

Approve / Deny Groovy script from notadmin in unboxed-pipeline:

```
node {
    echo 'Hello World'
}
```

You can also remove all previous script approvals: Clear Approvals

Figure 4. Approving an unsandboxed Scripted Pipeline

For scripts which use the Groovy Sandbox, but wish to execute an currently unapproved method signature will also be halted by Jenkins, and require an Administrator to approve the specific method signature before the script is allowed to execute:


No pending script approvals.

You can also remove all previous script approvals:

signature : staticMethod org.codehaus.groovy.runtime.DefaultGroovyMethods get java.util.Map java.lang.Object java.lang.Object

Signatures already approved:

Figure 5. Approving a new method signature

Approve assuming permissions check

Script approval provides three options: Approve, Deny, and "Approve assuming permissions check." While the purpose of the first two are self-evident, the third requires some additional understanding of what internal data scripts are able to access and how permissions checks inside of Jenkins function.

Consider a script which accesses the method `hudson.model.AbstractItem.getParent()`, which by itself is harmless and will return an object containing either the folder or root item which contains the currently executing Pipeline or Job. Following that method invocation, executing `hudson.model.ItemGroup.getItems()`, which will list items in the folder or root item, requires the `Job/Read` permission.

This could mean that approving the `hudson.model.ItemGroup.getItems()` method signature would allow a script to bypass built-in permissions checks.

Instead, it is usually more desirable to click **Approve assuming permissions check** which will cause the Script Approval engine to allow the method signature assuming the user running the script has the permissions to execute the method, such as the `Job/Read` permission in this example.

Managing Users

System Administration

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

For an overview of content in the Jenkins User Handbook, see User Handbook overview.

Backing-up/Restoring Jenkins

Monitoring Jenkins

Securing Jenkins

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 setup
- Standard Security Setup --- discusses the most common setup of letting Jenkins run its own user database and do finer-grained access control
- Apache frontend for security --- run Jenkins behind Apache and perform access control in Apache instead of Jenkins
- Authenticating scripted clients --- if you need to programmatically access security-enabled Jenkins web UI, use BASIC auth
- Matrix-based security|Matrix-based security --- Granting and denying finer-grained permissions

Protect users of Jenkins from other threats

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

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

- CSRF Protection --- prevent a remote attack against Jenkins running inside your firewall. This feature is **off by default** in Jenkins 1.x and when upgrading to 2.x.

- Security implication of building on master --- protect Jenkins master from malicious builds
- Slave To Master Access Control --- protect Jenkins master from malicious build agents
- Securing JENKINS_HOME --- protect Jenkins from users with local access

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

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

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

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

Managing Jenkins with Chef

Managing Jenkins with Puppet

Scaling Jenkins

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

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

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

For an overview of content in the Jenkins User Handbook, see User Handbook overview.

Appendix A: 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.

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.

Label

User-defined text for grouping Agents, typically by similar functionality or capability. For example `linux` for Linux-based agents or `docker` for Docker-capable agents.

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.

Stage
`stage` is part of Pipeline, and used 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.

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.