Pipeline Syntax

Table of Contents

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

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

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

Declarative Pipeline

Declarative Pipeline is a relatively recent addition to Jenkins Pipeline [1] which presents a more simplified and opinionated syntax on top of the Pipeline sub-systems.

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

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

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

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

Sections

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

agent

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

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

Parameters

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

any

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

none

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

label

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

node

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

docker

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

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

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'
    }
}
```

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`.

Example

Jenkinsfile (Declarative Pipeline)

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

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

Stage-level `agent` section

Jenkinsfile (Declarative Pipeline)

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

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

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

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

post

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

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.

`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.

`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.

Example

Jenkinsfile (Declarative Pipeline)

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

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

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

stages

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

Required	Yes
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

Example

Jenkinsfile (Declarative Pipeline)

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

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

steps

The `steps` section defines a series of one or more steps to be executed in a given `stage` directive.

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

Example

Jenkinsfile (Declarative Pipeline)

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

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

Directives

environment

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

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

Required	No
Parameters	*None*

Allowed Inside the `pipeline` block, or within `stage` directives.

Example

Jenkinsfile (Declarative Pipeline)

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

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

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

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

options

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

Required No

Parameters *None*

Allowed Only once, inside the `pipeline` block.

Available Options

buildDiscarder

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

disableConcurrentBuilds

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

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.

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() }`

checkoutToSubdirectory

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

timeout

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

retry

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

timestamps

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

Example

Jenkinsfile (Declarative Pipeline)

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

1 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.

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') (1)
            }
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

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

parameters

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

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

Available Parameters

string

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

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}"
            }
        }
    }
}
```

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

triggers

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

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

cron

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

pollSCM

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

upstream

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

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'
            }
        }
    }
}
```

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' (1)
    }
    stages {
        stage('Example') {
            steps {
                sh 'mvn --version'
            }
        }
    }
}
```

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

input

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

Configuration options

message

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

id

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

ok

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

submitter

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

submitterParameter

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

parameters

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

Example

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 `when` directive must contain at least one condition. If the `when` directive contains more than one condition, all the child conditions must return true for the stage to execute. This is the same as if the child conditions were nested in an `allOf` condition (see the examples below).

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' } }`

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

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

Examples

Jenkinsfile (Declarative Pipeline)

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

Jenkinsfile (Declarative Pipeline)

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

Jenkinsfile (Declarative Pipeline)

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

Jenkinsfile (Declarative Pipeline)

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

Jenkinsfile (Declarative Pipeline)

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

Jenkinsfile (Declarative Pipeline)

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

Parallel

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

In addition, you can force your `parallel` stages to all be aborted when one of them fails, by adding `failFast true` to the `stage` containing the `parallel`.

Example

Jenkinsfile (Declarative Pipeline)

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

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 [2] 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

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

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

Differences from plain Groovy

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

Syntax Comparison

When Jenkins Pipeline was first created, Groovy was selected as the foundation. Jenkins has long shipped with an embedded Groovy engine to provide advanced scripting capabilities for admins and users alike. Additionally, the implementors of Jenkins Pipeline found Groovy to be a solid foundation upon which to build what is now referred to as the "Scripted Pipeline" DSL. [2].

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

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

Where they differ however is in syntax and flexibility. Declarative limits what is available to the user with a more strict and pre-defined structure, making it an ideal choice for simpler continuous delivery pipelines. Scripted provides very few limits, insofar that the only limits on structure and syntax tend to be defined by Groovy itself, rather than any Pipeline-specific systems, making it an ideal choice for power-users and those with more complex requirements. As the name implies, Declarative Pipeline encourages a declarative programming model. [3] Whereas Scripted Pipelines follow a more imperative programming model.. [4]

---

1. Version 2.5 of the "Pipeline plugin" introduces support for Declarative Pipeline syntax
2. Domain-specific language
3. Declarative Programming
4. Imperative Programming

⇐ Pipeline Development Tools
⇑ Pipeline
Index
Scaling Pipelines ⇒

Was this page helpful?

Please submit your feedback about this page through this quick form.

Alternatively, if you don't wish to complete the quick form, you can simply indicate if you found this page helpful?

○ Yes ○ No

Type the answer to 9 plus 10 before clicking "Submit" below.

Submit

See existing feedback here.