Table of Contents

Certified Jenkins Engineer (CJE) – 2017

This exam is comprised of 4 sections:

1. Key CI/CD/Jenkins concepts
2. Jenkins usage
3. Building Continuous Delivery (CD) Pipelines
4. CD-as-code best practices

All **questions** are based on version 2.19.4 [new] of the Jenkins core.

Questions in sections 1–4 primarily cover questions about a "base" Jenkins installation, but knowledge of the "suggested" plugins will also be covered. Candidates are expected to know the functionality/uses of these plugins but will not be tested on detailed usage.

The "**suggested**" plugins are the default plugins installed by the "Setup Wizard" on a fresh new Jenkins installation. You can find the exhaustive list, bound to a fixed Jenkins version, by following this link: Jenkins 2.19.4 suggested plugin list.

> # Please refer to Jeanne’s experiences with the jenkins certification beta exam and download the Study-Notes.docx -
>
> http://www.selikoff.net/wp-content/uploads/2016/02/Study-Notes.docx
>
> This in addition to what was mentioned in her Study Notes. - Murali Bala

Jenkins Certification link

https://certificates.cloudbees.com/vly4nfhs

Continuous Integration, Continuous Delivery and Continuous Deployment

Continuous Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Continuous Integration practices

-
    - Single source repository
    - Automate the build
    - Make your build self testing
    - Everyone commits everyday
    - Every commit triggers a build
    - Fix broken builds immediately
    - Keep the commit build fast (and use pipeline for slower builds)
    - Test in a clone of the prod environment
    - Make it easy to get the latest build
    - Visibility
    - Automate deployment

Continuous Delivery is a software development discipline where you build software in such a way that the software **can be released to production at any time**.

You are doing continuous delivery when:[1]

- Your software is deployable throughout its lifecycle
- Your team prioritizes keeping the software deployable over working on new features
- Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them

- You can perform push-button deployments of any version of the software to any environment on demand

You achieve continuous delivery by continuously integrating the software done by the development team, building executables, and running automated tests on those executables to detect problems. Furthermore you push the executables into increasingly production-like environments to ensure the software will work in production. To do this you use a DeploymentPipeline.

The key test is that a business sponsor could request that the current development version of the software can be deployed into production at a moment's notice - and nobody would bat an eyelid, let alone panic.

To achieve continuous delivery you need:

- a close, collaborative working relationship between everyone involved in delivery (often referred to as a DevOpsCulture[2]).
- extensive automation of all possible parts of the delivery process, usually using a DeploymentPipeline

Continuous Delivery is sometimes confused with Continuous Deployment.

Continuous Deployment means that every change goes through the pipeline and **automatically gets put into production**, resulting in many production deployments every day.

Continuous Delivery *just means that you are able to do frequent deployments but may choose not to do it*, usually due to businesses preferring a slower rate of deployment. ***In order to do Continuous Deployment you must be doing Continuous Delivery***

Continuous Integration usually refers to integrating, building, and testing code within the development environment.

Continuous Delivery builds on this, dealing with the final stages required for production deployment.

The principal benefits of continuous delivery are:

- **Reduced Deployment Risk**: since you are deploying smaller changes, there's less to go wrong and it's easier to fix should a problem appear.
- **Believable Progress**: many folks track progress by tracking work done. If "**done**" means "developers declare it to be done" that's much less believable than if it's deployed into a production (or production-like) environment.

- **User Feedback**: the biggest risk to any software effort is that you end up building something that isn't useful. The earlier and more frequently you get working software in front of real users, the quicker you get feedback to find out how valuable it really is (particularly if you use ObservedRequirements).

One of the challenges of an automated build and test environment is you want your build to be fast, so that you can get fast feedback, but comprehensive tests take a long time to run. A deployment pipeline is a way to deal with this by breaking up your build into stages. Each stage provides increasing confidence, usually at the cost of extra time. Early stages can find most problems yielding faster feedback, while later stages provide slower and more through probing. Deployment pipelines are a central part of ContinuousDelivery.

Usually the first stage of a deployment pipeline will do any compilation and provide binaries for later stages. Later stages may include manual checks, such as any tests that can't be automated. Stages can be automatic, or require human authorization to proceed, they may be parallelized over many machines to speed up the build. Deploying into production is usually the final stage in a pipeline.

More broadly the deployment pipeline's job is to detect any changes that will lead to problems in production. These can include performance, security, or usability issues. A deployment pipeline should enable collaboration between the various groups involved in delivering software and provide everyone visibility about the flow of changes in the system, together with a thorough audit trail.

A good way to introduce continuous delivery is to model your current delivery process as a deployment pipeline, then examine this for bottlenecks, opportunities for automation, and collaboration points.

Continuous Delivery (CD) is a software strategy that enables organizations to deliver new features to users as fast and efficiently as possible. The core idea of CD is to create a repeatable, reliable and incrementally improving process for taking software from concept to customer. The goal of Continuous Delivery is to enable a constant flow of changes into production via an automated software production line. The Continuous Delivery pipeline is what makes it all happen.

The pipeline breaks down the software delivery process into stages. Each stage is aimed at verifying the quality of new features from a different angle to validate the new functionality and prevent errors from affecting your users. The pipeline should provide feedback to the team and visibility into the flow of changes to everyone involved in delivering the new feature/s.

There is no such thing as The Standard Pipeline, but a typical CD pipeline will include the following stages: build automation and continuous integration; test automation; and deployment automation.

Your Pipeline Needs Platform Provisioning and Configuration Management

The deployment pipeline is supported by platform provisioning and system configuration management, which allow teams to create, maintain and tear down complete environments automatically or at the push of a button.

Automated platform provisioning ensures that your candidate applications are deployed to, and tests carried out against, correctly configured and reproducible environments. It also facilitates horizontal scalability and allows the business to try out new products in a sandbox environment at any time.

Orchestrating it all: Release and Pipeline Orchestration

The multiple stages in a deployment pipeline involve different groups of people collaborating and supervising the release of the new version of your application. Release and pipeline orchestration provides a top-level view of the entire pipeline, allowing you to define and control the stages and gain insight into the overall software delivery process. By carrying out value stream mappings on your releases, you can highlight any remaining inefficiencies and hot spots, and pinpoint opportunities to improve your pipeline.

Don’t Add New Functionality Until You Get The Quality Right!

Poor system quality, low-user satisfaction and endless “quality band-aids” can be avoided by adopting the principle of not adding new functionality before getting the quality right. You should always first meet and maintain your quality levels and only then consider gradually adding functionality to the system.

Starting out on CD

1. Define a process
2. Ensure a blameless culture
3. Set metrics and measure your success
4. Adopt configuration as code
5. Orchestrating a process

You’ve defined your pipeline; now you need to orchestrate it. This is a long process, but here are a few steps you’ll want to take – as outlined in a whitepaper by Xebia Labs’ Andrew Phillips and CloudBees’ Kohsuke Kawaguchi.

- Ensure reproducible builds – Configure the build system to use a clean repository connected to the build job’s workspace and use a central shared repository for build dependencies.

- Share build artefacts through the pipeline – Ensure the candidate artefact is used by all subsequent builds in your pipeline.
- Choose the right granularity for each job – Distribute all steps in the pipeline across multiple jobs, allowing you to identify bottlenecks more easily.
- Visualize the pipeline – Create a clear, accessible view of the build pipeline to allow for smooth status communications and transparency of the process to the business managers and other stakeholders.

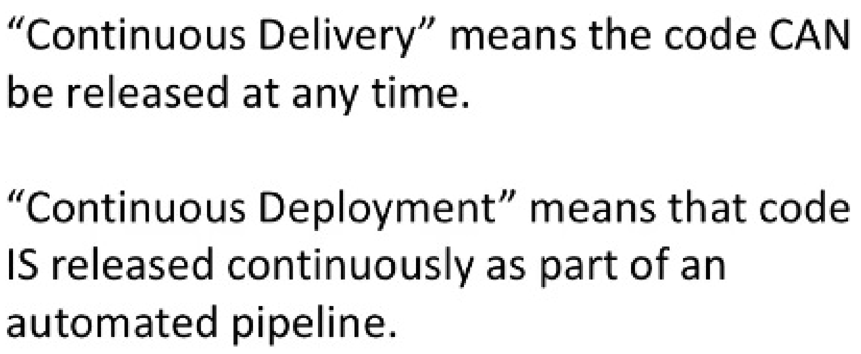
Continuous Testing is one of several continuous activities that should take place simultaneously in a DevOps pipeline, including

- Continuous Build;
- Continuous Integration (CI);
- Continuous Delivery (CD); and
- Continuous Deployment.

Continuous Build or build automation is the first stage in implementing a Continuous Delivery DevOps pipeline. If your developers are practicing test-driven development (TDD), they'll write unit tests for each piece of code they write, even before the code itself is written. An important part of the agile methodology, TDD helps developers think through the desired behavior of each unit of software they're building, including inputs, outputs, and error conditions. New features implemented by developers are then checked into a central code base prior to the software build, which compiles the source code into binary code.

Remember, Delivery means the ability to deploy to production. Deployment means actually doing so.

Continuous deployment depends on continuous delivery which depends on continuous integration.



"Continuous Delivery" means the code CAN be released at any time.

"Continuous Deployment" means that code IS released continuously as part of an automated pipeline.

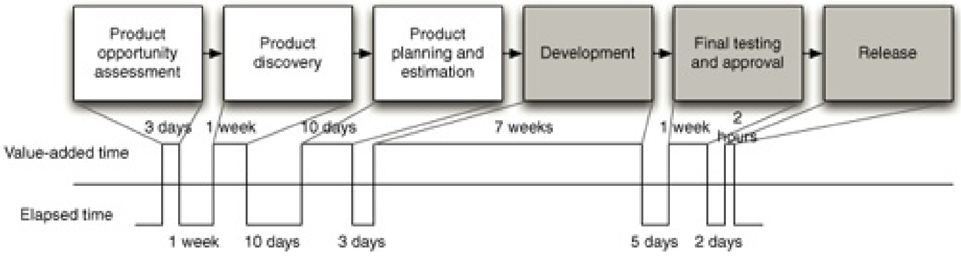
KPIs/metrics for CI/CD

- Cycle time
- Test coverage, cyclomatic complexity, duplication, etc
- Number of defects
- Velocity
- # Commits per day
- # Builds per day - success, failures and total
- Duration of build

Deployment Pipeline

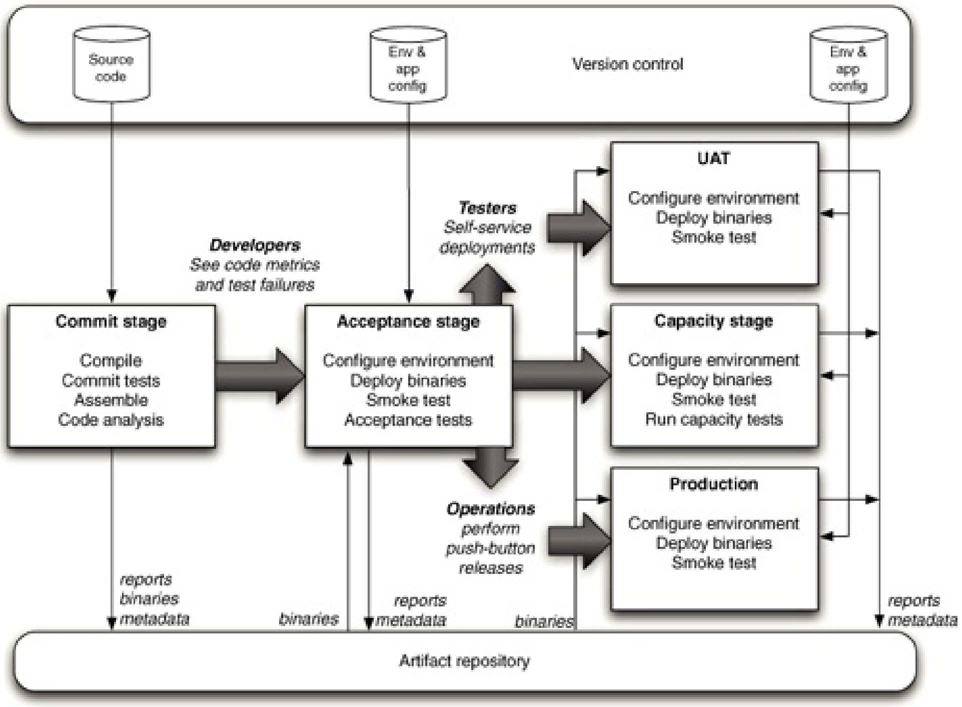
At an abstract level, a deployment pipeline is an automated manifestation of your process for getting software from version control into the hands of your users. Every change to your software goes through a complex process on its way to being released. That process involves building the software, followed by the progress of these builds through multiple stages of testing and deployment. This, in turn, requires collaboration between many individuals, and perhaps several teams. The deployment pipeline models this process, and its incarnation in a continuous integration and release management tool is what allows you to see and control the progress of each change as it moves from version control through various sets of tests and deployments to release to users.

Thus the process modeled by the deployment pipeline, the process of getting software from check-in to release, forms a part of the process of getting a feature from the mind of a customer or user into their hands. The entire process—from concept to cash—can be modeled as a value stream map. A high-level value stream map for the creation of a new product is shown below:



This value stream map tells a story. The whole process takes about three and a half months. About two and a half months of that is actual work being done—there are waits between the various stages in the process of getting the software from concept to cash. For example, there is a five-day wait between the development team completing work on the first release and the start of the testing process. This might be due to the time it takes to deploy the application to a production-like environment, for example. As an aside, it has been left deliberately unclear in this diagram whether or not this product is being developed in an iterative way. In an iterative process, you'd expect to see the development process itself consist of several iterations which include testing and showcasing. The whole process from discovery to release would also be repeated many times.

A Basic Deployment Pipeline



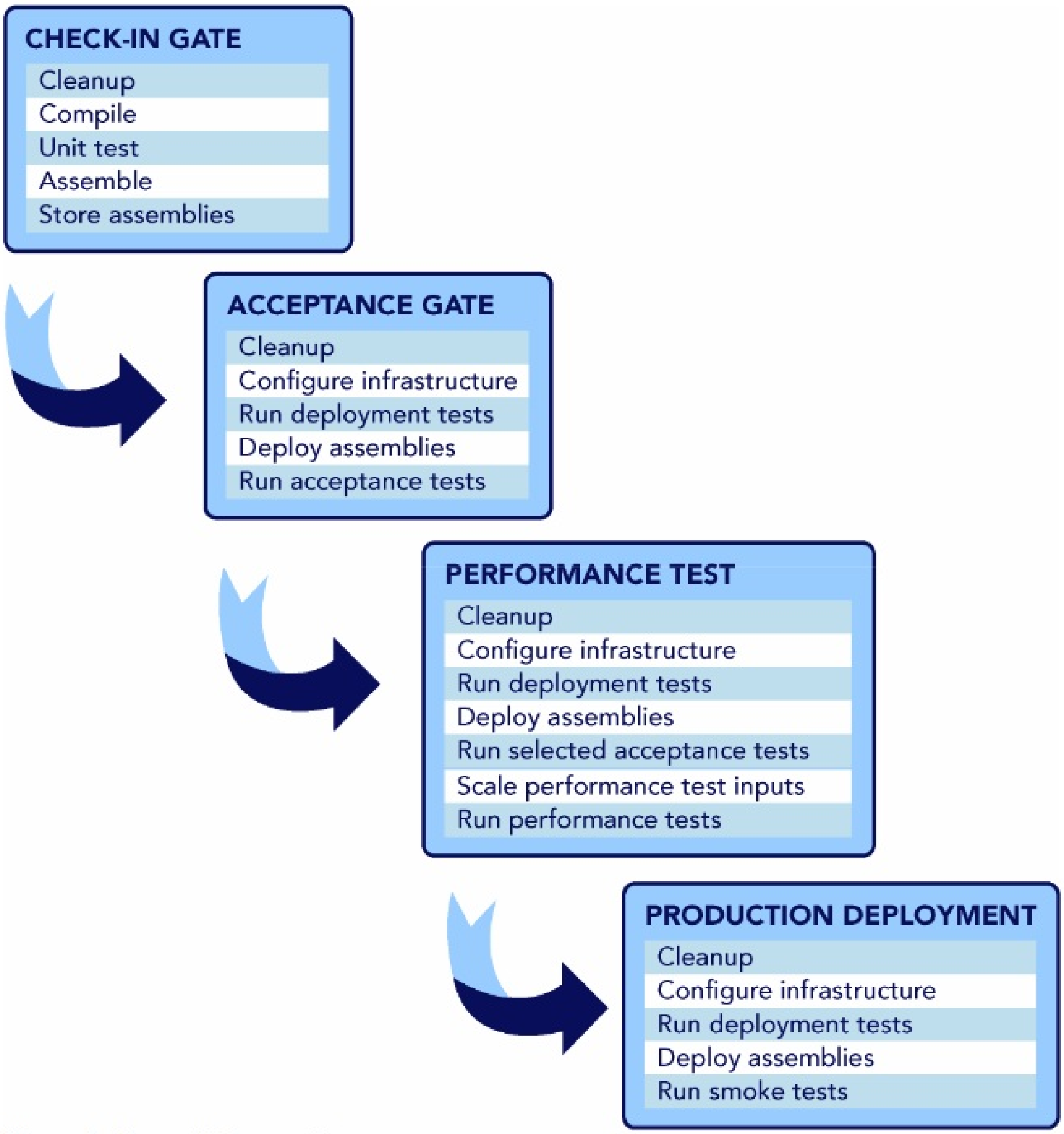
The process starts with the developers committing changes into their version control system. At this point, the continuous integration management system responds to the commit by triggering a new instance of our pipeline. The first (commit) stage of the pipeline compiles the code, runs unit tests, performs code analysis, and creates installers. If the unit tests all pass and the code is up to scratch, we assemble the executable code into binaries and store them in an artifact repository. Modern CI servers provide a facility to store artifacts like these and make them easily accessible both to the users and to the later stages in your pipeline. Alternatively, there are plenty of tools like Nexus and Artifactory which help you manage artifacts. There are other tasks that you might also run as part of the commit stage of your pipeline, such as preparing a test database to use for your acceptance tests. Modern CI servers will let you execute these jobs in parallel on a build grid.

The second stage is typically composed of longer-running automated acceptance tests. Again, your CI server should let you split these tests into suites which can be executed in parallel to increase their speed and give you feedback faster—typically within an hour or two. This stage will be triggered automatically by the successful completion of the first stage in your pipeline.

At this point, the pipeline branches to enable independent deployment of your build to various environments—in this case, UAT (user acceptance testing), capacity testing, and production. Often, you won't want these stages to be automatically triggered by the successful completion of your acceptance test stage. Instead, you'll want your testers or operations team to be able to self-service builds into their environments manually. To facilitate this, you'll need an automated script that performs this deployment. Your testers should be able to see the release candidates available to them as well as their status—which of the previous two stages each build has passed, what were the check-in comments, and any other comments on those builds. They should then be able to press a button to deploy the selected build by running the deployment script in the relevant environment.

Sequence of Gates (Deployment Pipeline)

Source: https://continuousdelivery.com/wp-content/uploads/2010/01/The-Deployment-Pipeline-by-Dave-Farley-2007.pdf



CHECK-IN GATE
- Cleanup
- Compile
- Unit test
- Assemble
- Store assemblies

ACCEPTANCE GATE
- Cleanup
- Configure infrastructure
- Run deployment tests
- Deploy assemblies
- Run acceptance tests

PERFORMANCE TEST
- Cleanup
- Configure infrastructure
- Run deployment tests
- Deploy assemblies
- Run selected acceptance tests
- Scale performance test inputs
- Run performance tests

PRODUCTION DEPLOYMENT
- Cleanup
- Configure infrastructure
- Run deployment tests
- Deploy assemblies
- Run smoke tests

Figure 2 – Example Process Steps

Jenkins Pipeline

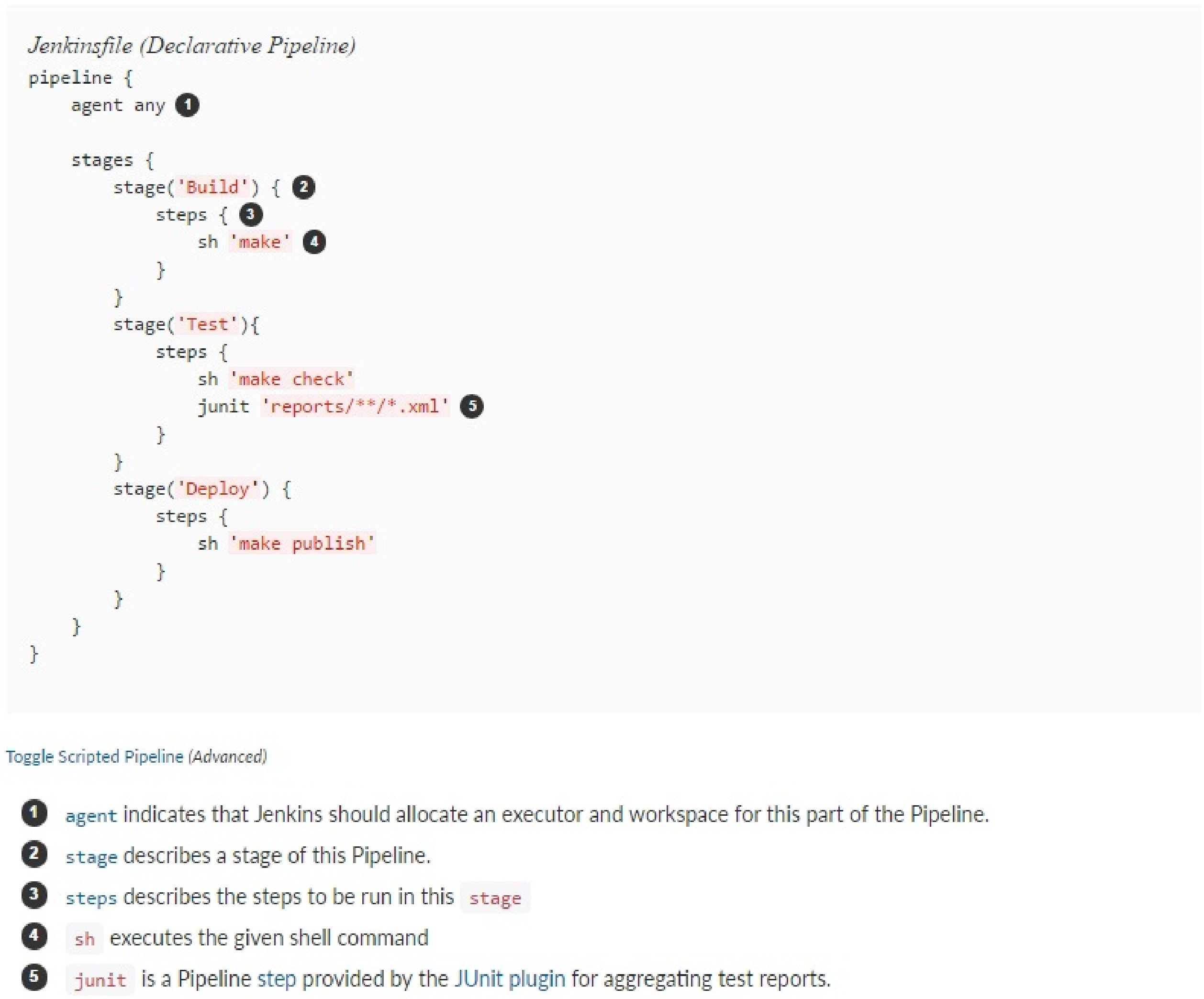
Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via thePipeline DSL.[1]

Declarative and scripted (edit based on the comment):

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

Remember Agent is for declarative pipelines and node is for scripted pipelines.

Typically, this "Pipeline as Code" would be written to a `Jenkinsfile` and checked into a project’s source control repository, for example:



Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any ❶

    stages {
        stage('Build') { ❷
            steps { ❸
                sh 'make' ❹
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml' ❺
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

❶ `agent` indicates that Jenkins should allocate an executor and workspace for this part of the Pipeline.

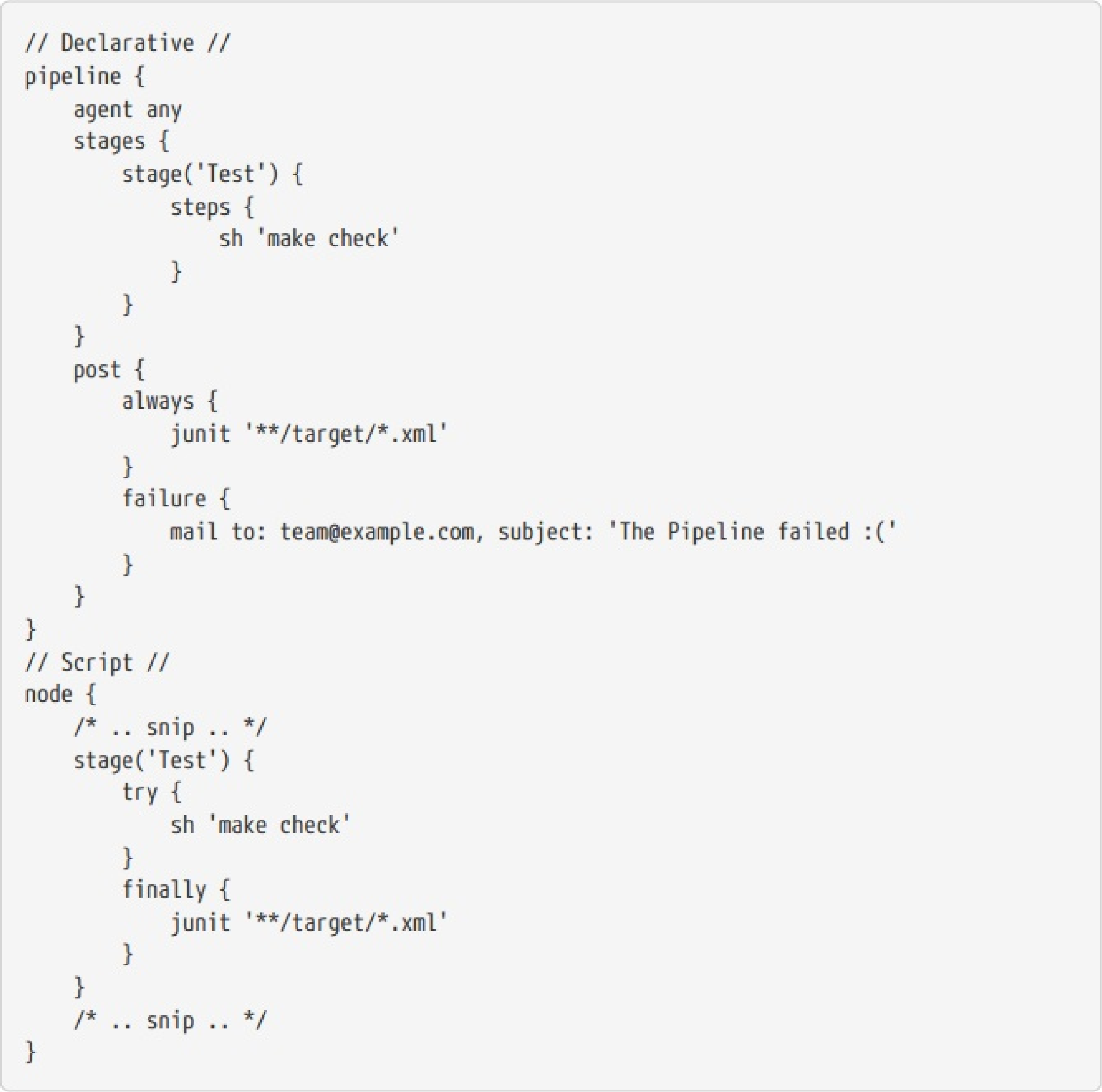
❷ `stage` describes a stage of this Pipeline.

❸ `steps` describes the steps to be run in this `stage`

❹ `sh` executes the given shell command

❺ `junit` is a Pipeline step provided by the JUnit plugin for aggregating test reports.

Difference between Declarative and Scripted Pipeline.



```
// Declarative //
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 :('
        }
    }
}
// Script //
node {
    /* .. snip .. */
    stage('Test') {
        try {
            sh 'make check'
        }
        finally {
            junit '**/target/*.xml'
        }
    }
    /* .. snip .. */
}
```

Why Pipeline?

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

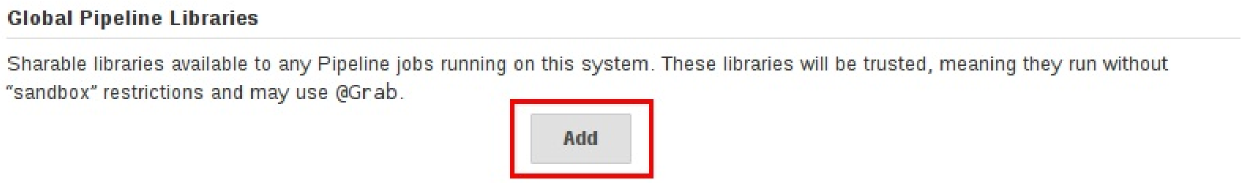
- **Code**: Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
- **Durable**: Pipelines can survive both planned and unplanned restarts of the Jenkins master.

- **Pausable**: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
- **Versatile**: Pipelines support complex real-world continuous delivery requirements, including the ability to fork/join, loop, and perform work in parallel.
- **Extensible**: The Pipeline plugin supports custom extensions to its DSL[1]and multiple options for integration with other plugins.

Reference: https://jenkins.io/doc/book/pipeline/

Global Shared Libraries

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



Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

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

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

Folder-level Shared Libraries

Any Folder created can have Shared Libraries associated with it. This mechanism allows scoping of specific libraries to all the Pipelines inside of the folder or subfolder. Folder-based libraries are not considered "trusted:" they run in the Groovy sandbox just like typical Pipelines. https://jenkins.io/doc/book/pipeline/shared-libraries/

Jenkins Best Practices

- **Polling must die (Trigger a build due to pushes instead of poll the repository every x minutes)**
- **Use Log Rotator (Not using log rotator can result in disk space problems on the master)**
- **Use slaves/labels (Jobs should be defined where to run)**
- **Don’t build on the master (In larger systems, don’t build on the master)**
- **Enforce plugin usage (For example: Timestamp, Mask-Passwords)**
- **Naming sanity (Limit project names to a sane (e.g. alphanumeric) character set)**
- **Analyze Groovy Scripts (For example: Prevent System.exit(0) in System Groovy Scripts)**

Hardware Recommendations

https://go.cloudbees.com/docs/cloudbees-documentation/cookbook/book.html#_architecting_for_scale

Additional Links:

https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+Best+Practices

https://www.slideshare.net/GergelyBrautigam/jenkins-best-practices

API

1. All job (at top level) latest status: JENKINS_URL/api/xml
2. Build numbers and urls for a job: JENKINS_URL/job/jobName/api/xml
3. Build result and details: JENKINS_URL/job/jobName/buildNumber/api/xml
4. Create job: POST to JENKINS_URL/createItem?name=jobName and post config.xml
5. Delete job: POST to JENKINS_URL/job/jobName/doDelete
6. Enable job: POST to JENKINS_URL/job/jobName/enable
7. Disable job: POST to JENKINS_URL/job/jobName/disable

Note: If your Jenkins uses the "Prevent Cross Site Request Forgery exploits" security option (which it should), when you make a POST request, you have to send a CSRF protection token (.crumb) as an HTTP request header.

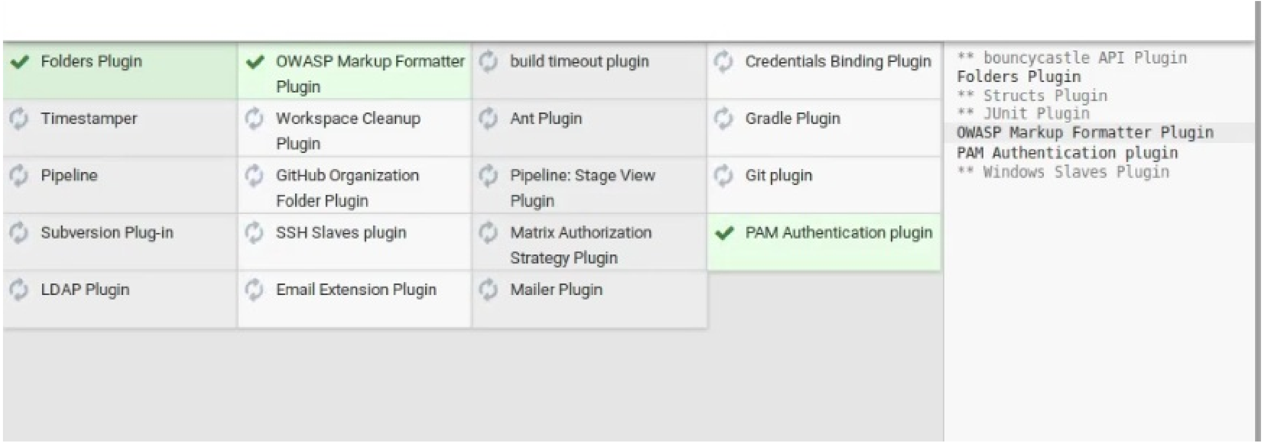
For curl/wget you can obtain the header needed in the request from the

URL JENKINS_URL/crumbIssuer/api/xml (or .../api/json).

Plugins

> ## CJE Exam is based on Suggested plugin list - https://github.com/jenkinsci/jenkins/blob/jenkins-2.19.4/core/src/main/resources/jenkins/install/platform-plugins.json

Suggested Plugin



✓ Folders Plugin	✓ OWASP Markup Formatter Plugin	build timeout plugin	Credentials Binding Plugin
Timestamper	Workspace Cleanup Plugin	Ant Plugin	Gradle Plugin
Pipeline	GitHub Organization Folder Plugin	Pipeline: Stage View Plugin	Git plugin
Subversion Plug-in	SSH Slaves plugin	Matrix Authorization Strategy Plugin	✓ PAM Authentication plugin
LDAP Plugin	Email Extension Plugin	Mailer Plugin	

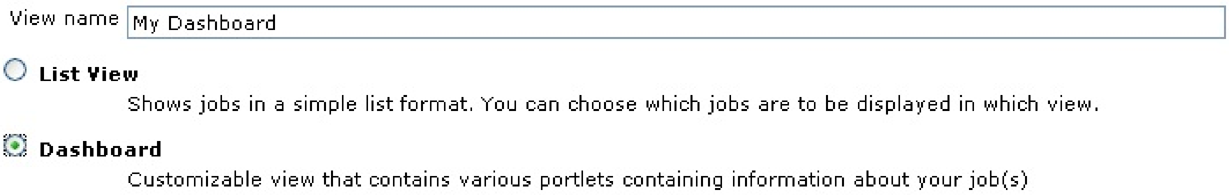
```
** bouncycastle API Plugin
Folders Plugin
** Structs Plugin
** JUnit Plugin
OWASP Markup Formatter Plugin
PAM Authentication plugin
** Windows Slaves Plugin
```

Organization and Administration

Dashboard View

https://wiki.jenkins-ci.org/display/JENKINS/Dashboard+View

This plugin contributes a new view implementation that provides a dashboard / portal-like view for Jenkins.



View name: My Dashboard

○ **List View**
Shows jobs in a simple list format. You can choose which jobs are to be displayed in which view.

● **Dashboard**
Customizable view that contains various portlets containing information about your job(s)

CloudBees Folders Plugin

https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Folders+Plugin

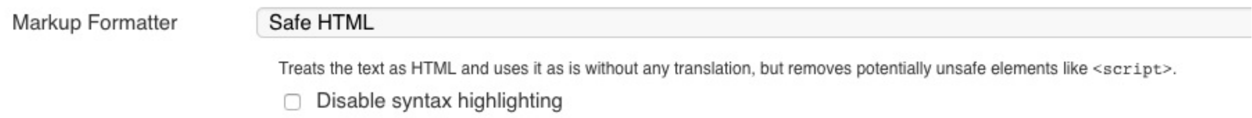
This plugin allows users to create "folders" to organize jobs. Users can define custom taxonomies (e.g. by project type, organization type). Folders are nestable and you can define views within folders.

OWASP Markup Formatter Plugin (antisamy-markup-formatter)

https://wiki.jenkins-ci.org/display/JENKINS/OWASP+Markup+Formatter+Plugin

Uses policy definitions to allow limited HTML markup in user-submitted text.

Configuration
Once installed 'Safe HTML' can be selected as Markup Formatter. User-submitted text will be sanitized by removing dangerous elements.



Markup Formatter: Safe HTML

Treats the text as HTML and uses it as is without any translation, but removes potentially unsafe elements like `<script>`.

□ Disable syntax highlighting

Build Features

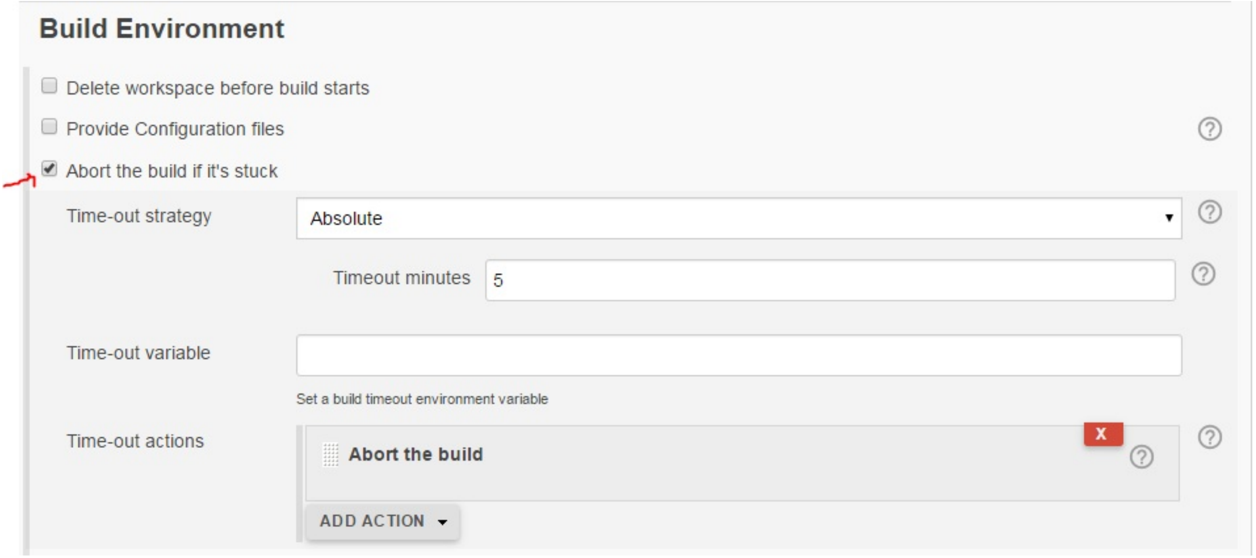
Build timeout

https://wiki.jenkins-ci.org/display/JENKINS/Build-timeout+Plugin

This plugin allows you to automatically abort a build if it's taking too long. Once the timeout is reached, Jenkins behaves as if an invisible hand has clicked the "abort build" button.

Instructions:

After installing the plugin, go to the configure page for your job and select "**Abort the build if it's stuck**".



Build Environment

□ Delete workspace before build starts

□ Provide Configuration files

☑ Abort the build if it's stuck

Time-out strategy: Absolute

Timeout minutes: 5

Time-out variable:

Set a build timeout environment variable

Time-out actions: Abort the build

Credentials Binding

https://wiki.jenkins-ci.org/display/JENKINS/Credentials+Binding+Plugin

Allows credentials to be bound to environment variables for use from miscellaneous build steps.

You may have a keystore for jarsigner, a list of passwords, or other confidential files or strings which you want to be used by a job but which should not be kept in its SCM, or even visible from its config.xml. Saving these files on the server and referring to them by absolute path requires you to have a server login, and does not work on slaves. This plugin gives you an easy way to package up all a job’s secret files and passwords and access them using a single environment variable during the build.

To use, first go to the Credentials link and add items of type*Secret file_and/or_Secret text*. Now in a freestyle job, check the box_Use secret text(s) or file(s)_and add some variable bindings which will use your credentials. The resulting environment variables can be accessed from shell script build steps and so on. (You probably want to start any shell script with `set +x` , or batch script with `@echo off` .JENKINS-14731). https://wiki.jenkins-ci.org/display/JENKINS/Credentials+Binding+Plugin

Time stamper

https://wiki.jenkins-ci.org/display/JENKINS/Timestamper

Adds timestamps to the Console Output.

Instructions

Enable time stamps within the "Build Environment" section of the build's configuration page.

To enable timestamps for multiple builds at once, use the Configuration Slicing Plugin version 1.32 or later.

Customization

- The timestamp format can be configured via the `Configure System` page.
- There is a panel on the left-hand side of the console page which allows either the system clock time or the elapsed time to be displayed.
- The time zone used to display the timestamps can be configured by setting a system parameter as described here: Change time zone

Workspace Cleanup Plugin (ws-cleanup)

This plugin deletes the workspace before the build or when a build is finished and artifacts saved.

Option for deleting workspace **before** build is in **Build Environment** section.
https://wiki.jenkins-ci.org/display/JENKINS/Workspace+Cleanup+Plugin

Build Tools

Ant Plugin

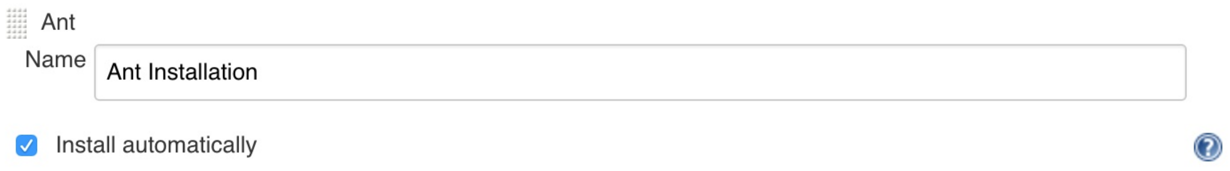
https://wiki.jenkins-ci.org/display/JENKINS/Ant+Plugin

This plugin adds Apache Ant support to Jenkins. This functionality used to be a part of the core, but as of Jenkins 1.431, it was split off into separate plugins.

Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. More generally, Ant can be used to pilot any type of process which can be described in terms of targets and tasks.

Installation

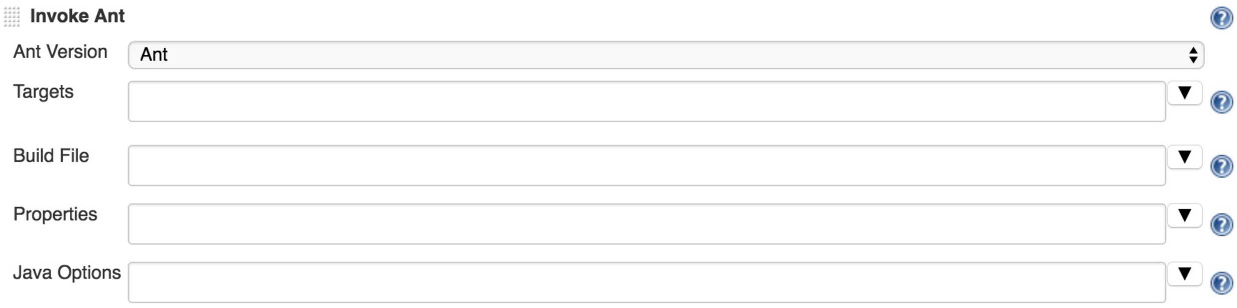
For this plugin to be used, an Ant installation must be specified in the global Jenkins configuration. It can be installed automatically:



Ant

Name: Ant Installation

☑ Install automatically



Invoke Ant

Ant Version: Ant

Targets

Build File

Properties

Java Options

Gradle

https://wiki.jenkins-ci.org/display/JENKINS/Gradle+Plugin

This plugin (Build Tool) makes it possible to invoke a Gradle build script as the main build step.

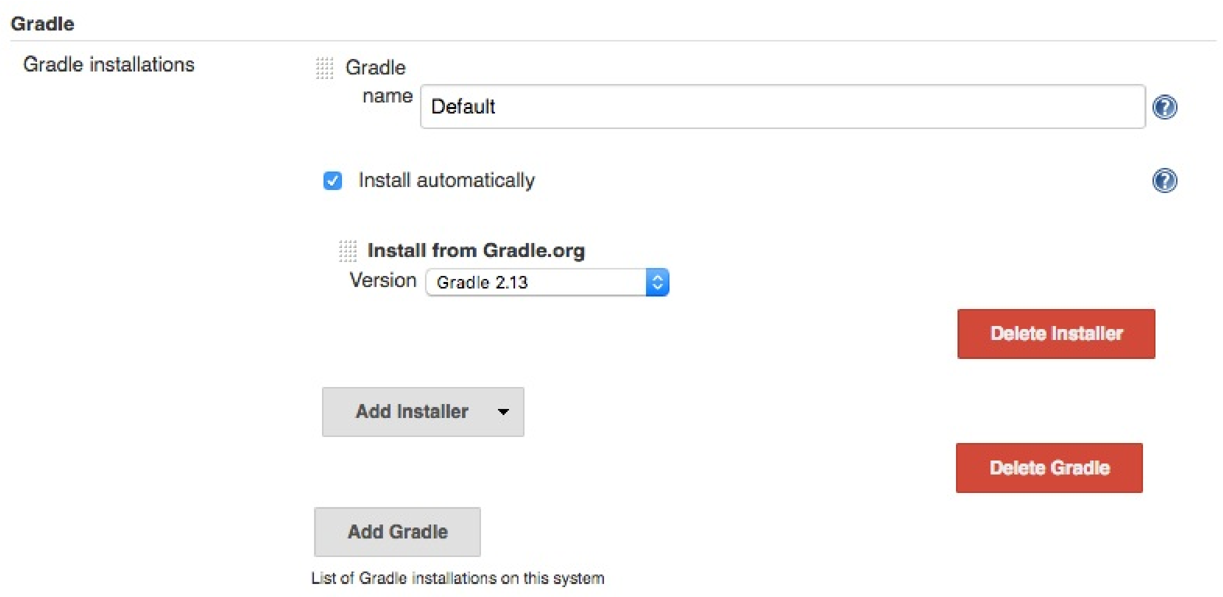
Description

This plugin adds Gradle Support to Jenkins. Gradle is managed as another tool inside Jenkins (the same way as Ant or Maven),
including support for automatic installation and a new build step is provided to execute Gradle tasks.

Configuration

Gradle configuration is performed in the **Configure System**(before Jenkins 2.0) or **Global Tool Configuration**(starting in Jenkins 2.0). In both cases these options reside in the **Manage Jenkins** section.

In the **Gradle** section provided by this plugin, several installations can be configured:



Gradle

Gradle installations

Gradle

name Default

☑ Install automatically

Install from Gradle.org

Version Gradle 2.13

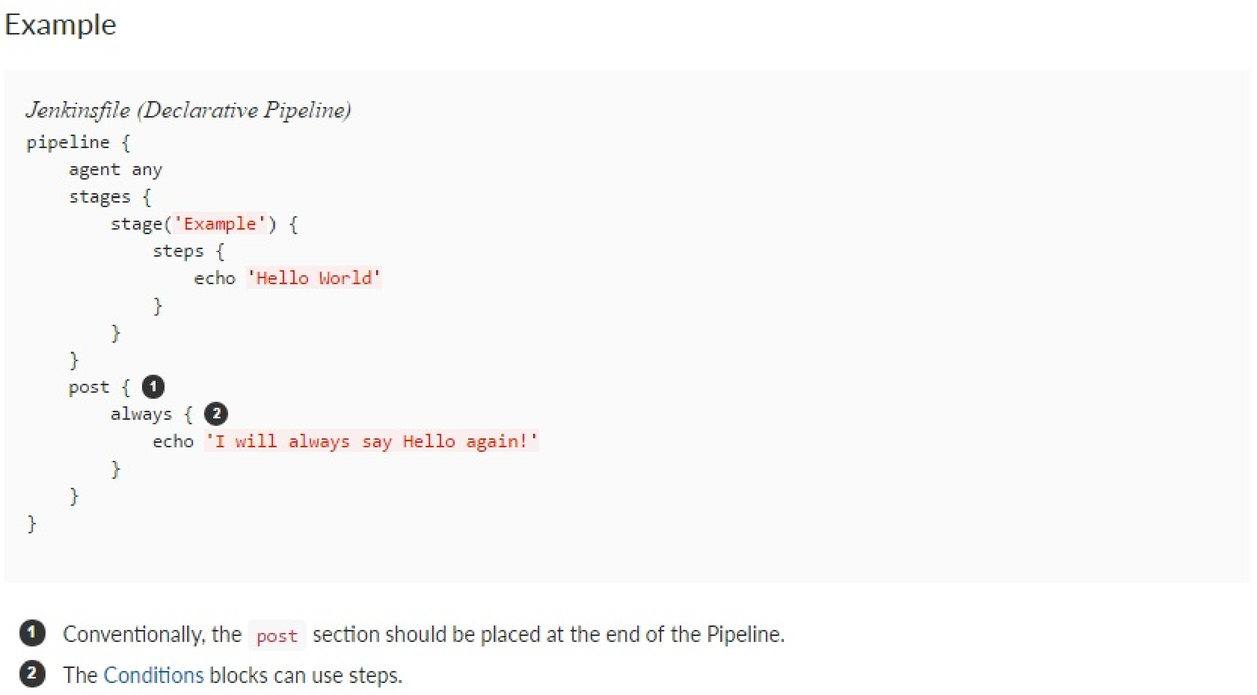
List of Gradle installations on this system

Pipelines and Continuous Delivery

Pipeline Plugin

https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Plugin

A suite of plugins that lets you orchestrate automation, simple or complex.



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) The Conditions blocks can use steps.

Syntax Reference: https://jenkins.io/doc/book/pipeline/syntax/

GitHub Organization Folder Plugin

https://wiki.jenkins-ci.org/display/JENKINS/GitHub+Organization+Folder+Plugin

Pipeline-as-Code support for a whole GitHub organization. Scans all the branches & repositories in GitHub organization and build them via Jenkins pipelines automatically.

Notes

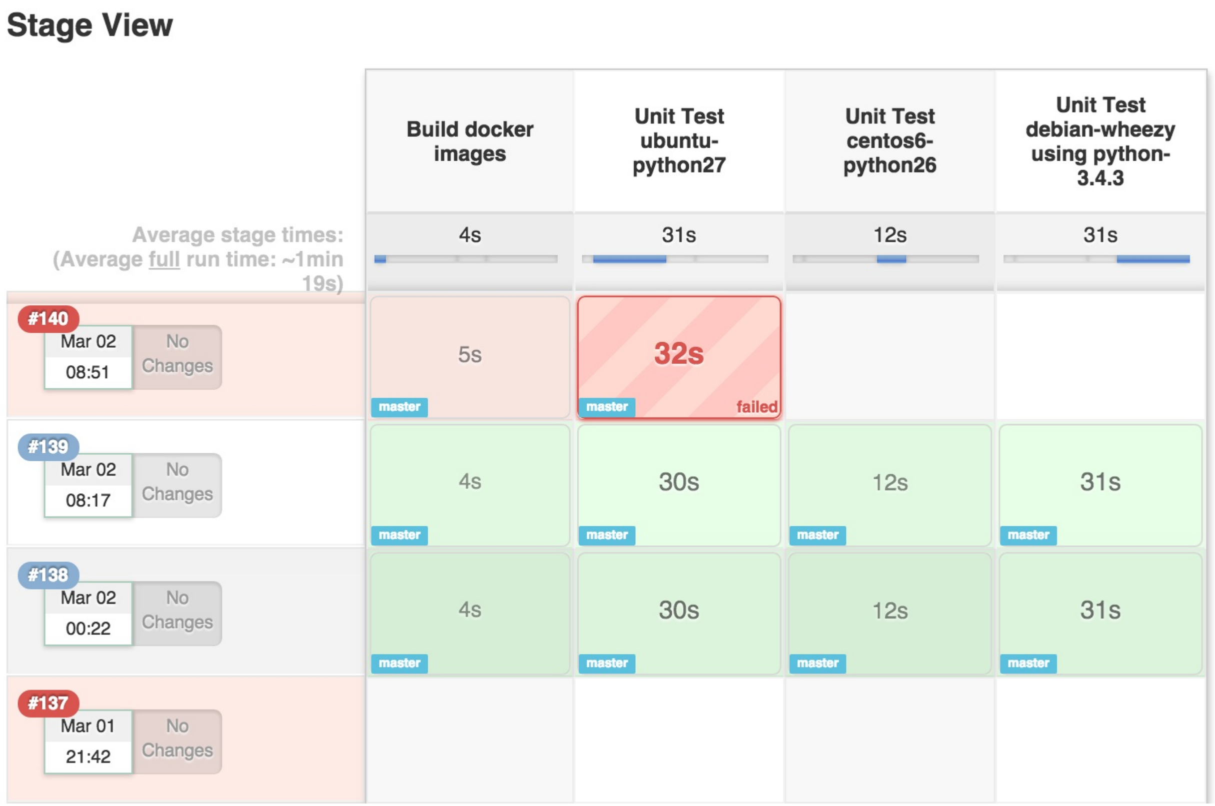
To determine the branch being built - use the environment variable `BRANCH_NAME` - e.g. `${env.BRANCH_NAME }`

Pipeline Stage View Plugin

https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Stage+View+Plugin

When you have complex builds Pipelines, it is useful to be able to see the progress of each stage. The Pipeline Stage View plugin includes an extended visualization of Pipeline build history on the index page of a flow project, under *Stage View*. (You can also click on *Full Stage View* to get a full-screen view.)

To take advantage of this view, you need to define stages in your flow. You can have as many stages as you like, in a linear sequence. (They may be inside other steps such as 'node' if that is convenient.) Give each a short name that will appear in the GUI.



Stage View

	Build docker images	Unit Test ubuntu-python27	Unit Test centos6-python26	Unit Test debian-wheezy using python-3.4.3
Average stage times: (Average full run time: ~1min 19s)	4s	31s	12s	31s
#140 Mar 02 08:51 No Changes	5s master	32s master failed		
#139 Mar 02 08:17 No Changes	4s master	30s master	12s master	31s master
#138 Mar 02 00:22 No Changes	4s master	30s master	12s master	31s master
#137 Mar 01 21:42 No Changes				

Source Code Management

Git Plugin

https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin

This plugin allows use of Git as a build SCM, including repository browsers for several providers. A recent Git runtime is required (1.7.9 minimum, 1.8.x recommended). Interaction with the Git runtime is performed by the use of the Git Client Plugin, which is only tested on official git client. Use exotic installations at your own risk.

Global Settings

In the Configure System page, the Git Plugin provides the following options

- Global Config user.name Value: if provided git config user.name <value> is called before builds. This can be overridden by individual projects.
- Global Config user.email Value: if provided git config user.email <value> is called before builds. This can be overridden by individual projects.
- Create new accounts base on author/committer's email: if checked, upon parsing of git change logs, new user accounts are created on demand for the identified committers / authors in the internal Jenkins database. The e-mail address is used as the id of the account.

A **Repository Browser** can also be configured, which adds links in "changes" views within Jenkins to an external system for browsing the details of those changes. The "Auto" selection attempts to infer the repository browser from other jobs, if supported by the SCM and a job with matching SCM details can be found, though it can also be selected manually.

Subversion plugin

https://wiki.jenkins-ci.org/display/JENKINS/Subversion+Plugin

This plugin adds the Subversion support (via SVNKit) to Jenkins.This plugin is bundled inside `jenkins.war`

Basic Usage

Once this plugin is installed, you'll see Subversion as one of the options in the SCM section of job configurations.

Distributed Builds

SSH Slaves Plugin

https://wiki.jenkins-ci.org/display/JENKINS/SSH+Slaves+plugin

This plugin allows you to manage slaves running on *nix machines over SSH. It adds a new type of slave launch method. This launch method will

- Open a SSH connection to the specified host as the specified username.
- Checks the default version of java for that user.
- [not implemented yet] If the default version is not compatible with Jenkins's slave.jar, tries to find a version of java that is.
- Once it has a suitable version of java, copies the latest slave.jar via SFTP (falling back to scp if SFTP is not available)
- Starts the slave process.

User Management and Security

Matrix Authorization Strategy Plugin

https://wiki.jenkins-ci.org/display/JENKINS/Matrix+Authorization+Strategy+Plugin

Offers matrix-based security authorization strategies (global and per-project).

Also checkout https://wiki.jenkins-ci.org/display/JENKINS/Matrix-based+security for more details.

Matrix-based security is one of the authorization strategies available for securing Jenkins. It allows you to grant specific permissions to users and groups.

PAM Authentication Plugin

https://wiki.jenkins-ci.org/display/JENKINS/PAM+Authentication+Plugin

Adds Unix Pluggable Authentication Module (PAM) support to Jenkins.

LDAP Plugin

https://wiki.jenkins-ci.org/display/JENKINS/LDAP+Plugin

This plugin provides yet another way of authenticating users using LDAP. It can be used with LDAP servers like Active Directory or OpenLDAP among others.

Notifications and Publishing

Email-ext plugin

https://wiki.jenkins-ci.org/display/JENKINS/Email-ext+plugin

This plugin allows you to configure every aspect of email notifications. You can customize when an email is sent, who should receive it, and what the email says.

This plugin extends Jenkins built in email notification functionality by giving you more control. It provides customization of 3 areas.

1. **Triggers** - Select the conditions that should cause an email notification to be sent.
2. **Content** - Specify the content of each triggered email's subject and body.
3. **Recipients** - Specify who should receive an email when it is triggered.

Mailer plugin

https://wiki.jenkins-ci.org/display/JENKINS/Mailer

This plugin allows you to configure email notifications for build results. This is a break-out of the original core based email component.

Usage

E-Mail notifications are configured in jobs by adding an **E-mail notification Post-build Action**. If configured, Jenkins will send out an e-mail to the specified recipients when a certain important event occurs:

1. Every failed build triggers a new e-mail.
2. A successful build after a failed (or unstable) build triggers a new e-mail, indicating that a crisis is over.
3. An unstable build after a successful build triggers a new e-mail, indicating that there's a regression.
4. Unless configured, every unstable build triggers a new e-mail, indicating that regression is still there.

The **Recipients** field must contain a whitespace or comma-separated list of recipient addresses. May reference build parameters like `$PARAM` .

Additional options include:

- **Send e-mail for every unstable build** : if checked, notifications will be sent for every unstable build and not only int first build after a successful one.

- **Send separate e-mails to individuals who broke the build** : if checked, the notification e-mail will be sent to individuals who have committed changes for the broken build (by assuming that those changes broke the build).

  If e-mail addresses are also specified in the recipient list, then both the individuals as well as the specified addresses get the notification e-mail. If the recipient list is empty, then only the individuals will receive e-mails.

Build Periodically and Trigger Schedule (CRON Syntax)

This field follows the syntax of cron (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

```
MINUTE HOUR DOM MONTH DOW
```

MINUTE	Minutes within the hour (0–59)
HOUR	The hour of the day (0–23)
DOM	The day of the month (1–31)
MONTH	The month (1–12)
DOW	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.)

1. 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.

Examples:

```
# every fifteen minutes (perhaps at :07, :22, :37, :52)
H/15 * * * *
# every ten minutes in the first half of every hour (three times, perhaps at :04, :14,
 :24)
H(0-29)/10 * * * *
# once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing a
t 3:45 PM every weekday.
45 9-16/2 * * 1-5
# once in every two hours slot between 9 AM and 5 PM every weekday (perhaps at 10:38 A
M, 12:38 PM, 2:38 PM, 4:38 PM)
H H(9-16)/2 * * 1-5
# once a day on the 1st and 15th of every month except December
H H 1,15 1-11 *
If you want to schedule your build every 5 minutes, this will do the job :*/5 * * * *
If you want to schedule your build every day at 8h00, this will do the job :0 8 * * *
```

Git Hooks with Git and Github Plugin*

Git plugin

Jenkins:

1. Check Poll SCM
2. No need to add anything in the schedule.

Github:

1. Project->Setting->Integration & Services->Add Services-> Jenkins (git plugin)

2. Jenkins url: http://jenkins_url:8080

Github plugin

Jenkins:

1. Project-> Check Github hook trigger for GitSCM polling

Github:

1. Project->Setting->Integration & Services->Add Services-> Jenkins (Github plugin)
2. Jenkins url: http://jenkins_url:8080/github_webhook/
3. Make sure Active box is checked.
4. Can test the integration by clicking on the Test Service.

In Jenkins, you can now check the Github Hook Log.

References

Certified Jenkins Engineer (CJE) – 2017

https://www.cloudbees.com/sites/default/files/cje-study-guide-2017.pdf

Jenkins CCJPE Course by DevOps Library

https://www.youtube.com/watch?v=IvcIlMK49Cg&list=PL6TwUbrFsOuN-db811WkXF1hwGTexiiOH

Declarative Pipelines in Jenkins: A new way to define your pipelines by Jenkins CI

https://www.youtube.com/watch?v=utztUGvZ_EA&index=1&list=WL&t=165s

Jenkins Certified Engineer: Folders

http://www.greenreedtech.com/jenkins-certified-engineer-folders/

Certified Jenkins Engineer (CJE) Flashcard

https://quizlet.com/174152296/certified-jenkins-engineer-cje-flash-cards/

Jenkins User Handbook

https://jenkins.io/user-handbook.pdf

CloudBees Jenkins Solutions Guides

https://go.cloudbees.com/doc/index.html

Jeanne’s experiences with the Jenkins Certification (beta exam)

https://www.selikoff.net/2016/02/27/jeannes-experiences-with-the-jenkins-certification-beta-exam/

http://www.selikoff.net/wp-content/uploads/2016/02/Study-Notes.docx

https://ankiweb.net/shared/info/1993369597

Jenkins the Definitive Guide by John Ferguson Smart (Book)

Certified Jenkins Engineer (CJE) Course - 2017

https://linuxacademy.com

Best Practices for Scalable Pipeline Code

https://jenkins.io/blog/2017/02/01/pipeline-scalability-best-practice/

Top 10 Best Practices for Jenkins Pipeline Plugin

https://www.cloudbees.com/blog/top-10-best-practices-jenkins-pipeline-plugin

My Blog

http://blog.muralibala.com/category/jenkins/