Pipeline Syntax

Table of Contents

- Declarative Pipeline
  - Limitations

    - Sections
        - agent
        - post
        - stages
        - steps
    - Directives
        - environment
        - options
        - parameters
        - triggers
        - Jenkins cron syntax
        - stage
        - tools
        - input
        - when
    - Sequential Stages
    - Parallel
    - Matrix
        - axes
        - stages
        - excludes (optional)
        - Matrix cell-level directives (optional)
    - Steps
        - script
- Scripted Pipeline
    - Flow Control
    - Steps
    - Differences from plain Groovy
- Syntax Comparison

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 a no-argument method invocation. So, for example, `input` is treated as `input()`.

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

Limitations

There is currently an open issue which limits the maximum size of the code within the `pipeline{}` block. This limitation does not apply to Scripted pipelines.

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.

Differences between top and stage level Agents

There are some nuances when adding an agent to the top level or a stage level, and this when the `options` directive is applied.

Top Level Agents

In `agents` declared at the outermost level of the Pipeline, the options are invoked **after** entering the agent. As an example, when using `timeout` it will be only applied to the execution **within** the agent.

```
node("myAgent") {
    timeout(unit: 'SECONDS', time: 5) {
        stage("One"){
            sleep 10
            echo 'hello'
        }
    }
}
```

Stage Agents

In `agents` declared within a stage, the options are invoked **before** entering the `agent` and **before** checking any `when` conditions. In this case, when using `timeout`, it is applied **before** the `agent` is allocated.

```
timeout(unit: 'SECONDS', time: 5) {
    stage("One"){
        node {
            sleep 10
            echo 'Hello'
        }
    }
}
```

This timeout will include the agent provisioning time. Because the timeout includes the agent provisioning time, the Pipeline may fail in cases where agent allocation is delayed.

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' }`

Label conditions can also be used. For example: `agent { label 'my-label1 && my-label2' }` or `agent { label 'my-label1 || my-label2' }`

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:3.8.1-adoptopenjdk-11' }` or

```
agent {
    docker {
        image 'maven:3.8.1-adoptopenjdk-11'
        label 'my-defined-label'
        args  '-v /tmp:/tmp'
```

```
    }
}
```

`docker` also optionally accepts a `registryUrl` and `registryCredentialsId` parameters which will help to specify the Docker Registry to use and its credentials. The parameter `registryCredentialsId` could be used alone for private repositories within the docker hub. For example:

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

dockerfile

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

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

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

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

kubernetes

Execute the Pipeline, or stage, inside a pod deployed on a Kubernetes cluster. In order to use this option, the `Jenkinsfile` must be loaded from either a **Multibranch Pipeline** or a **Pipeline from SCM**. The Pod template is defined inside the kubernetes { } block. For example, if you want a pod with a Kaniko container inside it, you would define it as follows:

```
agent {
    kubernetes {
        label podlabel
        yaml """
kind: Pod
metadata:
  name: jenkins-agent
spec:
  containers:
  - name: kaniko
    image: gcr.io/kaniko-project/executor:debug
    imagePullPolicy: Always
    command:
    - /busybox/cat
    tty: true
    volumeMounts:
      - name: aws-secret
        mountPath: /root/.aws/
      - name: docker-registry-config
        mountPath: /kaniko/.docker
  restartPolicy: Never
  volumes:
    - name: aws-secret
      secret:
        secretName: aws-secret
    - name: docker-registry-config
      configMap:
        name: docker-registry-config
"""
   }
```

You will need to create a secret `aws-secret` for Kaniko to be able to authenticate with ECR. This secret should contain the contents of `~/.aws/credentials`. The other volume is a ConfigMap which should contain the endpoint of your ECR registry. For example:

```
{
      "credHelpers": {
```

```
            "<your-aws-account-id>.dkr.ecr.eu-central-1.amazonaws.com": "ecr-login"
        }
    }
```

Refer to the following example for reference: https://github.com/jenkinsci/kubernetes-plugin/blob/master/examples/kaniko.groovy

Common Options

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

label

A string. The label or label condition 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 1. Docker Agent, Declarative Pipeline

```
pipeline {
    agent { docker 'maven:3.8.1-adoptopenjdk-11' } (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 (`3.8.1-adoptopenjdk-11`).

Example 2. Stage-level Agent Section

```
pipeline {
    agent none (1)
    stages {
        stage('Example Build') {
            agent { docker 'maven:3.8.1-adoptopenjdk-11' } (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 to contain its own `agent` section.

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

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

post

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

`unsuccessful`

Only run the steps in `post` if the current Pipeline's or stage's run has not a "success" status. This is typically denoted in the web UI depending on the status previously mentioned.

`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 3. Post Section, 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 4. Stages, 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 5. Single Step, 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 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.

Required No

Parameters *None*

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

Supported Credentials Type

Secret Text

the environment variable specified will be set to the Secret Text content

Secret File

: the environment variable specified will be set to the location of the File file that is temporarily created

Username and password

: the environment variable specified will be set to `username:password` and two additional environment variables will be automatically defined: `MYVARNAME_USR` and `MYVARNAME_PSW` respectively.

SSH with Private Key

: the environment variable specified will be set to the location of the SSH key file that is temporarily created and two additional environment variables may be automatically defined: `MYVARNAME_USR` and `MYVARNAME_PSW` (holding the passphrase).

Unsupported credentials type causes the pipeline to fail with the message:
`org.jenkinsci.plugins.credentialsbinding.impl.CredentialNotFoundException: No suitable binding handler could be found for type <unsupportedType>.`

Example 6. Secret Text Credentials, Declarative Pipeline

```
pipeline {
    agent any
    environment { (1)
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment { (2)
                AN_ACCESS_KEY = credentials('my-predefined-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.

Example 7. Username and Password Credentials

```
pipeline {
    agent any
    stages {
        stage('Example Username/Password') {
            environment {
                SERVICE_CREDS = credentials('my-predefined-username-password')
            }
            steps {
                sh 'echo "Service user is $SERVICE_CREDS_USR"'
                sh 'echo "Service password is $SERVICE_CREDS_PSW"'
                sh 'curl -u $SERVICE_CREDS https://myservice.example.com'
            }
        }
        stage('Example SSH Username with private key') {
            environment {
                SSH_CREDS = credentials('my-predefined-ssh-creds')
            }
            steps {
                sh 'echo "SSH private key is located at $SSH_CREDS"'
                sh 'echo "SSH user is $SSH_CREDS_USR"'
                sh 'echo "SSH passphrase is $SSH_CREDS_PSW"'
            }
        }
    }
}
```

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() }`

disableResume

Do not allow the pipeline to resume if the controller restarts. For example: `options { disableResume() }`

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(buildCount: 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') }`

Example 8. Global Timeout, 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.

timestamps

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

parallelsAlwaysFailFast

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

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') }`

Example 9. Stage Timeout, Declarative Pipeline

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

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

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() }`

parameters

The `parameters` directive provides a list of parameters that 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 Parameters, Declarative Pipeline 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: '') }`

password

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

Example 10. Parameters, 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')
    }
    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.

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 11. Triggers, Declarative Pipeline

```
// Declarative //
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 (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

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

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

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

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

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

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

Beware that for the day of month field, short cycles such as `*/3` or `H/3` will not work consistently near the end of most months, due to variable month lengths. For example, `*/3` 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{ cron('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{ cron('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{ cron('H H(9-16)/2 * * 1-5') }
```

once a day on the 1st and 15th of every month except December

```
triggers{ cron('H H 1,15 1-11 *') }
```

stage

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

Required At least one

Parameters One mandatory parameter, a string for the name of the stage.

Allowed Inside the `stages` section.

Example 12. Stage, Declarative Pipeline

```
// Declarative //
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 13. Tools, 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 `agent` block for that `stage` or evaluating the `when` condition of the `stage`. 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 14. Input Step, 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). 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 (ANT style path glob) given, for example: `when { branch 'master' }`. Note that this only works on a multibranch Pipeline.

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

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 pattern. Example: `when { changeset "**/*.js" }`

The optional parameter `comparator` may be added after an attribute to specify how any patterns are evaluated for a match: `EQUALS` for a simple string comparison, `GLOB` (the default) for an ANT style path glob case insensitive, this can be turned off with the `caseSensitive` parameter, or `REGEXP` for regular expression matching. For example: `when { changeset pattern: ".TEST\\.java", comparator: "REGEXP" }` or `when { changeset pattern: "*/*TEST.java", 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, Change in Gerrit, etc.). When no parameters are passed the stage runs on every change request, for example: `when { changeRequest() }`.

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

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

environment

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

equals

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

expression

Execute the stage when the specified Groovy expression evaluates to true, for example: `when { expression { return params.DEBUG_BUILD } }` Note that when returning strings from your expressions they must be converted to booleans or return `null` to evaluate to false. Simply returning "0" or "false" will still evaluate to "true".

tag

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

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

not

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

allOf

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

anyOf

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

triggeredBy

Execute the stage when the current build has been triggered by the param given. For example:

- `when { triggeredBy 'SCMTrigger' }`
- `when { triggeredBy 'TimerTrigger' }`
- `when { triggeredBy 'UpstreamCause' }`
- `when { triggeredBy cause: "UserIdCause", detail: "vlinde" }`

Evaluating `when` before entering `agent` in a `stage`

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

Evaluating `when` before the `input` directive

By default, the when condition for a stage will not be evaluated before the input, if one is defined. However, this can be changed by specifying the `beforeInput` option within the when block. If `beforeInput` is set to true, the when condition will be evaluated first, and the input will only be entered if the when condition evaluates to true.

`beforeInput true` takes precedence over `beforeAgent true`.

Evaluating `when` before the `options` directive

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

`beforeOptions true` takes precedence over `beforeInput true` and `beforeAgent true`.

Example 15. Single Condition, Declarative Pipeline

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

Example 16. Multiple Condition, 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'
            }
        }
    }
}
```

Example 17. Nested condition (same behavior as previous example)

```
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'
            }
        }
    }
}
```

Example 18. Multiple condition and nested condition

```
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'
            }
```

```
        }
    }
}
```

Example 19. Expression condition and nested condition

```
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'
            }
        }
    }
}
```

Example 20. `beforeAgent`

```
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'
            }
        }
    }
}
```

Example 21. `beforeInput`

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

Example 22. `beforeOptions`

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                beforeOptions true
                branch 'testing'
            }
```

```
            options {
                lock label: 'testing-deploy-envs', quantity: 1, variable: 'deployEnv'
            }
            steps {
                echo "Deploying to ${deployEnv}"
            }
        }
    }
}
```

Example 23. `triggeredBy`

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                triggeredBy "TimerTrigger"
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

Sequential Stages

Stages in Declarative Pipeline may have a `stages` section containing a list of nested stages to be run in sequential order. Note that a stage must have one and only one of `steps`, `stages`, `parallel`, or `matrix`. It is not possible to nest a `parallel` or `matrix` block within a `stage` directive if that `stage` directive is nested within a `parallel` or `matrix` block itself. However, a `stage` directive within a `parallel` or `matrix` block can use all other functionality of a `stage`, including `agent`, `tools`, `when`, etc.

Example 24. Sequential Stages, Declarative Pipeline

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

Parallel

Stages in Declarative Pipeline may have a `parallel` section containing a list of nested stages to be run in parallel. Note that a stage must have one and only one of `steps`, `stages`, `parallel`, or `matrix`. It is not possible to nest a `parallel` or `matrix` block within a `stage` directive if that `stage` directive

is nested within a `parallel` or `matrix` block itself. However, a `stage` directive within a `parallel` or `matrix` block can use all other functionality of a `stage`, including `agent`, `tools`, `when`, etc.

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

Example 25. Parallel Stages, 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"
                    }
                }
                stage('Branch C') {
                    agent {
                        label "for-branch-c"
                    }
                    stages {
                        stage('Nested 1') {
                            steps {
                                echo "In stage Nested 1 within Branch C"
                            }
                        }
                        stage('Nested 2') {
                            steps {
                                echo "In stage Nested 2 within Branch C"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Example 26. `parallelsAlwaysFailFast`

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

```
                stage('Branch C') {
                    agent {
                        label "for-branch-c"
                    }
                    stages {
                        stage('Nested 1') {
                            steps {
                                echo "In stage Nested 1 within Branch C"
                            }
                        }
                        stage('Nested 2') {
                            steps {
                                echo "In stage Nested 2 within Branch C"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Matrix

Stages in Declarative Pipeline may have a `matrix` section defining a multi-dimensional matrix of name-value combinations to be run in parallel. We'll refer these combinations as "cells" in a matrix. Each cell in a matrix can include one or more stages to be run sequentially using the configuration for that cell. Note that a stage must have one and only one of `steps`, `stages`, `parallel`, or `matrix`. It is not possible to nest a `parallel` or `matrix` block within a `stage` directive if that `stage` directive is nested within a `parallel` or `matrix` block itself. However, a `stage` directive within a `parallel` or `matrix` block can use all other functionality of a `stage`, including `agent`, `tools`, `when`, etc.

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

The `matrix` section must include an `axes` section and a `stages` section. The `axes` section defines the values for each `axis` in the matrix. The `stages` section defines a list of `stages` to run sequentially in each cell. A `matrix` may have an `excludes` section to remove invalid cells from the matrix. Many of the directives available on `stage`, including `agent`, `tools`, `when`, etc., can also be added to `matrix` to control the behavior of each cell.

axes

The `axes` section specifies one or more `axis` directives. Each `axis` consists of a `name` and a list of `values`. All the values from each axis are combined with the others to produce the cells.

Example 27. One-axis with 3 cells

```
matrix {
    axes {
        axis {
            name 'PLATFORM'
            values 'linux', 'mac', 'windows'
        }
    }
    // ...
}
```

Example 28. Two-axis with 12 cells (three by four)

```
matrix {
    axes {
        axis {
            name 'PLATFORM'
            values 'linux', 'mac', 'windows'
        }
        axis {
            name 'BROWSER'
            values 'chrome', 'edge', 'firefox', 'safari'
        }
    }
    // ...
}
```

Example 29. Three-axis matrix with 24 cells (three by four by two)

```
matrix {
    axes {
        axis {
            name 'PLATFORM'
            values 'linux', 'mac', 'windows'
        }
        axis {
            name 'BROWSER'
            values 'chrome', 'edge', 'firefox', 'safari'
        }
        axis {
            name 'ARCHITECTURE'
            values '32-bit', '64-bit'
        }
    }
    // ...
}
```

stages

The `stages` section specifies one or more `stage`s to be executed sequentially in each cell. This section is identical to any other `stages section.

Example 30. One-axis with 3 cells, each cell runs three stages - "build", "test", and "deploy"

```
matrix {
    axes {
        axis {
            name 'PLATFORM'
            values 'linux', 'mac', 'windows'
        }
    }
    stages {
        stage('build') {
            // ...
        }
        stage('test') {
            // ...
        }
        stage('deploy') {
            // ...
        }
    }
}
```

Example 31. Two-axis with 12 cells (three by four)

```
matrix {
    axes {
        axis {
            name 'PLATFORM'
            values 'linux', 'mac', 'windows'
        }
        axis {
            name 'BROWSER'
            values 'chrome', 'edge', 'firefox', 'safari'
        }
    }
    stages {
        stage('build-and-test') {
            // ...
        }
    }
}
```

excludes (optional)

The optional `excludes` section lets authors specify one or more `exclude` filter expressions that select cells to be excluded from the expanded set of matrix cells (aka, sparsening). Filters are constructed using a basic directive structure of one or more of exclude `axis` directives each with a `name` and `values` list.

The `axis` directives inside an `exclude` generate a set of combinations (similar to generating the matrix cells). The matrix cells that match all the values from an `exclude` combination are removed from the matrix. If more than one `exclude` directive is supplied, each is evaluated separately to remove cells.

When dealing with a long list of values to exclude, exclude `axis` directives can use `notValues` instead of `values`. These will exclude cells that **do not** match one of the values passed to `notValues`.

Example 32. Three-axis matrix with 24 cells, exclude '32-bit, mac' (4 cells excluded)

```
matrix {
    axes {
        axis {
            name 'PLATFORM'
            values 'linux', 'mac', 'windows'
        }
        axis {
            name 'BROWSER'
            values 'chrome', 'edge', 'firefox', 'safari'
        }
        axis {
            name 'ARCHITECTURE'
            values '32-bit', '64-bit'
        }
    }
    excludes {
        exclude {
            axis {
                name 'PLATFORM'
                values 'mac'
            }
            axis {
                name 'ARCHITECTURE'
                values '32-bit'
            }
        }
    }
    // ...
}
```

Exclude the `linux, safari` combination and exclude any platform that is **not** `windows` with the `edge` browser.

Example 33. Three-axis matrix with 24 cells, exclude '32-bit, mac' and invalid browser combinations (9 cells excluded)

```
matrix {
    axes {
        axis {
            name 'PLATFORM'
            values 'linux', 'mac', 'windows'
        }
        axis {
            name 'BROWSER'
            values 'chrome', 'edge', 'firefox', 'safari'
        }
        axis {
            name 'ARCHITECTURE'
            values '32-bit', '64-bit'
        }
    }
    excludes {
        exclude {
            // 4 cells
            axis {
                name 'PLATFORM'
                values 'mac'
            }
            axis {
                name 'ARCHITECTURE'
                values '32-bit'
            }
        }
        exclude {
            // 2 cells
            axis {
                name 'PLATFORM'
                values 'linux'
            }
            axis {
                name 'BROWSER'
                values 'safari'
            }
        }
        exclude {
            // 3 more cells and '32-bit, mac' (already excluded)
            axis {
                name 'PLATFORM'
                notValues 'windows'
            }
            axis {
                name 'BROWSER'
                values 'edge'
            }
        }
    }
    // ...
}
```

Matrix cell-level directives (optional)

Matrix lets users efficiently configure the overall environment for each cell, by adding stage-level directives under `matrix` itself. These directives behave the same as they would on a stage but they can also accept values provided by the matrix for each cell.

The `axis` and `exclude` directives define the static set of cells that make up the matrix. That set of combinations is generated before the start of the pipeline run. The "per-cell" directives, on the other hand, are evaluated at runtime.

These directives include:

- agent
- environment
- input
- options
- post
- tools
- when

Example 34. Complete Matrix Example, Declarative Pipeline

```
pipeline {
    parameters {
        choice(name: 'PLATFORM_FILTER', choices: ['all', 'linux', 'windows', 'mac'], description: 'Run on specific platform')
    }
    agent none
    stages {
        stage('BuildAndTest') {
```

```
            matrix {
                agent {
                    label "${PLATFORM}-agent"
                }
                when { anyOf {
                    expression { params.PLATFORM_FILTER == 'all' }
                    expression { params.PLATFORM_FILTER == env.PLATFORM }
                } }
                axes {
                    axis {
                        name 'PLATFORM'
                        values 'linux', 'windows', 'mac'
                    }
                    axis {
                        name 'BROWSER'
                        values 'firefox', 'chrome', 'safari', 'edge'
                    }
                }
                excludes {
                    exclude {
                        axis {
                            name 'PLATFORM'
                            values 'linux'
                        }
                        axis {
                            name 'BROWSER'
                            values 'safari'
                        }
                    }
                    exclude {
                        axis {
                            name 'PLATFORM'
                            notValues 'windows'
                        }
                        axis {
                            name 'BROWSER'
                            values 'edge'
                        }
                    }
                }
                stages {
                    stage('Build') {
                        steps {
                            echo "Do Build for ${PLATFORM} - ${BROWSER}"
                        }
                    }
                    stage('Test') {
                        steps {
                            echo "Do Test for ${PLATFORM} - ${BROWSER}"
                        }
                    }
                }
            }
        }
    }
}
```

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 35. Script Block in 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:

Example 36. Conditional Statement `if`, 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:

Example 37. Try-Catch Block, 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 controller, Scripted Pipeline must serialize data back to the controller. 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
Pipeline Best Practices ⇒

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 6 plus 3 before clicking "Submit" below.

Submit

See existing feedback here.

Improve this page | Report a problem

Resources

- Downloads
- Blog
- Documentation
- Plugins
- Security
- Contributing

Project

- Structure and governance
- Issue tracker
- Roadmap
- GitHub
- Jenkins on Jenkins

Community

- Events
- Mailing lists
- Chats
- Special Interest Groups
- Twitter
- Reddit

Other

- Code of Conduct
- Press information
- Merchandise
- Artwork
- Awards