Pipeline Table

Jenkins Pipeline	`pipeline {` `  agent any` `  options {` `  skipStagesAfterUnstable()` `  }` `  stages {` `  stage('Build') {` `  steps {` `  sh 'make'` `  }` `  }` `  stage('Test'){` `  steps {` `  sh 'make check'` `  junit 'reports/**/*.xml'` `  }` `  }` `  stage('Deploy') {` `  steps {` `  sh 'make publish'` `  }` `  }` `  }` `}`
The variables provided by default in Pipeline are	**env** Exposes environment variables, for example: `env.PATH` or `env.BUILD_ID`. Consult the built-in global variable reference at `${YOUR_JENKINS_URL}/pipeline-syntax/globals#env` for a complete, and up to date, list of environment variables available in Pipeline. **params** Exposes all parameters defined for the Pipeline as a read-only Map, for example: `params.MY_PARAM_NAME`. **currentBuild** May be used to discover information about the currently executing Pipeline, with properties such as `currentBuild.result`, `currentBuild.displayName`,

	etc. Consult the built-in global variable reference at `${YOUR_JENKINS_URL}/pipeline-syntax/globals` for a complete, and up to date, list of properties available on `currentBuild` `pipeline {` `  agent any` `  stages {` `  stage('Deploy') {` `  when {` `  expression {` `  currentBuild.result == null || currentBuild.result == 'SUCCESS'` `  }` `  }` `  steps {` `  sh 'make publish'` `  }` `  }` `  }` `}`
Using environment variables	Jenkins Pipeline exposes environment variables via the global variable `env`, which is available from anywhere within a `Jenkinsfile`. The full list of environment variables accessible from within Jenkins Pipeline is documented at ${YOUR_JENKINS_URL}/pipeline-syntax/globals#env and includes: **BUILD_ID BUILD_NUMBER BUILD_TAG BUILD_URL EXECUTOR_NUMBER** **JAVA_HOME JENKINS_URL JOB_NAME NODE_NAME**
Setting environment variables	Declarative Pipeline supports an environment directive. `pipeline {` `  agent any` `  environment {` `  CC = 'clang'` `  }` `  stages {` `  stage('Example') {` `  environment {`

	`` DEBUG_FLAGS = '-g' } steps { sh 'printenv' } } } } ```
Setting environment variables dynamically	Each script can either `returnStatus` or `returnStdout`. ``` —----- environment { // Using returnStdout CC = """${sh(returnStdout: true, script: 'echo "clang"')}""" // Using returnStatus EXIT_STATUS = """${sh(returnStatus: true, script: 'exit 1')}""" } —--------- ```
For secret text, usernames and passwords, and secret files	``` environment { AWS_ACCESS_KEY_ID = credentials('jenkins-aws-secret-key-id') AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key') } ```
Handling parameters	``` parameters { string(name: 'Greeting', defaultValue: 'Hello', description: 'How should I greet the world?') } ```
Handling failure	Declarative Pipeline supports robust failure handling by default via its post section which allows declaring a number of different "post conditions" such as: `always`, `unstable`, `success`, `failure`, and `changed`. *Jenkinsfile (Declarative Pipeline)* ``` pipeline { agent any stages { stage('Test') { steps { sh 'make check' ```

	`` } } } post { always { junit '**/target/*.xml' } failure { mail to: team@example.com, subject: 'The Pipeline failed :(' } } } ```
Using multiple agents	``` stage('Test on Linux') { agent { label 'linux' } steps { } ```
Using Docker with Pipeline	``` pipeline { agent { docker { image 'node:14-alpine' } } ```
Workspace synchronization	When `reuseNode` set to `true`: no new workspace will be created, and current workspace from current agent will be mounted into container, and container will be started at the same node, so whole data will be synchronized.
Caching data for containers	``` pipeline { agent { docker { image 'maven:3.8.1-adoptopenjdk-11' args '-v $HOME/.m2:/root/.m2' } } ```
Using multiple containers	``` pipeline { agent none stages { stage('Back-end') { ```

	`` agent { docker { image 'maven:3.8.1-adoptopenjdk-11' } } steps { sh 'mvn --version' } } stage('Front-end') { agent { docker { image 'node:14-alpine' } } steps { sh 'node --version' } } } } ```

Pipeline elements

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. ``` agent any agent none agent { label 'my-label1 && my-label2' } or agent { label 'my-label1 || my-label2' } agent { docker { image 'myregistry.com/node' label 'my-defined-label' registryUrl 'https://myregistry.com/' registryCredentialsId 'myPredefinedCredentialsInJenkins' } } ```
Post	``` post { always { ```

The `post` section defines one or more additional steps that are run upon the completion of a Pipeline's or stage's run	`echo 'I will always say Hello again!'` `}` `}`
environment	`pipeline {` `agent any` `environment {` `CC = 'clang'` `}` `stages {` `stage('Example') {` `environment {` `AN_ACCESS_KEY = credentials('my-predefined-secret-text')` `}` `steps {` `sh 'printenv'` `}` `}` `}` `}`
options	The `options` directive allows configuring Pipeline-specific options from within the Pipeline itself. **Retry:** On failure, retry the entire Pipeline the specified number of times. For example: `options { retry(3) }` **Timeout** Set a timeout period for the Pipeline run, after which Jenkins should abort the Pipeline. For example: `options { timeout(time: 1, unit: 'HOURS') }` `pipeline {` `agent any` `options {` `timeout(time: 1, unit: 'HOURS')` `}` `stages {` `stage('Example') {` `steps {` `echo 'Hello World'` `}` `}` `}`

	`}` 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`. `pipeline {` `  agent any` `  stages {` `  stage('Example') {` `  options {` `  timeout(time: 1, unit: 'HOURS')` `  }` `  steps {` `  echo 'Hello World'` `  }` `  }` `  }` `}`
Available Parameters	`string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')` **Text** **booleanParam** **Choice** **Password** `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}" } } } } ```
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` **Cron** `triggers { cron('H */4 * * 1-5') }` **pollSCM** `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. `triggers { upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS) }` ``` pipeline { agent any triggers { cron('H */4 * * 1-5') } stages { stage('Example') { steps { ```

<table>
<tr><td></td><td>

```
                echo 'Hello World'
            }
        }
    }

}
```

</td></tr>
<tr><td>tools
A section defining tools to auto-install and put on the <code>PATH</code>. This is ignored if <code>agent none</code> is specified.</td><td>Supported Tools
maven
jdk
gradle
Example 13. Tools, Declarative Pipeline

```
pipeline {
    agent any
    tools {
        maven 'apache-maven-3.0.1'
    }
    stages {
        stage('Example') {
            steps {
                sh 'mvn --version'
            }
        }
    }
}
```

</td></tr>
<tr><td>input</td><td>The <code>input</code> directive on a <code>stage</code> allows you to prompt for input, using the <code>input</code> step.</td></tr>
<tr><td>When</td><td>The <code>when</code> directive allows the Pipeline to determine whether the stage should be executed depending on the given condition.
branch
for example: <code>when { branch 'master' }</code>

Environment
<code>when { environment name: 'DEPLOY_TO', value: 'production' }</code>

Not
<code>when { not { branch 'master' } }</code></td></tr>
</table>

	allOf `when { allOf { branch 'master'; environment name: 'DEPLOY_TO', value: 'production' } }` **anyOf** `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' }`
	Parallel Stages in Declarative Pipeline may have a `parallel` section containing a list of nested stages to be run in parallel. `stage('Parallel In Sequential') { parallel { stage('In Parallel 1') { steps { echo "In Parallel 1" } } stage('In Parallel 2') { steps { echo "In Parallel 2" } } } }`
trigger one pipeline after another jenkins	`pipeline { agent any stages { stage('E2E Tests') { steps { echo "Running E2E tests..."`

```
                echo "MY_PARAM=${env.MY_PARAM}"
            }
        }
    }

}
```

Now we will build our main pipeline which will trigger the e2e tests.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo "Build step"
            }
        }

        stage('Test') {
            steps {
                echo "Test step"
            }
        }

        stage('Upload artifacts') {
            steps {
                echo "Upload artifacts step"
            }
        }
    }
    post {
        success {
            echo 'Run E2E Test pipeline!'
            build job: 'E2E_tests_pipeline', parameters:
[string(name: 'MY_PARAM', value: 'value from Build
pipeline')]
```

	<pre>} } }</pre>

Theory

ways to trigger a Jenkins Job/Pipeline	• Trigger an API (POST) request to the target job URL with the required data. • Trigger it manually from the Jenkins web application. • Trigger it using Jenkins CLI from the master/slave nodes. • Time-based Scheduled Triggers like a cron job. • Event-based Triggers like SCM Actions (Git Commit, Pull Requests), WebHooks, etc. • Upstream/Downstream triggers by other Jenkins jobs.
credential types supported by Jenkins	• Secret text - A token such as an API token, JSON token, etc. • Username and password - Basic Authentication can be stored as a credential as well. • Secret file - A secret file used to authenticate some secure data services & security handshakes. • SSH Username with a private key - An SSH public/private key pair for Machine to Machine authentication. • Certificate - a PKCS#12 certificate file and an optional password. • Docker Host Certificate Authentication credentials.

Scopes of Jenkins Credentials	Global - the credential will be usable across all the jobs configured in the Jenkins instance (i.e. for all jobs). This is more suited for user Jobs (i.e. for the freestyle, pipeline, or other jobs) to authenticate itself with target services/infrastructures to accomplish the purpose of the job) System - This is a special scope that will allow the Jenkins itself (i.e. the core Jenkins functionalities & some installed plugins) to authenticate itself to external services/infrastructures to perform some defined tasks. E.g. sending emails, etc.
Jenkins Shared Library	Shared libraries are a set of code that can be common for more than one pipeline job and can be maintained separately. Such libraries improve the maintenance, modularity & readability of the pipeline code. And it also speeds up the automation for new jobs
