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](https://jenkins.io/doc/book/pipeline/syntax). [[1](https://jenkins.io/doc/book/pipeline/#_footnotedef_1)]

The definition of a Jenkins Pipeline is written into a text file (called a [Jenkinsfile](https://jenkins.io/doc/book/pipeline/jenkinsfile)) which in turn can be committed to a project’s source control repository. [[2](https://jenkins.io/doc/book/pipeline/#_footnotedef_2)] 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](https://jenkins.io/doc/book/pipeline/#_footnotedef_3)] 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](https://jenkins.io/doc/book/pipeline/#pipeline-concepts) and [Pipeline syntax overview](https://jenkins.io/doc/book/pipeline/#pipeline-syntax-overview) below.

Why Pipeline?

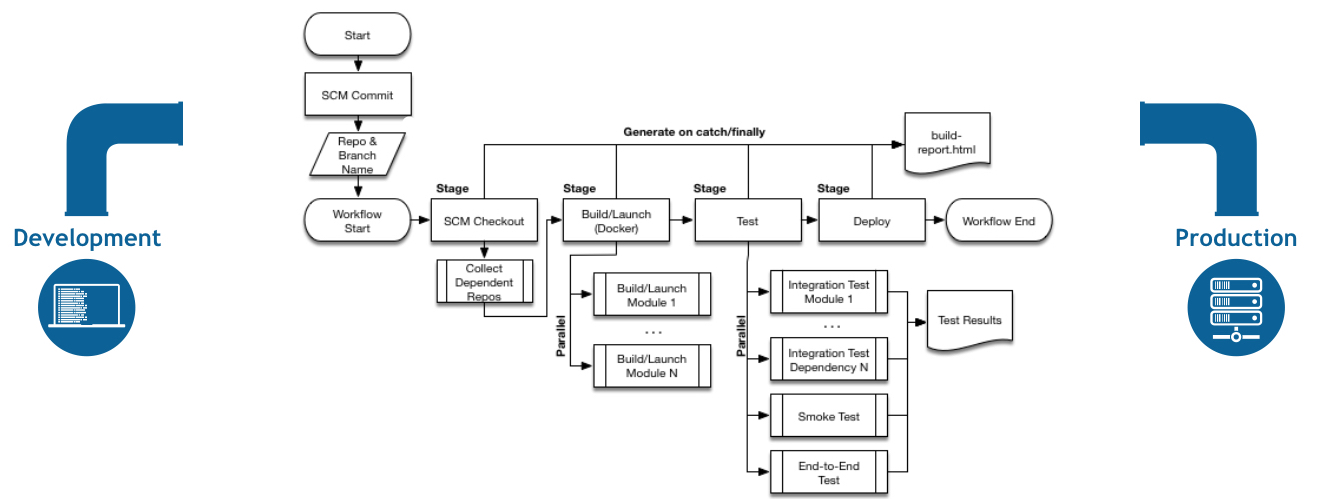
Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive CD pipelines. By modeling a series of related tasks, users 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](https://jenkins.io/doc/book/pipeline/#_footnotedef_1)] 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](https://jenkins.io/doc/book/pipeline/#_footnotedef_4)] 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](https://jenkins.io/doc/book/pipeline/shared-libraries) and by plugin developers. [[5](https://jenkins.io/doc/book/pipeline/#_footnotedef_5)]

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](https://jenkins.io/doc/book/pipeline/#pipeline-syntax-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](https://jenkins.io/doc/book/pipeline/#declarative-pipeline-fundamentals).

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](https://jenkins.io/doc/book/pipeline/#scripted-pipeline-fundamentals).

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](https://jenkins.io/doc/book/pipeline/#_footnotedef_6)]

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](https://jenkins.io/doc/book/pipeline/#_footnotedef_1)] that typically means the plugin has implemented a new *step*.

}

### Archive Build Output Artifacts

##### **Synopsis**

This is a simple demonstration of how to archive the build output artifacts in workspace for later use.

// This shows a simple example of how to archive the build output artifacts.

node {

stage "Create build output"

// Make the output directory.

sh "mkdir -p output"

// Write an useful file, which is needed to be archived.

writeFile file: "output/usefulfile.txt", text: "This file is useful, need to archive it."

// Write an useless file, which is not needed to be archived.

writeFile file: "output/uselessfile.md", text: "This file is useless, no need to archive it."

stage "Archive build output"

// Archive the build output artifacts.

archiveArtifacts artifacts: 'output/\*.txt', excludes: 'output/\*.md'

}

### Artifactory Generic Upload Download

##### **Synopsis**

This is a simple demonstration of how to download dependencies, upload artifacts and publish build info to Artifactory.   
Read the full documentation [here](https://www.jfrog.com/confluence/display/RTF/Working+With+Pipeline+Jobs+in+Jenkins).

node {

git url: 'https://github.com/jfrogdev/project-examples.git'

// Get Artifactory server instance, defined in the Artifactory Plugin administration page.

**def** server = Artifactory.server "SERVER\_ID"

// Read the upload spec and upload files to Artifactory.

**def** downloadSpec =

'''{

"files": [

{

"pattern": "libs-snapshot-local/\*.zip",

"target": "dependencies/",

"props": "p1=v1;p2=v2"

}

]

}'''

**def** buildInfo1 = server.download spec: downloadSpec

// Read the upload spec which was downloaded from github.

**def** uploadSpec =

'''{

"files": [

{

"pattern": "resources/Kermit.\*",

"target": "libs-snapshot-local",

"props": "p1=v1;p2=v2"

},

{

"pattern": "resources/Frogger.\*",

"target": "libs-snapshot-local"

}

]

}'''

// Upload to Artifactory.

**def** buildInfo2 = server.upload spec: uploadSpec

// Merge the upload and download build-info objects.

buildInfo1.append buildInfo2

// Publish the build to Artifactory

server.publishBuildInfo buildInfo1

}

### Artifactory Gradle Build

##### **Synopsis**

This is a simple demonstration of how to run a Gradle build, that resolves dependencies, upload artifacts and publish build info to Artifactory.   
Read the full documentation [here](https://www.jfrog.com/confluence/display/RTF/Working+With+Pipeline+Jobs+in+Jenkins).

node {

// Get Artifactory server instance, defined in the Artifactory Plugin administration page.

**def** server = Artifactory.server "SERVER\_ID"

// Create an Artifactory Gradle instance.

**def** rtGradle = Artifactory.newGradleBuild()

**def** buildInfo

stage('Clone sources') {

git url: 'https://github.com/jfrogdev/project-examples.git'

}

stage('Artifactory configuration') {

// Tool name from Jenkins configuration

rtGradle.tool = "Gradle-2.4"

// Set Artifactory repositories for dependencies resolution and artifacts deployment.

rtGradle.deployer repo:'ext-release-local', server: server

rtGradle.resolver repo:'remote-repos', server: server

}

stage('Gradle build') {

buildInfo = rtGradle.run rootDir: "gradle-examples/4/gradle-example-ci-server/", buildFile: 'build.gradle', tasks: 'clean artifactoryPublish'

}

stage('Publish build info') {

server.publishBuildInfo buildInfo

}

}

### Artifactory Maven Build

##### **Synopsis**

This is a simple demonstration of how to run a Maven build, that resolves dependencies, upload artifacts and publish build info to Artifactory.   
Read the full documentation [here](https://www.jfrog.com/confluence/display/RTF/Working+With+Pipeline+Jobs+in+Jenkins).

node {

// Get Artifactory server instance, defined in the Artifactory Plugin administration page.

**def** server = Artifactory.server "SERVER\_ID"

// Create an Artifactory Maven instance.

**def** rtMaven = Artifactory.newMavenBuild()

**def** buildInfo

stage('Clone sources') {

git url: 'https://github.com/jfrogdev/project-examples.git'

}

stage('Artifactory configuration') {

// Tool name from Jenkins configuration

rtMaven.tool = "Maven-3.3.9"

// Set Artifactory repositories for dependencies resolution and artifacts deployment.

rtMaven.deployer releaseRepo:'libs-release-local', snapshotRepo:'libs-snapshot-local', server: server

rtMaven.resolver releaseRepo:'libs-release', snapshotRepo:'libs-snapshot', server: server

}

stage('Maven build') {

buildInfo = rtMaven.run pom: 'maven-example/pom.xml', goals: 'clean install'

}

stage('Publish build info') {

server.publishBuildInfo buildInfo

}

}

Pipeline syntax overview

The following Pipeline code skeletons illustrate the fundamental differences between [Declarative Pipeline syntax](https://jenkins.io/doc/book/pipeline/#declarative-pipeline-fundamentals) and [Scripted Pipeline syntax](https://jenkins.io/doc/book/pipeline/#scripted-pipeline-fundamentals).

Be aware that both [stages](https://jenkins.io/doc/book/pipeline/#stage) and [steps](https://jenkins.io/doc/book/pipeline/#step) (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.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

//

}

}

stage('Test') {

steps {

//

}

}

stage('Deploy') {

steps {

//

}

}

}

}

|  |  |
| --- | --- |
|  | Execute this Pipeline or any of its stages, on any available agent. |
|  | Defines the "Build" stage. |
|  | Perform some steps related to the "Build" stage. |
|  | Defines the "Test" stage. |
|  | Perform some steps related to the "Test" stage. |
|  | Defines the "Deploy" stage. |
|  | Perform some steps related to the "Deploy" stage. |

Scripted Pipeline fundamentals

In Scripted Pipeline syntax, one or more node blocks do/es 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](https://issues.jenkins-ci.org/browse/JENKINS-2111) for more information.

*Jenkinsfile (Scripted Pipeline)*

node {

stage('Build') {

//

}

stage('Test') {

//

}

stage('Deploy') {

//

}

}

|  |  |
| --- | --- |
|  | Execute this Pipeline or any of its stages, on any available agent. |
|  | 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. |
|  | Perform some steps related to the "Build" stage. |
|  | Defines the "Test" stage. |
|  | Perform some steps related to the "Test" stage. |
|  | Defines the "Deploy" stage. |
|  | 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:

*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'

}

}

}

}

[Toggle Scripted Pipeline](https://jenkins.io/doc/book/pipeline/) *(Advanced)*

|  |  |
| --- | --- |
|  | [pipeline](https://jenkins.io/doc/book/pipeline/syntax#declarative-pipeline) is Declarative Pipeline-specific syntax that defines a "block" containing all content and instructions for executing the entire Pipeline. |
|  | [agent](https://jenkins.io/doc/book/pipeline/syntax#agent) is Declarative Pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline. |
|  | stage is a syntax block that describes a [stage of this Pipeline](https://jenkins.io/doc/book/pipeline/#stage). Read more about stage blocks in Declarative Pipeline syntax on the [Pipeline syntax](https://jenkins.io/doc/book/pipeline/syntax#stage) page. As mentioned [above](https://jenkins.io/doc/book/pipeline/#scripted-pipeline-fundamentals), stage blocks are optional in Scripted Pipeline syntax. |
|  | [steps](https://jenkins.io/doc/book/pipeline/syntax#steps) is Declarative Pipeline-specific syntax that describes the steps to be run in this stage. |
|  | sh is a Pipeline [step](https://jenkins.io/doc/book/pipeline/syntax#steps) (provided by the [Pipeline: Nodes and Processes plugin](https://plugins.jenkins.io/workflow-durable-task-step)) that executes the given shell command. |
|  | junit is another a Pipeline [step](https://jenkins.io/doc/book/pipeline/syntax#steps) (provided by the [JUnit plugin](https://plugins.jenkins.io/junit)) for aggregating test reports. |
|  | 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. |

#### 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](https://jenkins.io/doc/book/pipeline/syntax/#agent-parameters) |
| **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](https://jenkins.io/doc/book/glossary/#node) pre-configured to accept Docker-based Pipelines, or on a node matching the optionally defined label parameter. docker also optionally accepts an argsparameter 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'

}

}

**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'

}

}

##### **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**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'maven:3-alpine' }

stages {

stage('Example Build') {

steps {

sh 'mvn -B clean verify'

}

}

}

}

|  |  |
| --- | --- |
|  | 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**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent none

stages {

stage('Example Build') {

agent { docker 'maven:3-alpine' }

steps {

echo 'Hello, Maven'

sh 'mvn --version'

}

}

stage('Example Test') {

agent { docker 'openjdk:8-jre' }

steps {

echo 'Hello, JDK'

sh 'java -version'

}

}

}

}

|  |  |
| --- | --- |
|  | Defining agent none at the top-level of the Pipeline ensures that [an Executor](https://jenkins.io/doc/book/glossary/#executor) will not be assigned unnecessarily. Using agent none also forces each stage section to contain its own agent section. |
|  | Execute the steps in this stage in a newly created container using this image. |
|  | 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](https://jenkins.io/doc/book/pipeline/syntax/#declarative-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](https://jenkins.io/doc/book/pipeline/syntax/#post-conditions) 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.

**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**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

post {

always {

echo 'I will always say Hello again!'

}

}

}

|  |  |
| --- | --- |
|  | Conventionally, the post section should be placed at the end of the Pipeline. |
|  | [Post-condition](https://jenkins.io/doc/book/pipeline/syntax/#post-conditions) blocks contain [steps](https://jenkins.io/doc/book/pipeline/syntax/#declarative-steps) the same as the [steps](https://jenkins.io/doc/book/pipeline/syntax/#steps) section. |

#### stages

Containing a sequence of one or more [stage](https://jenkins.io/doc/book/pipeline/syntax/#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](https://jenkins.io/doc/book/pipeline/syntax/#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**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

|  |  |
| --- | --- |
|  | 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](https://jenkins.io/doc/book/pipeline/syntax/#declarative-steps) to be executed in a given stage directive.

|  |  |
| --- | --- |
| **Required** | Yes |
| **Parameters** | None |
| **Allowed** | Inside each stage block. |

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

|  |  |
| --- | --- |
|  | 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**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

environment {

CC = 'clang'

}

stages {

stage('Example') {

environment {

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

}

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. |
|  | 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() }

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

options {

timeout(time: 1, unit: 'HOURS')

}

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

|  |  |
| --- | --- |
|  | Specifying a global execution timeout of one hour, after which Jenkins will abort the Pipeline run. |
|  | A comprehensive list of available options is pending the completion of [INFRA-1503](https://issues.jenkins-ci.org/browse/INFRA-1053). |

##### **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**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Example') {

options {

timeout(time: 1, unit: 'HOURS')

}

steps {

echo 'Hello World'

}

}

}

}

|  |  |
| --- | --- |
|  | 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](https://jenkins.io/doc/book/pipeline/syntax/#parameters-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**

*Jenkinsfile (Declarative Pipeline)*

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**}**"

}

}

}

}

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

#### triggers

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

|  |  |
| --- | --- |
|  | The pollSCM trigger is only available in Jenkins 2.22 or later. |

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

triggers {

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

}

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

#### Jenkins cron syntax

The Jenkins cron syntax follows the syntax of the [cron utility](https://en.wikipedia.org/wiki/Cron) (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)</td> | The month (1–12) | The day of the week (0–7) where 0 and 7 are Sunday.</td> |

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](https://jenkins.io/doc/book/pipeline/syntax/#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**. |

#### input

The input directive on a stage allows you to prompt for input, using the [input step](https://jenkins.io/doc/pipeline/steps/pipeline-input-step/#input-wait-for-interactive-input). 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](https://jenkins.io/doc/book/pipeline/syntax/#parameters) for more information.

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

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."

}

}

}

}

#### when

The when directive allows the Pipeline to determine whether the stage should be executed depending on the given condition. The whendirective 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](https://jenkins.io/doc/book/pipeline/syntax/#when-example)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".