# **Jenkins**

# **Installation:**

* Jenkins can run stand-alone in its own process using its own built-in web server (jetty)
* It can also run as one servlet in an existing framework, such as Tomcat or Glassfish application servers.
* Version 1.1.0 and 2.0
* **/var/log/jenkins/jenkins.log =>** Check this file if you are troubleshooting Jenkins.
* 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.
* **/var/lib/Jenkins/secrets/initialAdminPassword** => location where the default admin password (initial) for Jenkins

# **Managing Jenkins**

## **Managing Security:**

## **Enabling security:**

* By default, from the Jenkins 2.0 security is checked or enabled (Enable Security in configure global security of Jenkins UI)

### JNLP TCP Port:

Jenkins uses a TCP port to communicate with agents launched via the JNLP (Java N/W Launch Protocol) 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:

**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.

**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:** primary mechanism for securing the Jenkins environment from unauthorized usage.

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

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

### 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 (CSRF):

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 a malicious actor to delete projects, alter builds, or modify Jenkins' system configuration. To guard against this class of vulnerabilities, CSRF protection has been enabled by default with all Jenkins versions since 2.0.

When the option is enabled, Jenkins will check for a CSRF token, or "crumb", on any request that may change data in the Jenkins environment. This includes any form submission and calls to the remote API, including those using "Basic" authentication. It is strongly recommended that this option be left enabled, including on instances operating on private, fully trusted networks.

**Caveats**

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

* Some Jenkins features, like the remote API, are more difficult to use when this option is enabled.
* 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.

**Plugins:**

ldap[LDAP plugin]

Plugin:active-directory[Active Directory]

plugin:github-oauth[GitHub Authentication]

plugin:crowd2[Atlassian Crowd 2]

plugin:matrix-auth[Matrix Authorization Strategy Plugin] => provides matrix based and project authorization for users.

### Agent/Master Access Control

The Agent/Master Access Control system was introduced to allow Jenkins administrators to add more granular access control definitions between the Jenkins master and the connected agents.

* As of Jenkins 2.0, this option has been enabled or turned on by default

**Customizing Access**

* 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.

##### **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.

## **Managing Tools**

The built-in tool providers are

* Ant
* Git
* JDK
* Maven

## **Managing Plugins**

* There are over thousand different plugins in Jenkins

### Installing Plugin

Jenkins provides a couple of different methods for installing plugins.

* Using plugin manager (from the Jenkins web UI)
* Using Jenkins CLI install-plugin command

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

**Using Jenkins 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

### 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.

#### **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.

## **Updating a plugin**

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

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

## **Removing a plugin**

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

### Uninstalling a plugin

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

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

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.

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

**Pinned Plugins:**

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

The notion of **pinned plugins** applies to plugins that are bundled with Jenkins 1.x, such as the Matrix Authorization plugin.

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

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

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

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

The Plugin Manager allows plugins to be explicitly unpinned.

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

## **Jenkins CLI**

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

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

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:

https://JENKINS\_URL/user/USERNAME/configure and paste an SSH public key into the appropriate text area.

### Common Commands

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

* % ssh -l kohsuke -p 53801 localhost help

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

#### **build**

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

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

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

#### **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

**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

## **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 https://JENKINS\_URL/jnlpJars/jenkins-cli.jar

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

### Using the client

The general syntax for invoking the client is as follows:

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

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

### Client connection modes

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

#### **HTTP connection mode**

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

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

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

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

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

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

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

#### **SSH connection mode**

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

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

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

By default, the client will try to connect to an SSH port on the same host as is used in the JENKINS\_URL.

If Jenkins is behind an HTTP reverse proxy, this will not generally work, so run Jenkins with the system property -Dorg.jenkinsci.main.modules.sshd.SSHD.hostName=ACTUALHOST to define a hostname or IP address for the SSH endpoint.

#### **Remoting connection mode**

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

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

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

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

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

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

###### **Operation timed out**

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

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

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

## **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 Extended Email Plugin
* The Groovy Plugin -> when using execute system groovy script step
* The 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 Script Security plugin, to manage which unsafe methods, if any, should be allowed in the Jenkins environment.

The Script Security plugin is installed automatically by the Post-Install Setup Wizard, although initially no additional scripts or operations are approved for use.

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 un-sandboxed 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 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.

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. Disabling the Groovy Sandbox requires that the **entire** script must be reviewed and manually approved by an administrator.

## **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.

For scripts which use the Groovy Sandbox, but wish to execute a 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

### 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.

### Webhooks vs Polling

**What are webhooks, and why should you be using them?**

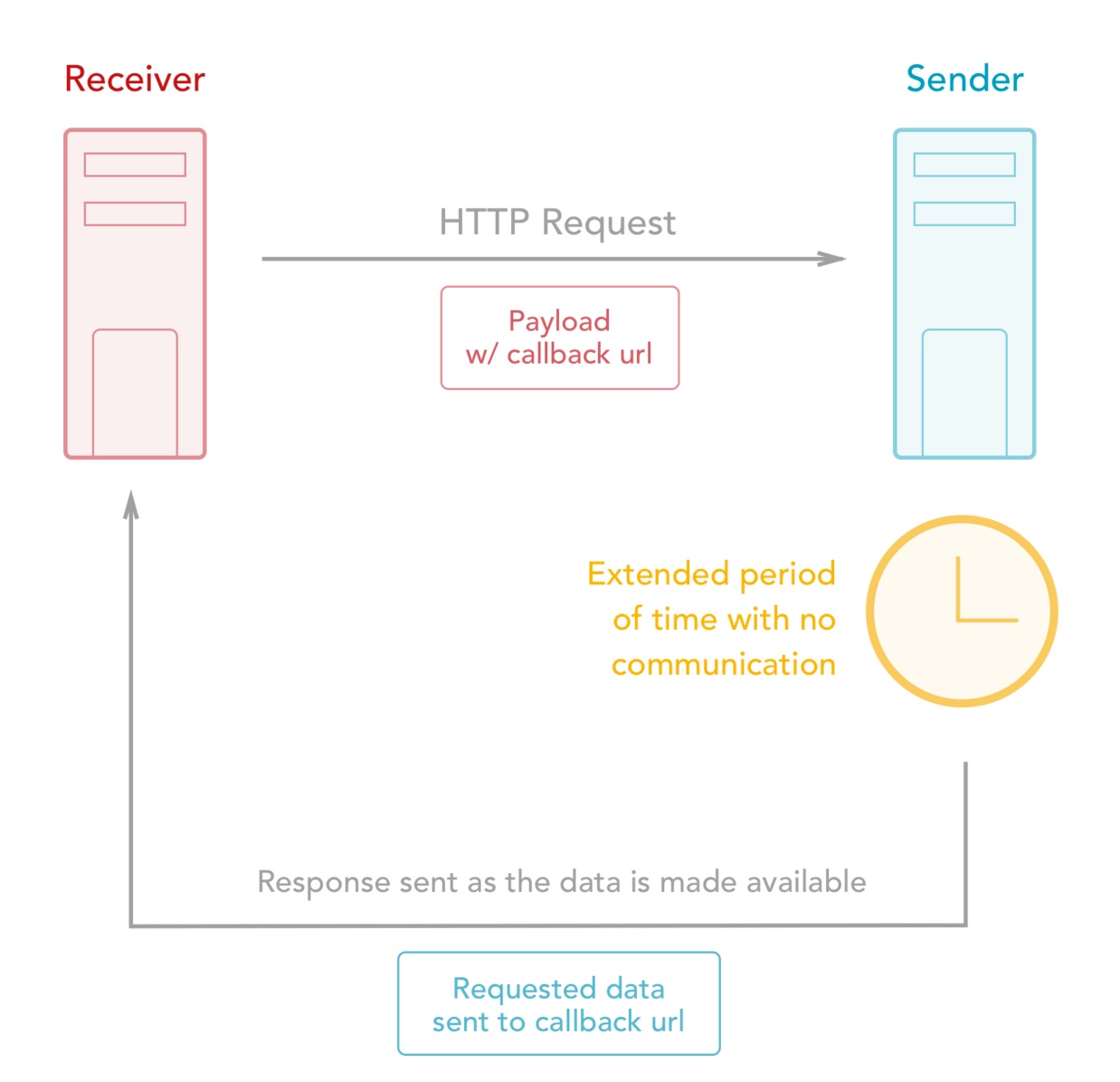
A webhook is a way to deliver real-time data to applications. Unlike traditional APIs where you need to poll for data frequently in order to get quasi real-time information, webhooks send data immediately.

You can think about webhooks like push notifications on your mobile phone. Rather than burning up the battery on your phone fetching information (polling) from applications to get updates, push notifications (webhooks) automatically send data based on event triggers. And just like push notifications, webhooks are less resource-intensive.

#### **How Do Webhooks Work?**

A webhook is an HTTP request (typically a POST sent to a pre-defined callback URI, where the server application is configured to handle the request on that URI. In many cases, webhooks are triggered by stimulus events, making them a faster and more efficient method of handling events (push-based vs. poll-based).

Webhooks have two components: the outbound side (sender) and the inbound side (recipient). The recipient is responsible for specifying the callback URI so that the sender knows where to send the asynchronous request.

[](https://cloud.githubusercontent.com/assets/1907738/12208877/8618748c-b604-11e5-8d2d-8d43a9ae19a6.jpg)

The **outbound side** is typically associated with some sort of stimulus event that gets triggered in the server code. This could be an event such as a status change on an object like assigning a pull request, or the completion of a process like publishing a release. Think of it as subscribe and notify model where the subscription may be long lasting, or transient – created and expired for the duration of a single stimulus request.

The **inbound side** of a webhook is the location or URI that the webhook is sent to. Recipients are expecting to receive webhooks and will take some form of action based on the contents of the received message. Hence, recipients of a webhook require setup and configuration to receive and handle the inbound request.

#### **Why Use Webhooks Instead of Polling an API?**

Webhooks are far more efficient than polling, from a resource and communication standpoint. Zapier did a study across 30 million poll requests made through their service, and found that 98.5% of polls are wasted and they spent 66x more resources on polling.

Data is always old. The very nature of webhooks and the fact that they are typically event-triggered means they are providing you with near real-time information. Due to this, if you want information as close to real-time as possible, you should elect to use webhooks over polling.

**List of useful plugins in Jenkins (most used)**

* **Clone Workspace SCM**

This plugin makes it possible to archive the workspace from builds of one project and reuse them as the SCM source for another project. This is useful in terms of avoiding space issues, and avoiding copying over the same source code which is not absolutely needed in a couple of cases, resulting in faster execution and feedback.

* **HipChat**

This plugin is a HipChat notifier that can publish build status to HipChat rooms. This plugin add more visibility and awareness in terms of build status for the team, resulting in a better sense of collective ownership.

* **Parameterized Trigger**

This plugin lets you trigger new builds when your build has completed, with various ways of specifying parameters for the new build. You can add multiple configurations: each has a list of projects to trigger, a condition for when to trigger them (based on the result of the current build), and a parameters section.

* **Shared Workspace**

This plugin allows you to share workspaces for Jenkins jobs with the same SCM repos. It saves some disk space and repetitive steps, if you have different jobs with identical repos. The importance of this plugin is not well understood by the developers, considering memory is cheap. But following standard practices and promoting reusability can be fruitful in the future.

* **SSH Agent**

This plugin allows you to provide SSH credentials to builds via a ssh agent in Jenkins.

* **SSH**

You can use the SSH Plugin to run shell commands on a remote machine via ssh.

* **SSH Slaves**

This plugin allows you to manage slaves running on Unix machines over SSH. It adds a new type of slave launch method. This launch method will open a SSH connection to the specified host as the specified username. Once it has a suitable version of java, it copies the latest slave.jar via SFTP. It starts the slave process.

* **Subversion**

This plugin adds the Subversion support (via SVNKit) to Jenkins.

* **Git** => Git SCM support.
* **Multiple SCMs** => Support for multiple SCM stages in freestyle job.
* **Copy Artifact** => Allows to copy artifacts between different builds and jobs.
* **JobConfigHistory** => Allows to store history of changes for system and job configuration files.
* **thinBackup** => Just a backup tool.
* **EnvInject** => Allows to inject and print environment variables.
* **Workspace Cleanup** => Allows to clean workspace before the build.
* **Template Project** => Allows to use SCM/build/publish steps from one job in another.
* **EZ Templates** => Allows to use one job as a template for another (the other job can overwrite default values for job parameters).
* **Green Balls** => Replaces blue balls with green one.
* **Modern Status Alternative** => set of status and action icons.
* **SafeRestart** => Adds SafeRestart button that allows to restart jenkins server.
* **Show Build** => Parameters Shows build parameters on main build page.
* **Rebuild** => Adds Rebuild button.

## **Usefull**

* **CloudBees Folders** => Adds folders support for jobs (jobs can be put into folders).
* **Promoted Builds Simple** => Allows to add custom label to builds.
* **GitLab** => GitLab integration.
* **Conditional BuildStep** => Adds conditional build steps support.
* **Build-timeout** => Allows to specify time-outs for build steps.
* **Role Strategy** => Allows to specify permissions per role and assign users to different roles.
* **NodeLabel** => Parameter Allows to set node/label as job parameter.
* **Heavy Job** => Allows to make job occupy more than one executor.
* **Throttle Concurrent Builds** => Allows to prevent concurrent execution of some jobs.
* **Copy to Slave** => Allows to copy some file from jenkins home directory to slave workspace.
* **Parameter Separator** => Adds separator between jobs parameters.
* **Compact Columns** => More compact columns for showing last success and failure.
* **Console Column** => Provide a fast-path console link available for views.
* **AnsiColor** => Allows to print color output in colsole view.
* **Timestamper** => Add timestamps to console output.
* **Build Trigger** => Badge Show who/what trigger the job.

## **Pipelines**

* **Parameterized Trigger** => Trigger another jobs with parameter.
* **Downstream buildview** => Shows all downstream jobs.
* **Delivery Pipeline** => Extra view for simple jobs pipeline.
* **Build Pipeline** => Extra view for simple jobs pipeline.
* **Multijob** => Job that is a container for another job
* **Build Flow** => Allows to describe build flow in groovy script.
* **Build Flow** **Test Aggregator** => Aggregate test results for Build Flow job.
* **Build Graph View** => Represents build flow job as a graph.
* **Pipeline** => Describe whole build as pipeline script.
* **Pipeline Stage View** => Additional view for Pipeline jobs.

## **Misc**

* **Promoted Builds** => Allows to promote build, mark it and perform some actions.
* **Gravatar** => Gravatar service support.
* **Disk Usage** => Shows disk usage statistics.
* **ArtifactDeployer** => Allows to deploy build artifacts to remote server.
* **Dashboard View** => Additional view for jobs with extra widgets.
* **Build Monitor** => Additional view for jobs.
* **Mail Watcher** => Allows to track changes in job configurations view email.
* **Configuration Slicing** => Allows to edit property for multiple jobs in a time.
* **Claim** => Allows to claim failed build by responsible developer.
* **Clone Workspace SCM** => Allows to clone workspace of one job to another job as SCM step.
* **CMake** => Support for CMake based projects.
* **Ownership** => Manages ownership for slaves and jobs.

## **Publishers**

* **Warnings** => Parses compiler warnings.
* **xUnit** => Parses GTest, CTest, etc. output.
* **Valgrind** => Valgrind run/publish support.
* **Cppcheck** => Cppcheck results publishing.
* **Cobertura** => Code coverage results publishing.
* **Doxygen** => Doxygen output HTML publishing.
* **HTML Publisher** => Custom HTML pages publishing.
* **Build Failure Analyzer** => Generic analyser for console output.
* **Task Scanner** => Searches for TODO/FIXME/BUG/... in code.
* **Flexible Publish** => Adds conditional publishing steps.
* **Image Gallery** => Allows to publish images.
* **Email-ext** => Extended email support.
* **Performance** => Publish performance data.
* **DRY** => Publish CPD results (code duplication).
* **Progress Bar Column** => Adds progress bar column to jobs view.
* **Configure Job Column** => Adds configure link to jobs view.

## **Not tested**

* **Analysis Collector** => Collects results from different analysers.
* **Priority Sorter** => Assign priorities for jobs.
* **CloudBees Docker Custom Build Environment** => Run build in docker environment.
* **Text-finder** => Search console output to mark build as failure.
* **MSBuild** => Support for MSBuild tool.
* **Post build** => Run a post-build task depending on console output.
* **CloudBees Docker Pipeline** => Run Pipeline jobs inside docker containers.
* **Monitoring** => Monitors jenkins server.
* **Nested View** => Group job views into several levels.
* **Build Name Setter** => Sets custom build name.
* **View Job Filters** => Filter jobs in view by regular expressions.
* **Job Restrictions** => Restricts jobs execution (restrict which jobs can be run on which nodes).
* **File System SCM** => Simulate File System as SCM.
* **Gradle** => Invoke Gradle as build step.
* **Windows Exe Runner** =>Run Windows executable as build step.
* **DiskCheck** =>Checks for disk space on the slave machine before starting a build.
* **pre-scm-buildstep** =>Run some build steps prior to SCM checkout.
* **Publish Over FTP** =>Publish artifacts over FTP.
* **Publish Over SSH** =>Publish artifacts over SSH.
* **Copy project link** =>Adds the "Copy project" link into left side panel in the main project page.
* **Custom Job Icon** => Allows to configure a custom icon for each job.
* **Avatar** =>Allows avatar images to be uploaded and associated with Jenkins users.
* **Bulk Builder** =>Trigger multiple builds at once.
* **Build Environment** =>Shows information about the environment of a build and gives the option to compare the environments of two builds.
* **Label Linked Jobs** => Facilitate maintenance when using numerous/complex labels.

# **Pipeline**

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

Mostly, the Jenkins file consists of three basic components or parameters in order to do a build for a small or sample application, they are build, test and deploy.

Typically, this "Pipeline as Code" would be written to a [Jenkinsfile](https://jenkins.io/doc/book/pipeline/jenkinsfile/) and checked into a project’s source control repository, for example:

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'make'

}

}

stage('Test'){

steps {

sh 'make check'

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

}

}

stage('Deploy') {

steps {

sh 'make publish'

}

}

}

}

|  |  |
| --- | --- |
|  | In the above-mentioned sample Jenkins file:   * [agent](https://jenkins.io/doc/book/pipeline/syntax/#agent) indicates that Jenkins should allocate an executor and workspace for this part of the Pipeline. |
|  | * [stage](https://jenkins.io/doc/book/pipeline/syntax/#stage) describes a stage of this Pipeline. |
|  | * [steps](https://jenkins.io/doc/book/pipeline/syntax/#steps) describes the steps to be run in this stage |
|  | * sh executes the given shell command |
|  | * junit is a Pipeline step provided by the JUnit plugin for aggregating test reports |

## **Why Pipeline?**

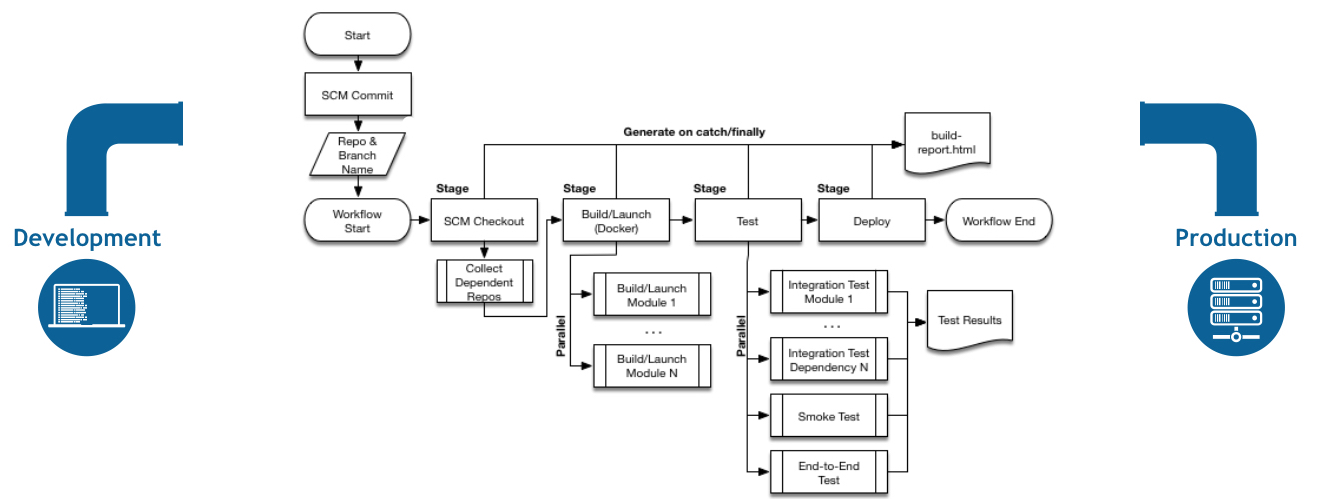
Jenkins is an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive continuous delivery pipelines. By modeling a series of related tasks, users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

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

While Jenkins has always allowed rudimentary forms of chaining Freestyle Jobs together to perform sequential tasks, 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.

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



## **Pipeline Terms**

**Step:**

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

**Node:**

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

* 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.
* Creates a workspace (a directory specific to that particular Pipeline) where work can be done on files checked out from source control.

**Stage**

* stage is a step for defining a conceptually distinct subset of the entire Pipeline, for example: "Build", "Test", and "Deploy", which is used by many plugins to visualize or present Jenkins Pipeline status/progress.

## **Getting Started with Pipeline**

## **Defining a Pipeline**

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

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

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

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

### Defining a Pipeline in SCM

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

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

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

The first line of a Jenkinsfile should be **#!/usr/bin/env** groovy which text editors, IDEs, GitHub, etc will use to syntax highlight the Jenkinsfile properly as Groovy code.

## **Built-in Documentation**

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

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

### Snippet Generator

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

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

To generate a step snippet with the Snippet Generator:

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

To access additional information and/or documentation about the step selected, click on the help icon

## **Using Jenkinsfile**

Creating a Jenkinsfile, which is checked into source control, provides a number of immediate benefits:

* Code review/iteration on the Pipeline
* Audit trail for the Pipeline
* Single source of truth [[2](https://jenkins.io/doc/book/pipeline/jenkinsfile/#_footnote_2)] 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

**Sample Jenkinsfile:**

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building..'

}

}

stage('Test') {

steps {

echo 'Testing..'

}

}

stage('Deploy') {

steps {

echo 'Deploying....'

}

}

}

}

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

Jenkinsfile (Scripted Pipeline)

node {

checkout scm

/\* .. snip .. \*/

}

|  |  |
| --- | --- |
|  | 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.

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'make'

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

}

}

}

}

|  |  |
| --- | --- |
|  | The sh step invokes the make command and will only continue if a zero exit code is returned by the command. Any non-zero exit code will fail the Pipeline. |
|  | archiveArtifacts captures the files built matching the include pattern (\*\*/target/\*.jar) and saves them to the Jenkins master for later retrieval. |
|  | **Note:** Archiving artifacts is not a substitute for using external artifact repositories such as Artifactory or Nexus and should be considered only for basic reporting and file archival. |

### Test

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

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

Jenkinsfile (Declarative Pipeline)

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'

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

}

}

}

}

|  |  |
| --- | --- |
|  | Using an inline shell conditional (sh 'make || true') ensures that the sh step always sees a zero exit code, giving the junit step the opportunity to capture and process the test reports. Alternative approaches to this are covered in more detail in the Handling Failures section below. |
|  | junit captures and associates the JUnit XML files matching the inclusion pattern (\*\*/target/\*.xml) |

### Deploy

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

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

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

stages {

stage('Deploy') {

when {

expression {

currentBuild.result == null || currentBuild.result == 'SUCCESS'

}

}

steps {

sh 'make publish'

}

}

}

}

|  |  |
| --- | --- |
|  | 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.

## **Advanced Syntax for Pipeline**

### String Interpolation

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

**def** singlyQuoted = 'Hello'

**def** doublyQuoted = "World"

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

**def** username = 'Jenkins'

echo 'Hello Mr. ${username}'

echo "I said, Hello Mr. **${**username**}**"

### Working with the Environment

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

**BUILD\_ID**

The current build ID, identical to BUILD\_NUMBER for builds created in Jenkins versions 1.597+

**JOB\_NAME**

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

**JENKINS\_URL**

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

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

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

stages {

stage('Example') {

steps {

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.

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

environment {

CC = 'clang'

}

stages {

stage('Example') {

environment {

DEBUG\_FLAGS = '-g'

}

steps {

sh 'printenv'

}

}

}

}

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

* An environment directive defined within a stage will only apply the given environment variables to steps within the stage

**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.

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

##### **Available Parameters**

* **string**

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

* **booleanParam**

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

##### **Example**

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

parameters {

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

}

stages {

stage('Example') {

steps {

echo "Hello **${**params.PERSON**}**"

}

}

}

}

**Triggers**

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

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

* **cron**

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

* **pollSCM**

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

#### **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**

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

#### **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**

Jenkinsfile (Declarative Pipeline)

pipeline {

agent any

tools {

maven 'apache-maven-3.0.1'

}

stages {

stage('Example') {

steps {

sh 'mvn --version'

}

}

}

}

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

**When**

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

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.

* **environment**

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

* **expression**

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

* **not**

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

* **allOf**

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

* **anyOf**

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

**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**

Jenkinsfile (Declarative Pipeline)

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"

}

}

}

}

}

}

**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 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:

Jenkinsfile (Scripted Pipeline)

node {

stage('Example') {

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

echo 'I only execute on the master branch'

} **else** {

echo 'I execute elsewhere'

}

}

}

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:

Jenkinsfile (Scripted Pipeline)

node {

stage('Example') {

**try** {

sh 'exit 1'

}

**catch** (exc) {

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

**throw**

}

}

}

**Steps**

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.

## **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 implementers of Jenkins Pipeline found Groovy to be a solid foundation upon which to build what is now referred to as the "Scripted Pipeline" DSL.

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.

* Declarative Pipeline was created to offer a simpler and more opinionated syntax for authoring Jenkins Pipeline.
* The two are both fundamentally the same Pipeline sub-system underneath. They are both durable implementations of "Pipeline as code." They are both able to use steps built into Pipeline or provided by plugins. Both are able utilize Shared Libraries

Where they differ however is in syntax and flexibility:

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

**Difference b/w Declarative and Scripted pipelines (Agent vs Node)**

The simple answer is, **Agent** is for **declarative** pipelines and **node** is for **scripted** pipelines.

In declarative pipelines, the **agent** directive is used for specifying which agent/slave the job/task is to be executed on. This directive only allows you to specify where the task is to be executed, which agent, slave, label or docker image.

On the other hand, in scripted pipelines the **node** step can be used for executing a script/step on a specific agent, label, slave. The **node** step optionally takes the agent or label name and then a closure with code that is to be executed on that node.

**declarative and scripted (edit based on the comment):**

* **declarative pipelines** are a new extension of the pipeline DSL (it is basically a pipeline script with only one step, a pipeline step with arguments (called directives), these directives should follow a specific syntax. The point of this new format is that it is stricter and therefore should be easier for those new to pipelines, allow for graphical editing and much more.
* **scripted pipelines** are the fallback for advanced requirements.

**Another Explanation:**

* With declarative can have more of the job configuration in the Jenkinsfile like parameters and SCM polling. It means the Jenkins server can pick up the Jenkinsfiles for projects automatically with a MultiBranch pipeline container or GH Organisation. So, you don't have to create the jobs configs, just add the Jenkinsfile in the repo with the code.
* I really like the post section handling in Declarative for handling errors and failures. You can have post handing at the job or stage level. It means you don't need the try-catch-finally handling that you have. Seems cleaner to me.
* Declaration and use of tools is cleaner. Also, setup of environment variables for the build.

I don't see any issues with your scripted pipeline. I would use the "error ('Some failure occurred')" step instead of throwing / re-throwing exceptions for errors, It allows to generate the error message at the point the failure occurs. I do google searches against Github looking for interesting Jenkinsfiles or look in the CloudBees / Jenkins repos there.

**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".

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 ${library.yourLibName.version} somewhere in the configuration of the SCM, so that during checkout the plugin will expand this variable to select the desired version. For example, for Subversion, you can set the Repository URL to https://svnserver/project/${library.yourLibName.version} and then use versions such as trunk or branches/dev or tags/1.0

### Directory structure

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

(root)

+- src # Groovy source files

| +- org

| +- foo

| +- Bar.groovy # for org.foo.Bar class

+- vars

| +- foo.groovy # for global 'foo' variable

| +- foo.txt # help for 'foo' variable

+- resources # resource files (external libraries only)

| +- org

| +- foo

| +- bar.json # static helper data for org.foo.Bar

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

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

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

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

Other directories under the root are reserved for future enhancements.

### Global Shared Libraries

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

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

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

### Folder-level Shared Libraries

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

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

### Automatic Shared Libraries

Other plugins may add ways of defining libraries on the fly. For example, the 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:

@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**

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()

}

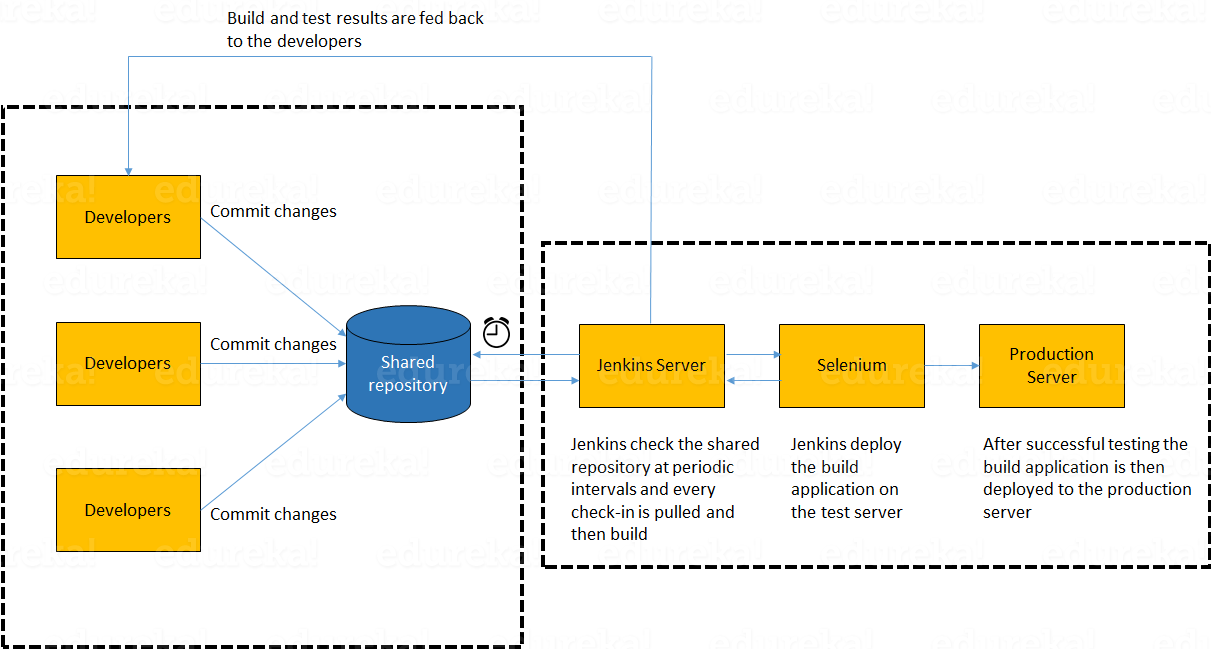
echo useSomeLib(**new** Helper('some text'))

Global Variables however, are resolved at runtime

**What is meant by Continuous Integration?**

It is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

I suggest that you explain how you have implemented it in your previous job. You can refer the below given example:



In the diagram shown above:

1. Developers check out code into their private workspaces.
2. When they are done with it they commit the changes to the shared repository (Version Control Repository).
3. The CI server monitors the repository and checks out changes when they occur.
4. The CI server then pulls these changes and builds the system and also runs unit and integration tests.
5. The CI server will now inform the team of the successful build.
6. If the build or tests fails, the CI server will alert the team.
7. The team will try to fix the issue at the earliest opportunity.
8. This process keeps on repeating.

**What is Jenkins?**

Jenkins is an open source automation tool written in Java with plugins built for Continuous Integration purpose. Jenkins is used to build and test your software projects continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. It also allows you to continuously deliver your software by integrating with many testing and deployment technologies.

you can refer the below mentioned use case:

* First, a developer commits the code to the source code repository. Meanwhile, the Jenkins server checks the repository at regular intervals for changes.
* Soon after a commit occurs, the Jenkins server detects the changes that have occurred in the source code repository. Jenkins will pull those changes and will start preparing a new build.
* If the build fails, then the concerned team will be notified.
* If built is successful, then Jenkins deploys the built in the test server.
* After testing, Jenkins generates a feedback and then notifies the developers about the build and test results.
* It will continue to check the source code repository for changes made in the source code and the whole process keeps on repeating.

**What are the benefits of using Jenkins?**

I will suggest you to include the following benefits of Jenkins, if you can recall any other benefit apart from the below mentioned points you can include that as well.

* At integration stage, build failures are cached.
* For each change in the source code an automatic build report notification is generated.
* To notify developers about build report success or failure, it is integrated with LDAP mail server.
* Achieves continuous integration agile development and test-driven development.
* With simple steps, maven release project is automated.
* Easy tracking of bugs at early stage in development environment than production.

**Why do you need a Continuous Integration of Dev & Testing?**

For this answer, you should focus on the need of Continuous Integration. My suggestion would be to mention the below explanation in your answer:

Continuous Integration of Dev and Testing improves the quality of software, and reduces the time taken to deliver it, by replacing the traditional practice of testing after completing all development.

It allows Dev team to easily detect and locate problems early because developers need to integrate code into a shared repository several times a day (more frequently). Each check-in is then automatically tested.

**What are the success factors for Continuous Integration?**

Here you have to mention the requirements for Continuous Integration. You could include the following points in your answer:

* Maintain a code repository
* Automate the build
* Make the build self-testing
* Everyone commits to the baseline every day
* Every commit (to baseline) should be built
* Keep the build fast
* Test in a clone of the production environment
* Make it easy to get the latest deliverables
* Everyone can see the results of the latest build
* Automate deployment

**Explain how you can move or copy Jenkins from one server to another?**

I will approach this task by copying the jobs directory from the old server to the new one.

There are multiple ways to do that:

* Move a job from one installation of Jenkins to another by simply copying the corresponding job directory.
* Make a copy of an existing job by making a clone of a job directory by a different name.
* Rename an existing job by renaming a directory. Note that if you change a job name you will need to change any other job that tries to call the renamed job.

Can you tell me how you deployed? Do u have any deploy plugin?

In recent application team, I have written some shell scripts for deploying into weblogic by providing the credentials. The way it works is from jenkins I have selected the production environment like QA, UAT so based on those parameters it will kick off the scripts. The shell script will have the properties like “envdata” file to get the environment details, like of you select QA then it will get the QA environment details, then it will take the artifacts (like which version to be specified) from nexus repository, then it will copy the artifact to the specified environment and after that it will stop the server and start server and also restarts the server after the deployment is done. Based on that we will get notifications like if the deployment is success or unsuccessful and troubleshoot it if there are any issues.

So this is by using shell scripts, another way to do it is by using cookbooks. I have written cookbooks from the scratch. It also takes the similar architecture but we will install, configure clusters and data sources on the fly for dynamic environments.

**Explain how can create a backup and copy files in Jenkins?**

To create a backup, all you need to do is to periodically back up your JENKINS\_HOME directory. This contains all your build jobs configurations, your slave node configurations, and your build history.

To create a back-up of your Jenkins setup, just copy this directory. You can also copy a job directory to clone or replicate a job or rename the directory.

**Explain how you can setup Jenkins job?**

My approach to this answer will be to first mention how to create Jenkins job. Go to Jenkins top page, select “New Job”, then choose “Build a free-style software project”.  
Then you can tell the elements of this freestyle job:

* Optional SCM, such as CVS or Subversion where your source code resides.
* Optional triggers to control when Jenkins will perform builds.
* Some sort of build script that performs the build (ant, maven, shell script, batch file, etc.) where the real work happens.
* Optional steps to collect information out of the build, such as archiving the artifacts and/or recording javadoc and test results.
* Optional steps to notify other people/systems with the build result, such as sending e-mails, IMs, updating issue tracker, etc.

**Mention some of the useful plugins in Jenkins.**

Below, I have mentioned some important Plugins:

* Maven 2 project
* Amazon EC2
* HTML publisher
* Copy artifact
* Join
* Green Balls

These Plugins, I feel are the most useful plugins. If you want to include any other Plugin that is not mentioned above, you can add them as well. But, make sure you first mention the above stated plugins and then add your own.

**How will you secure Jenkins?**

The way I secure Jenkins is mentioned below. If you have any other way of doing it, please mention it in the comments section below:

* Ensure global security is on.
* Ensure that Jenkins is integrated with my company’s user directory with appropriate plugin.
* Ensure that matrix/Project matrix is enabled to fine tune access.
* Automate the process of setting rights/privileges in Jenkins with custom version controlled script.
* Limit physical access to Jenkins data/folders.
* Periodically run security audits on same.

**Explain how you can deploy a custom build of a core plugin?**

Below are the steps to deploy a custom build of a core plugin:

* Stop Jenkins.
* Copy the custom HPI to **$Jenkins\_Home/plugins**.
* Delete the previously expanded plugin directory.
* Make an empty file called **<plugin>.hpi.pinned**.
* Start Jenkins.

**What you do when you see a broken build for your project in Jenkins?**

There can be multiple answers to this question I will approach this task in the following way:

I will open the console output for the broken build and try to see if any file changes were missed. If I am unable to find the issue that way, then I will clean and update my local workspace to replicate the problem on my local and try to solve it.

**What are the various ways in which build can be scheduled in Jenkins?**

You can schedule a build in Jenkins in the following ways:

* By source code management commits
* After completion of other builds
* Can be scheduled to run at specified time ( crons )
* Manual Build Requests