Studying for the Jenkins Engineer certification

memefected

Published with GitBook

Table of Contents

Introduction 0
Plugins 1
Section 1 2
Various notes 3
Section 1 Bibliography 4

My Awesome Book

This file file serves as your book's preface, a great place to describe your book's content and ideas.

Plugins

This exam will primarily cover questions about a “base” Jenkins installation, but knowledge of the following plugins will also be covered. Candidates are expected to know the functionality/uses of these plugins but will not be tested on detailed usage:

Amazon EC2 Plugin

Allow Jenkins to start slaves on EC2 or Eucalyptus on demand, and kill them as they get unused. With this plugin, if Jenkins notices that your build cluster is overloaded, it'll start instances using the EC2 API and automatically connect them as Jenkins slaves. When the load goes down, excessive EC2 instances will be terminated. This set up allows you to maintain a small in-house cluster, then spill the spiky build/test loads into EC2 or another EC2 compatible cloud.

Build Pipeline Plugin

This plugin provides a Build Pipeline View of upstream and downstream connected jobs that typically form a build pipeline. In addition, it offers the ability to define manual triggers for jobs that require intervention prior to execution, e.g. an approval process outside of Jenkins.

Continuous Integration has become a widely adopted practice in modern software development. Jenkins & Hudson are great tools for supporting Continuous Integration.

Taking it to the next level: Continuous integration can become the centerpiece of your deployment pipeline, orchestrating the promotion of a version of software through quality gates and into production. By extending the concepts of CI you can create a chain of jobs each one subjecting your build to quality assurance steps. These QA steps may be a combination of manual and automated steps. Once a build has passed all these, it can be automatically deployed into production.

In order to better support this process, we have developed the Build Pipeline Plugin. This gives the ability to form a chain of jobs based on their upstream/downstream dependencies. Downstream jobs may, as per the default behaviors, be triggered automatically ,or by a suitable authorized user manually triggering it.

You can also see a history of pipelines in a view, the current status and where each version got to in the chain based on its revision number in VCS.

CloudBees Docker Build and Publish Plugin

This Jenkins plugin allows to build Docker images on a Docker server and then publish them to Docker Hub or other Docker registries.

Features summary:

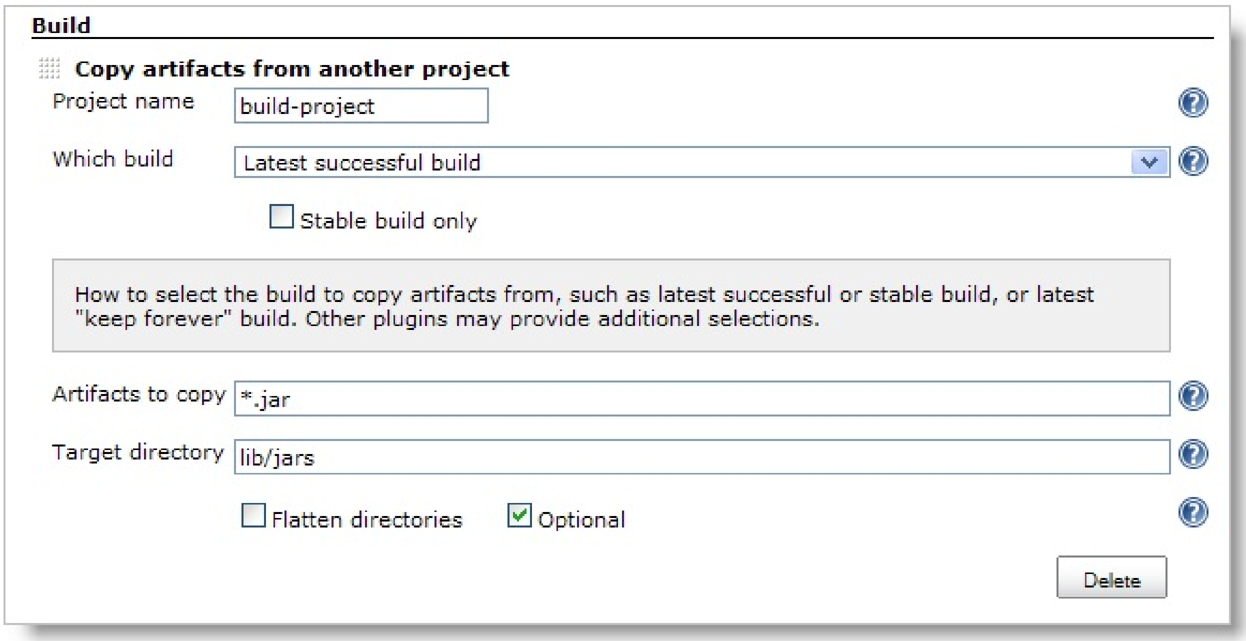
- The entire functionality is implemented as a single build step
- Support of `Dockerfile` specifications
- Publishing to docker index/registry including Docker Hub
- Credentials support for Docker servers and registries (provided by Docker Commons Plugin)
- On-demand tagging of built images
- On-demand fingerprinting of built images
- Image clean build with `--no-cache` option (rebuild of all steps from Dockerfile steps)

CloudBees Folders Plugin

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

Copy Artifact Plugin

Adds a build step to copy artifacts from another project. The plugin lets you specify which build to copy artifacts from (e.g. the last successful/stable build, by build number, or by a build parameter). You can also control the copying process by filtering the files being copied, specifying a destination directory within the target project, etc. Click the help icon on each field to learn the details, such as selecting Maven or multiconfiguration projects or using build parameters. You can also copy from the workspace of the latest completed build of the source project, instead of its artifacts. All artifacts copied are automatically fingerprinted for you.


Build

Copy artifacts from another project

Project name: build-project

Which build: Latest successful build

☐ Stable build only

How to select the build to copy artifacts from, such as latest successful or stable build, or latest "keep forever" build. Other plugins may provide additional selections.

Artifacts to copy: *.jar

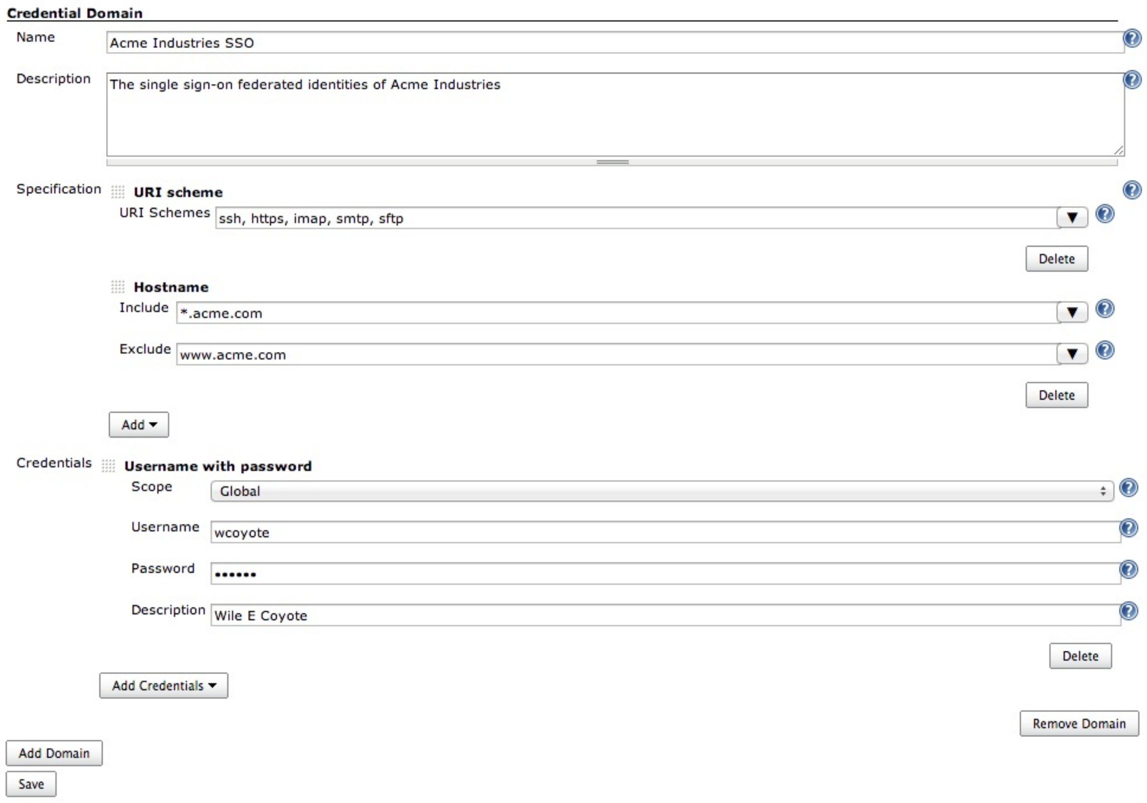
Target directory: lib/jars

☐ Flatten directories ☑ Optional

Credentials Plugin

This plugin allows you to store credentials in Jenkins. The credentials plugin provides a standardized API for other plugins to store and retrieve different types of credentials. User visible features are:

- A “Manage Credentials” screen on the “Manage Jenkins” screen allowing you to manage system and global credentials.
- If you are using Jenkins security, when you go to “Users” / your username / “Configure” you would see the option to manage personal credentials.
- Anywhere those credentials are needed, there is a drop down list of the appropriate available credentials, and you just select the appropriate one.
- When the time comes to change the password, you just change it once.



Credential Domain

Name: Acme Industries SSO

Description: The single sign-on federated identities of Acme Industries

Specification: URI scheme

URI Schemes: ssh, https, imap, smtp, sftp

Hostname

Include: *.acme.com

Exclude: www.acme.com

Credentials: Username with password

Scope: Global

Username: wcoyote

Password: ••••••

Description: Wile E Coyote

Plugins that provide credentials

- Page: Azure PublisherSettings Credentials Plugin — This plugin manage Azure PublisherSettings using Jenkins Crendentials API.
- Page: Config File Provider Plugin — Adds the ability to provide configuration files (i.e., settings.xml for maven, XML, groovy, custom files, etc.) loaded through the Jenkins UI which will be copied to the job's workspace.
- Page: Docker Commons Plugin — APIs for using Docker from other plugins.
- Page: Google Container Registry Auth Plugin — This plugin allows the credential provider to use Google Cloud Platform OAuth Credentials (provided by the Google OAuth Credentials plugin) to access Docker images from Google Container Registry (GCR).
- Page: Google OAuth Plugin — This plugin implements the OAuth Credentials interfaces for surfacing Google Service Accounts to Jenkins.
- Page: Plain Credentials Plugin — Allows use of plain strings and files as credentials.
- Page: Pushbullet Credentials Plugin — This plugin integrates Jenkins with Pushbullet.
- Page: Rally plugin — This plugin allows pushing information to rally
- Page: SSH Credentials Plugin — This plugin allows you to store SSH credentials in Jenkins.

Plugins that consume credentials

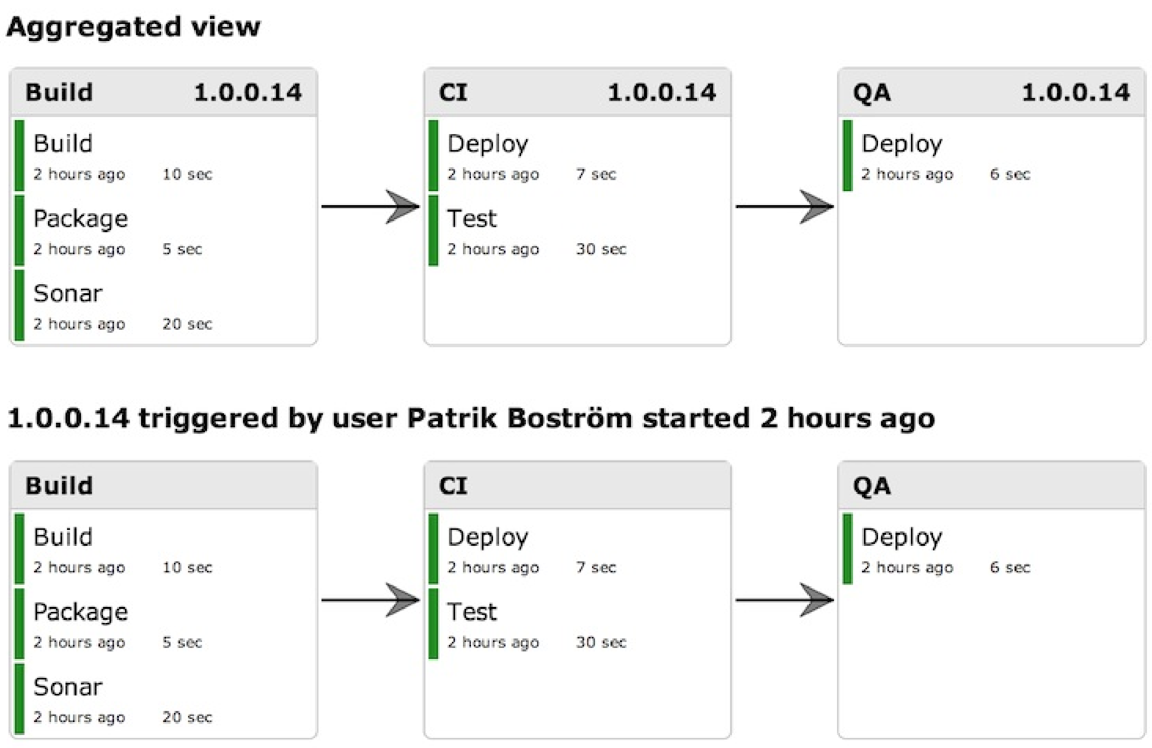
- Page: Ansible Plugin — This plugin allows to execute Ansible tasks as a job build step.
- Page: CloudBees Docker Build and Publish plugin

- Page: Config File Provider Plugin — Adds the ability to provide configuration files (i.e., settings.xml for maven, XML, groovy, custom files, etc.) loaded through the Jenkins UI which will be copied to the job's workspace.
- Page: Credentials Binding Plugin — Allows credentials to be bound to environment variables for use from miscellaneous build steps.
- Page: Docker build step plugin — This plugin allows to add various Docker commands into your job as a build step
- Page: GitHub pull request builder plugin — This plugin builds pull requests in github and report results.
- Page: New Relic Deployment Notifier Plugin — Jenkins plugin to notify New Relic about deployments.
- Page: Phabricator Differential Plugin — Integrates with Phabricator's Differential and Harbormaster apps
- Page: Pushbullet Notifier Plugin — This plugin integrates Jenkins with Pushbullet.
- Page: Rally plugin — This plugin allows pushing information to rally
- Page: SSH Agent Plugin — This plugin allows you to provide SSH credentials to builds via a ssh-agent in Jenkins.
- Page: SSH Slaves plugin — This plugin allows you to manage slaves running on *nix machines over SSH.
- Page: Subversion Plugin — This plugin adds the Subversion support (via SVNKit) to Jenkins.
- Page: XL TestView Plugin — The XL TestView Plugin integrates Jenkins with XebiaLabs XL TestView

Delivery Pipeline Plugin

Visualisation of Delivery/Build Pipelines, renders pipelines based on upstream/downstream jobs. Full screen view for information radiators.

In Continuous Delivery feedback and visualisation of the delivery process is one of the most important areas. When using Jenkins as a build server it is now possible with the Delivery Pipeline Plugin to visualise one or more Delivery Pipelines in the same view even in full screen.



Jenkins jobs is tagged with a stage and a taskname. In the screenshot above the pipeline consists of four stages called Build, CI, QA and Production. The second stage is called CI and consists of two tasks called Deploy and Test.

The plugin requires Jenkins jobs with downstream/upstream relationships. For automatic steps use the Parameterized Trigger Plugin or for manual steps use the Build Pipeline Plugin manual trigger.

Aggregated view shows the latest version for each stage.

Disk Usage Plugin

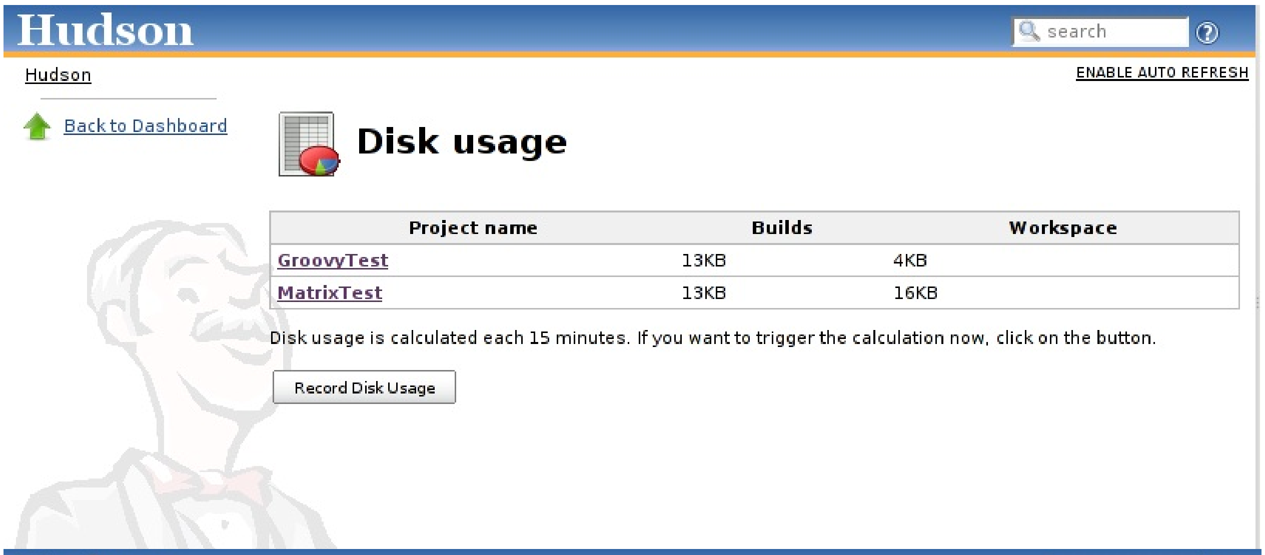
Configuration

Showing disk usage trend graph is optional - unselect the "Show disk usage trend graph" checkbox on the global configuration page ("Manage Jenkins" -> "System configuration") if you don't want to see the graph on the project page.

Usage

When you install this plugin, disk usage is calculated each 60 minutes. You can see project list with occupied disk space by going to the "Disk Usage" page in the management section (Dashboard -> "Manage Jenkins" -> "Disk Usage"). The same page also allows you to schedule disk usage calculation immediately.

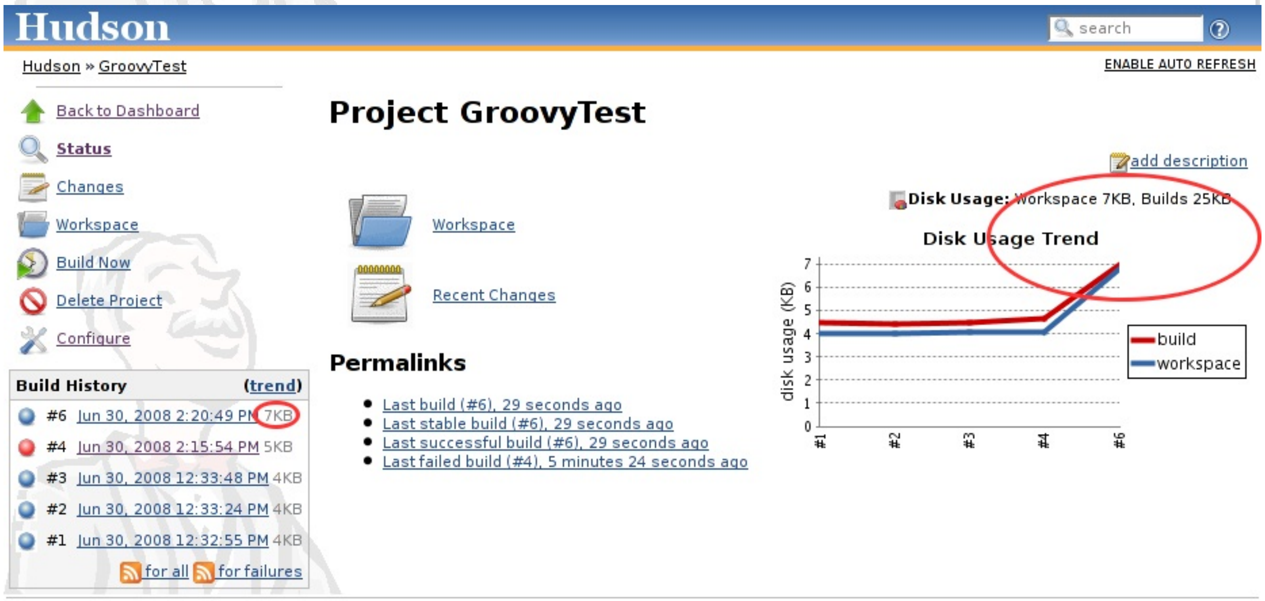
When you install this plugin, disk usage is calculated each 60 minutes. You can see project list with occupied disk space by going to the "Disk Usage" page in the management section (Dashboard -> "Manage Jenkins" -> "Disk Usage"). The same page also allows you to schedule disk usage calculation immediately.



Disk usage

Project name	Builds	Workspace
GroovyTest	13KB	4KB
MatrixTest	13KB	16KB

Disk usage is calculated each 15 minutes. If you want to trigger the calculation now, click on the button.



Project GroovyTest

Disk Usage: Workspace 7KB, Builds 25KB

Build History (trend)

#6	Jun 30, 2008 2:20:49 PM	7KB
#4	Jun 30, 2008 2:15:54 PM	5KB
#3	Jun 30, 2008 12:33:48 PM	4KB
#2	Jun 30, 2008 12:33:24 PM	4KB
#1	Jun 30, 2008 12:32:55 PM	4KB

for all for failures

Permalinks

- Last build (#6), 29 seconds ago
- Last stable build (#6), 29 seconds ago
- Last successful build (#6), 29 seconds ago
- Last failed build (#4), 5 minutes 24 seconds ago

Docker Plugin

The aim of the docker plugin is to be able to use a docker host to dynamically provision a slave, run a single build, then tear-down that slave.

Optionally, the container can be committed, so that (for example) manual QA could be performed by the container being imported into a local docker provider, and run from there.

A quick setup is :

- get a docker environment running
- follow the instructions for creating a system that has an ssh server installed, and a JDK

- store that image with a known ID (e.g: Jenkins) so that it appears in the output of "docker images" command

Creating a docker image

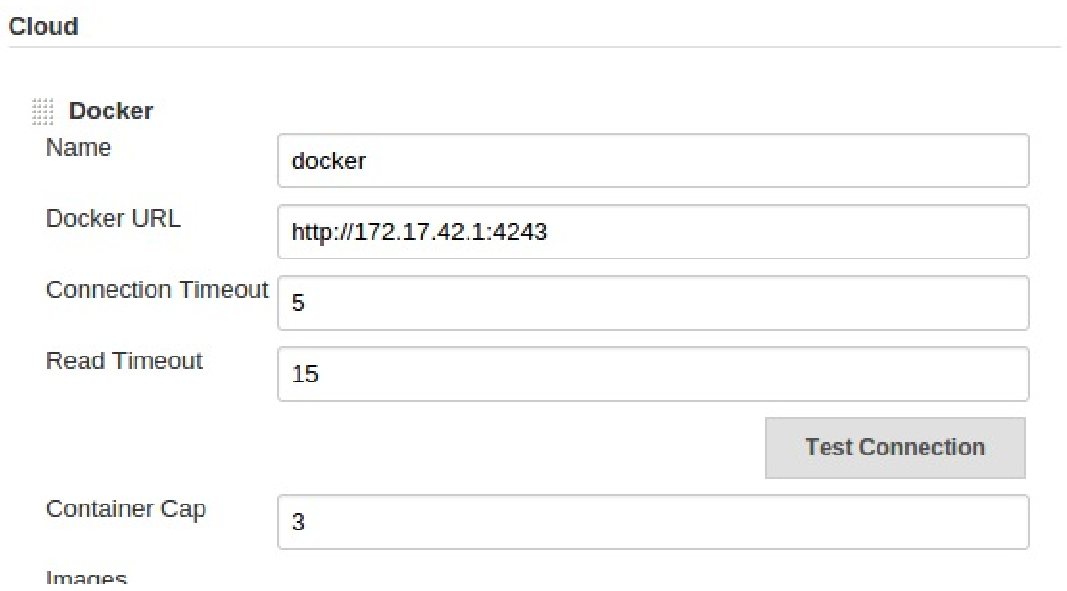
You need a docker image that has, as a minimum, an ssh server installed. You probably want a JDK, and you will also want a 'jenkins' user that can log in.

Once the container has been created, you need to commit it with a name to be used later.

You may wish to periodically update your build image -- e.g: if you are using maven, then it would be advantageous to update your local maven repository with released artifacts, to prevent having to download them again (and thus speeding up your builds).

Configuration

Docker appears in the 'Cloud' section of the Jenkins configuration, select "Docker" from the "Add a new cloud" drop down menu.


Cloud

Docker

Name	docker
Docker URL	http://172.17.42.1:4243
Connection Timeout	5
Read Timeout	15
Container Cap	3

The docker cloud configuration has the following options:

Name	Choose a name for this Docker cloud provider
Docker URL	The URL to use to access your Docker server API (e.g: http://172.16.42.43:4243)
Connection Timeout	
Read Timeout	
Container	Cap The maximum number of containers that this provider is allowed to have running, 0 disables capacity

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.

Fingerprint Plugin

When you have interdependent projects on Jenkins, it often becomes hard to keep track of which version of this is used by which version of that. Jenkins supports "file fingerprinting" to simplify this.

For example, suppose you have the TOP project that depends on the MIDDLE project, which in turn depends on the BOTTOM project. You are working on the BOTTOM project. The TOP team reported that bottom.jar that they are using causes an NPE, which you (a member of the BOTTOM team) thought you fixed in BOTTOM #32. Jenkins can tell you which MIDDLE builds and TOP builds are using (or not using) your bottom.jar #32.

How do I set it up?

To make this work, all the relevant projects need to be configured to record fingerprints of the jar files (in this case, bottom.jar.)

For example, if you just want to track which BOTTOM builds are used by which TOP builds, configure TOP and BOTTOM to record bottom.jar. If you also want to know which MIDDLE builds are using which bottom.jar, also configure MIDDLE.

Since recording fingerprints is a cheap operation, the simplest thing to do is just blindly record all fingerprints of the followings:

1. `jar` files that your project produce

2. `jar` files that your project rely on

The disk usage is affected more by the number of files fingerprinted, as opposed to the size of files or the number of builds they are used. So unless you have a plenty of disk space, you don't want to fingerprint `**/*`.

To set-up file fingerprinting, go to your project and click "configure" in the left navigation bar. Click on the button to add a post-build action. From the list that appears, select "Record fingerprints of files to track usage". The post-build action configuration fields provide you with a pattern option to match the files you want to fingerprint as well as a couple check-box selections to do your file fingerprinting.

Maven job type does this automatically for its dependencies and artifacts.

How does it work?

The fingerprint of a file is simply a MD5 checksum. Jenkins maintains a database of md5sum, and for each md5sum, Jenkins records which builds of which projects used. This database is updated every time a build runs and files are fingerprinted.

To avoid the excessive disk usage, Jenkins does not store the actual file. Instead, it just stores md5sum and their usages. These files can be seen in $JENKINS_HOME/fingerprints.

Plugins can store additional information to these records. For example, Deployment Notification Plugin tracks files deployed on servers via chef/puppet through fingerprints.

Git Plugin

This plugin allows use of Git as a build SCM. A recent Git runtime is required (1.7.9 minimum, 1.8.x recommended). Plugin is only tested on official git client. Use exotic installations at your own risk.

This is an extensive plugin, pretty important too. It sports extensive documentation as well.

Mailer Plugin

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

Configuration

In order to be able to send E-Mail notifications mail server configuration must be introduced in the **Manage Jenkins** page, **E-mail Notification** section. Available options are:

- **SMTP server**: Name of the mail server. If empty the system will try to use the default server (which is normally the one running on localhost). Jenkins uses JavaMail for sending out e-mails, and JavaMail allows additional settings to be given as system properties to the container. See this document for possible values and effects.
- **Default user e-mail suffix**: If your users' e-mail addresses can be computed automatically by simply adding a suffix, then specify that suffix if this field. Otherwise leave it empty. Note that users can always override the e-mail address selectively. For example, if this field is set to @acme.org, then user foo will by default get the e-mail address foo@acme.org.

There are some advanced options as well:

- **Use SMTP Authentication**: check this option to use SMTP authentication when sending out e-mails. If your environment requires the use of SMTP authentication, specify the user name and the password in the fields shown when this option is checked.
- **Use SSL**: Whether or not to use SSL for connecting to the SMTP server. Defaults to port 465. Other advanced configurations can be done by setting system properties. See this document for possible values and effects.
- **SMTP Port**: Port number for the mail server. Leave it empty to use the default port for the protocol (465 if using SSL, 25 if not).
- **Reply-To Address**: Address to include in the Reply-To header. As of version 1.16, only one address is allowed.
- **Charset**: character set to use to construct the message.

In order to test the configuration, you can check the Test configuration by sending test e-mail checkbox, provide a destination address at the Test e-mail recipient field and clicking the Test configuration button.

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.

Instant Messaging Plugin

This plugin provides generic support for build notifications and a 'bot' via instant messaging protocols. This plugin itself is of no use for end users. Please use one of the derived plugins like the Jabber, Skype or the IRC plugin!

IRC Plugin

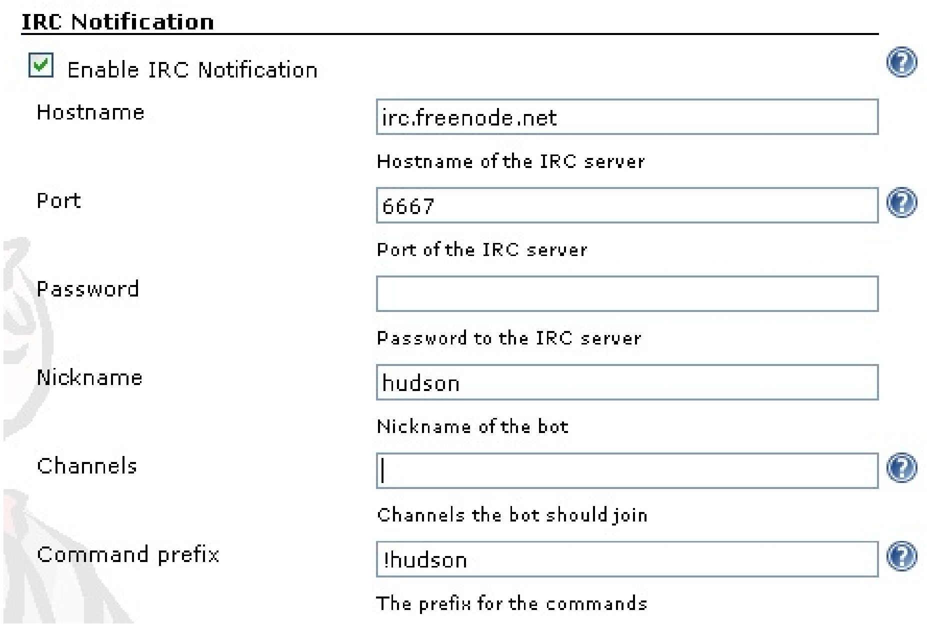
This plugin enables Jenkins to send build notífications via IRC and lets you interact with Jenkins via an IRC bot. Note that you also need to install the instant-messaging plugin .

Installation Requirements

This plugin needs the instant-messaging plugin.

Usage

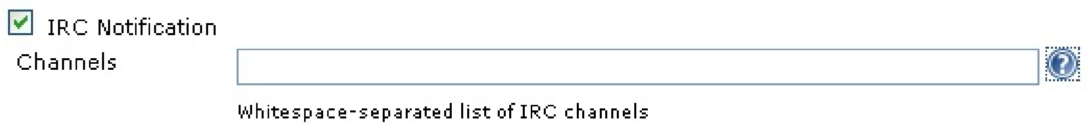
When you install this plugin, your Hudson configuration page gets additional "IRC Notification" option as illustrated below:



IRC Notification

✓ Enable IRC Notification	
Hostname	irc.freenode.net Hostname of the IRC server
Port	6667 Port of the IRC server
Password	 Password to the IRC server
Nickname	hudson Nickname of the bot
Channels	 Channels the bot should join
Command prefix	!hudson The prefix for the commands

In addition, each project should add a "Post-build Actions"> "IRC Notification" configuration as illustrated below:



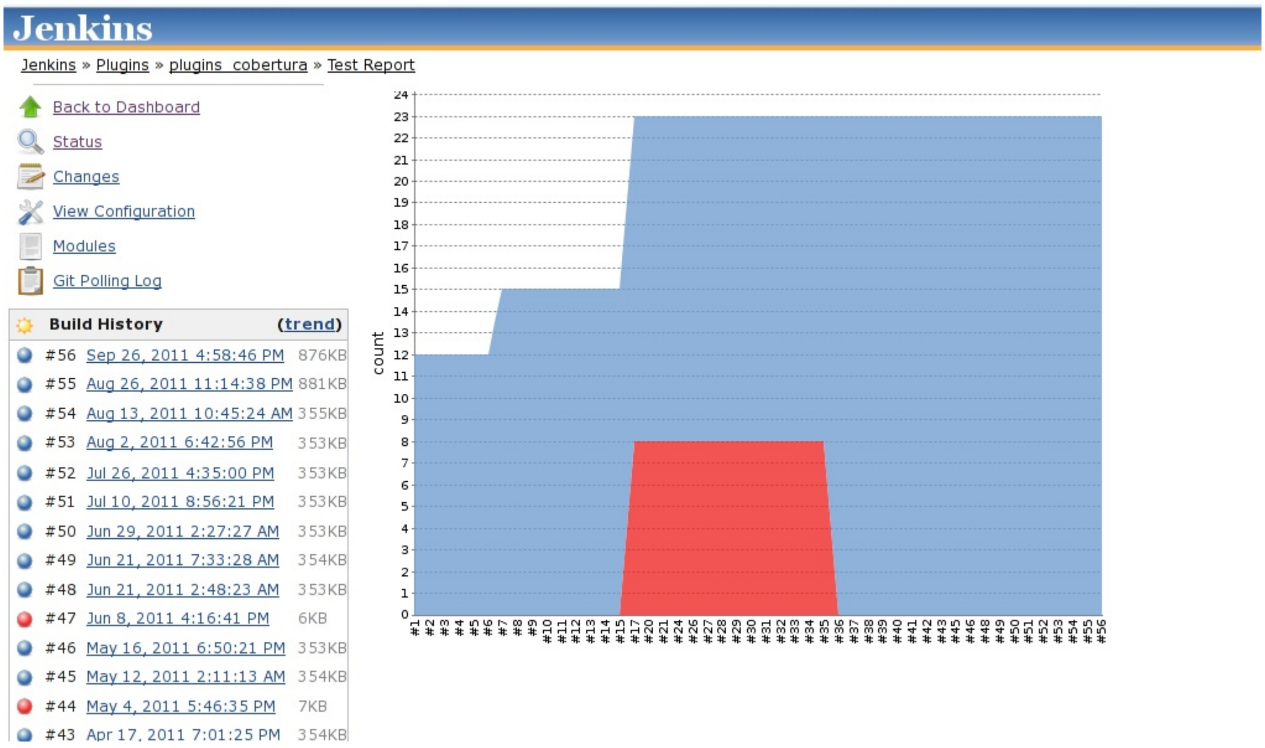
✓ IRC Notification	
Channels	 Whitespace-separated list of IRC channels

For the project configuration, leave the Channels blank to default to the channels defined in the master IRC configration.

JUnit Plugin

Allows JUnit-format test results to be published.

Sample UI: JUnit graph



Jenkins » Plugins » plugins_cobertura » Test Report

Back to Dashboard
Status
Changes
View Configuration
Modules
Git Polling Log

Build History		(trend)
#56	Sep 26, 2011 4:58:46 PM	876KB
#55	Aug 26, 2011 11:14:38 PM	881KB
#54	Aug 13, 2011 10:45:24 AM	355KB
#53	Aug 2, 2011 6:42:56 PM	353KB
#52	Jul 26, 2011 4:35:00 PM	353KB
#51	Jul 10, 2011 8:56:21 PM	353KB
#50	Jun 29, 2011 2:27:27 AM	353KB
#49	Jun 21, 2011 7:33:28 AM	354KB
#48	Jun 21, 2011 2:48:23 AM	353KB
#47	Jun 8, 2011 4:16:41 PM	6KB
#46	May 16, 2011 6:50:21 PM	353KB
#45	May 12, 2011 2:11:13 AM	354KB
#44	May 4, 2011 5:46:35 PM	7KB
#43	Apr 17, 2011 7:01:25 PM	354KB

Jabber Plugin

Integrates Jenkins with the Jabber/XMPP instant messaging protocol. Note that you also need to install the instant-messaging plugin. This plugin enables Jenkins to send build notifications via Jabber, as well as let users talk to Jenkins via a 'bot' to run commands, query build status etc..

Installation Requirements

This plugin needs the instant-messaging plugin

Matrix Project Plugin

See Building a matrix project for user information.

Extensions See all extension points here: Matrix Project Extension Points

Matrix Axis Extension

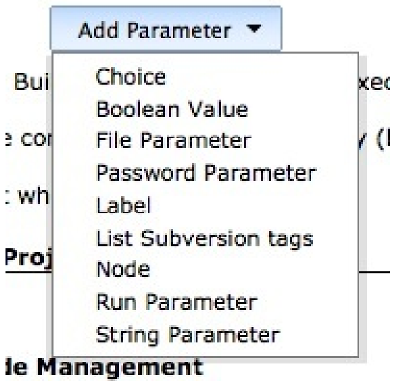
- Page: DynamicAxis Plugin — This plugin allows you to define a matrix build axis that is dynamically populated from an environment variable:
- Page: Matrix Groovy Execution Strategy Plugin — A plugin to decide the execution order and valid combinations of matrix projects.

- Page: NodeLabel Parameter Plugin — This plugin adds two new parameter types to job configuration - node and label, this allows to dynamically select the node where a job/project should be executed.
- Page: Selenium Axis Plugin — Creates an axis based on a local Selenium grid and also build against the SauceLabs Selenium capability at the same time.
- Page: Yaml Axis Plugin — Matrix project axis creation and exclusion plugin using yaml file (It's similar to .travis.yml)
- Page: Sauce OnDemand Plugin — This plugin allows you to integrate Sauce Selenium Testing with Jenkins.

NodeLabel Parameter Plugin

This plugin adds two new parameter types to job configuration - **node** and **label**, this allows to dynamically select the node where a job/project should be executed.

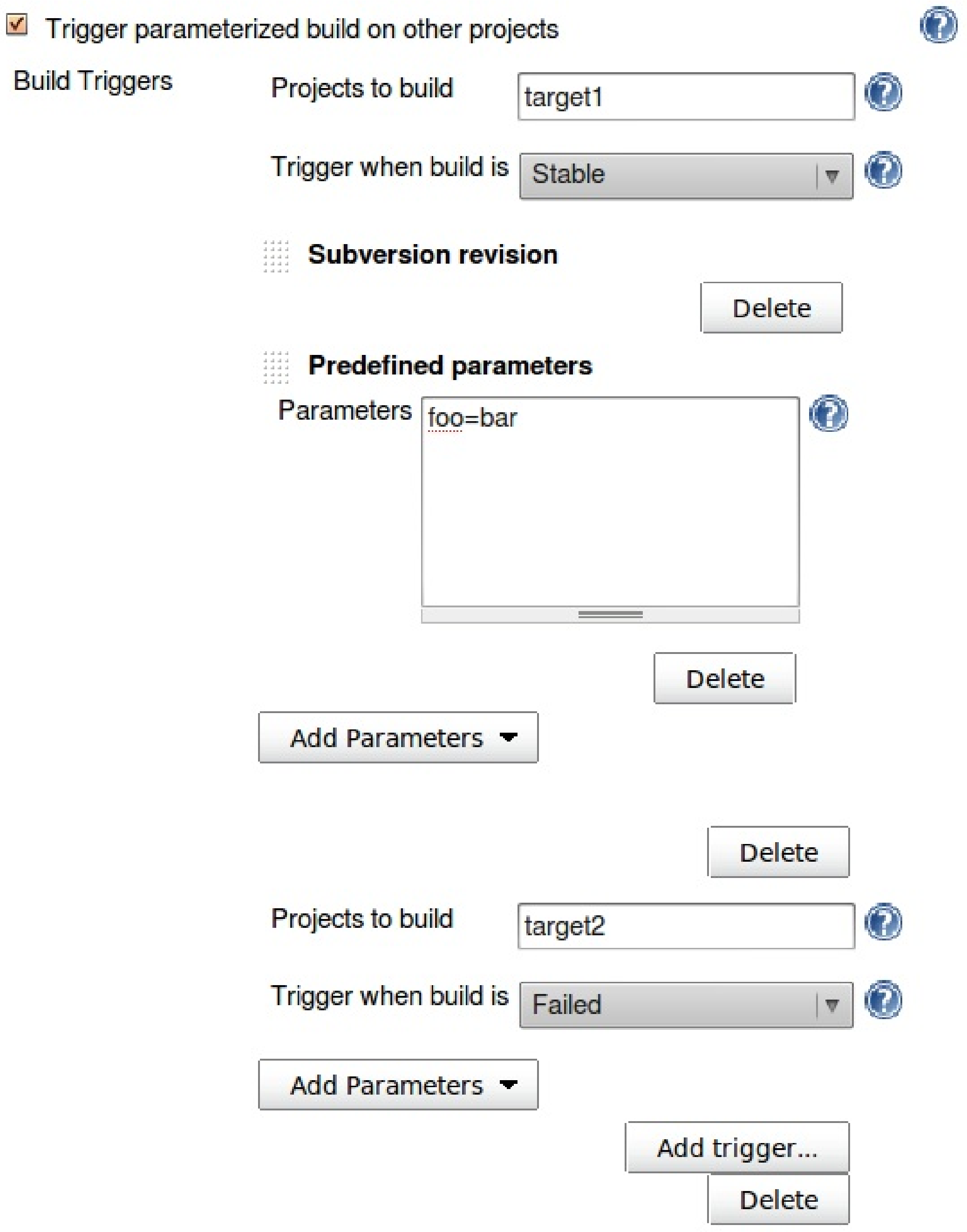
The plugin allows to configure additional parameters for a job. These new parameter types are 'Node' and 'Label'. This is specially useful if you want to execute the job on different nodes without changing the configuration over and over again. It also allows you to use Jenkins in a scenario where you would like to setup different nodes with the same script/jobs configured - e.g. SW provisioning. Another usage scenario would be to configure a node maintenance job which you could trigger on request on each node.



For (much) more info, check out the plugin's page

Parameterized Trigger Plugin

This plugin lets you trigger new builds when your build has completed, with various ways of specifying parameters for the new build.


Trigger parameterized build on other projects

Build Triggers

Projects to build: target1

Trigger when build is: Stable

Subversion revision

Predefined parameters

Parameters: foo=bar

Projects to build: target2

Trigger when build is: Failed

You can add multiple configurations: each has a list of projects to trigger, a condition for when to trigger them (based on the result of the current build), and a parameters section.

There is also a Parameterized Remote Trigger Plugin in case you want to trigger a build on a different/remote Jenkins Master.

The parameters section can contain a combination of one or more of the following:

- a set of predefined properties
- properties from a properties file read from the workspace of the triggering build
- the parameters of the current build
- Subversion revision: makes sure the triggered projects are built with the same revision(s) of the triggering build. You still have to make sure those projects are actually

configured to checkout the right Subversion URLs.
- Restrict matrix execution to a subset: allows you to specify the same combination filter expression as you use in the matrix project configuration and further restricts the subset of the downstream matrix builds to be run.

The parameter section is itself pluggable, and other plugins can contribute other sources of parameters.

This triggering mechanism can be used both as a post-build step or as a build step, in which case you can also block for the completion of the triggered builds. This lets you create a "function call" like semantics.

Usage as a Build step

When using the "Trigger/Call builds on another project" item. If the trigger is configured with the "Block until the triggered projects finish their builds" enabled, the following Environment variables are made available for further build steps

Env variables for future build steps

- `LAST_TRIGGERED_JOB_NAME` ="Last project started"
- `TRIGGERED_BUILD_NUMBER_<project name>` ="Last build number triggered" **from version 2.17 onwards**
- `TRIGGERED_JOB_NAMES` ="Comma separated list of all triggered projects"
- `TRIGGERED_BUILD_NUMBERS_<project name>` ="Comma separated list of build numbers triggered"
- `TRIGGERED_BUILD_RESULT_<project name>` ="Last triggered build result of project"
- `TRIGGERED_BUILD_RESULT_<project name>RUN<build number>` ="Result of triggered build for build number"
- `TRIGGERED_BUILD_RUN_COUNT_project name>` ="Number of builds triggered for the project"

From 2.17 onwards

> All Project names have characters not a-zA-Z or 0-9 replaced by *(multiple characters are condensed into a single*).

Note that with the BuildStep a variable can be used for the project name, I.E. `${projectName}` .

Please submit bugs and feature requests to the issue tracker and not (only) in the comments.

Use of the plugin in a Matrix job

Post build task

When using the trigger parameterized build as a post build task for a matrix job the triggering will be done. once when all of the different matrix configurations have completed. In this case some of the Environment variables may not be resolvable as passing them to downstream jobs will fail. You also cannot use a variable for the downstream project name. If this functionality is needed, the BuildStep must be used.

Environment variables that should be available are the default shell ones (/env-vars.html) and ones defined as Parameters. Variables added by the other plugins as a buildwrappers may not be available.

Build step

When using the trigger parameterized build as a buildstep it will be called for every different configuration, so if triggering another project with no parameters it will be done the same number of times as you have configurations, possible causing the triggered job to run more than once.

However this also allows you to trigger other jobs with parameters relating to the current configuration, i.e. triggering a build on the same node with the same JDK.

Plugins contributing additional parameter types to this plugin

- Git Plugin — This plugin allows use of Git as a build SCM. A recent Git runtime is required (1.7.9 minimum, 1.8.x recommended). Plugin is only tested on official git client. Use exotic installations at your own risk.
- NodeLabel Parameter Plugin — This plugin adds two new parameter types to job configuration - node and label, this allows to dynamically select the node where a job/project should be executed.

Pipeline Plugin (formerly known as Workflow)

See here for more information.

The default interaction model with Jenkins, historically, has been very web UI driven, requiring users to manually create jobs, then manually fill in the details through a web browser. This requires additional effort to create and manage jobs to test and build multiple projects, it also keeps the configuration of a job to build/test/deploy separate from the actual code being built/tested/deployed. This prevents users from applying their existing CI/CD best practices to the job configurations themselves.

Pipeline

With the introduction of the Pipeline plugin, users now can implement a project's entire build/test/deploy pipeline in a Jenkinsfile and store that alongside their code, treating their pipeline as another piece of code checked into source control.

The Pipeline plugin was inspired by the Build Flow plugin but aims to improve upon some concepts explored by Build Flow with features like:

- the ability to suspend/resume of executing jobs.
- checking the pipeline definition into source control (Jenkinsfile)
- support for extending the domain specific language with additional, organization specific steps, via the "global library" feature

Promoted Builds Plugin

This plugin allows you to distinguish good builds from bad builds by introducing the notion of 'promotion'. Put simply, a promoted build is a successful build that passed additional criteria (such as more comprehensive tests that are set up as downstream jobs.) The typical situation in which you use promotion is where you have multiple 'test' jobs hooked up as downstream jobs of a 'build' job. You'll then configure the build job so that the build gets promoted when all the test jobs passed successfully. This allows you to keep the build job run fast (so that developers get faster feedback when a build fails), and you can still distinguish builds that are good from builds that compiled but had runtime problems.

Another variation of this usage is to manually promote builds (based on instinct or something else that runs outside Jenkins.) Promoted builds will get a star in the build history view, and it can be then picked up by other teams, deployed to the staging area, etc., as those builds have passed additional quality criteria. In more complicated scenarios, one can set up multiple levels of promotions. This fits nicely in an environment where there are multiple stages of testings (for example, QA testing, acceptance testing, staging, and production.)

Promotion Action

When a build is promoted, you can have Jenkins perform some actions (such as running a shell script, triggering other jobs, etc. — or in Jenkins lingo, you can run build steps.) This is useful for example to copy the promoted build to somewhere else, deploy it to your QA server. You can also define it as a separate job and then have the promotion action trigger that job.

Do not rely on files in the workspace The promotion action uses the workspace of the job as the current directory (and as such the execution of the promotion action is mutually exclusive from any on-going builds of the job.) But by the time promotion runs, this workspace can contain files from builds that are totally unrelated from the build being promoted.

To access the artifacts, use the Copy Artifact Plugin and choose the permalink.

Usage

To use this plugin, look for the "Promote builds when..." checkbox, on the Job-configuration page. Define one or a series of promotion processes for the job.

Then, after the promotion processes have been added and another build is run, a 'Promotion Status' menu item will be added to the new build's menu options. Note that this means that builds run before this point cannot be promoted.

How might you use promoted builds in your environment? Here are a few use cases.

Artifact storage -- you may not want to push an artifact to your main artifact repository on each build. With build promotions, you can push only when an artifact meets certain criteria. For example, you might want to push it only after an integration test is run.

Manual Promotions - You can choose a group of people who can run a promotion manually. This gives a way of having a "sign off" within the build system. For example, a developer might validate a build and approve it for QA testing only when a work product is completed entirely. Then another promotion can be added for the QA hand off to production.

Aggregation of artifacts - If you have a software release that consists of several not directly related artifacts that are in separate jobs, you might want to aggregate all the artifacts of a proven quality to a distribution location. To do this, you can create a new job, adding a "Copy artifacts from another job" (available through Copy Artifact plugin") for each item you want to aggregate. To get a certain promotion, select "Use permalink" in the copy artifact step, then your promoted build should show up in the list of items to copy.

Notes

On Downstream Promotion Conditions

One of the possible criteria for promoting a build is "When the following downstream projects build successfully", which basically says if all the specified jobs successfully built (say build BD of job JD), the build in the upstream will be promoted (say build BU of job JU.)

This mechanism crucially relies on a "link" between BD and BU, for BU isn't always the last successful build. We say "BD qualifies BU" if there's this link, and the qualification is established by one of the following means:

1. If BD records fingerprints and one of the fingerprints match some files that are produced by BU (which is determined from the fingerprint records of BU), then BD qualifies BU. Intuitively speaking, this indicates that BD uses artifacts from BU, and thus BD helped verify BU's quality.
2. If BU triggers BD through a build trigger, then BD qualifies BU. This is somewhat weak and potentially incorrect, as there's no machine-readable guarantee that BD actually used anything from BU, but nonetheless this condition is considered as qualification for those who don't configure fingerprints.

Note that in the case #1 above, JU and JD doesn't necessarily have to have any triggering relationship. All it takes is for BD to use some fingerprinted artifacts from BU, and records those fingerprints in BD. It doesn't matter how those artifacts are retrieved either — it could be via Copy Artifact Plugin, it could be through a maven repository, etc. This also means that you can have a single test job (perhaps parameterized), that can promote a large number of different upstream jobs.

Note that after installing this plugin and configuring a promotion process, the option to promote the build will not be available for builds run before the promotion process was configured.

Available Environment Variables

The following environment variables are added for use in scripts, etc. These were retrieved from github here.

- `PROMOTED_URL` - URL of the job being promoted ex: http://jenkins/job/job_name_being_promoted/77/
- `PROMOTED_JOB_NAME` - Promoted job name ex: job_name_being_promoted
- `PROMOTED_NUMBER` - Build number of the promoted job ex: 77
- `PROMOTED_ID` - ID of the build being promoted NOTE: The format changed as a part of JENKINS-24380 in the Jenkins core. Since the original format was used ti retrieve timestamps, the new PROMOTED_TIMESTAMP field has been added in 2.25
- `PROMOTED_TIMESTAMP` - timestamp of the promotion. Format example: 2016-02-17T10:46:14Z
- `PROMOTED_USER_NAME` - the user who triggered the promotion

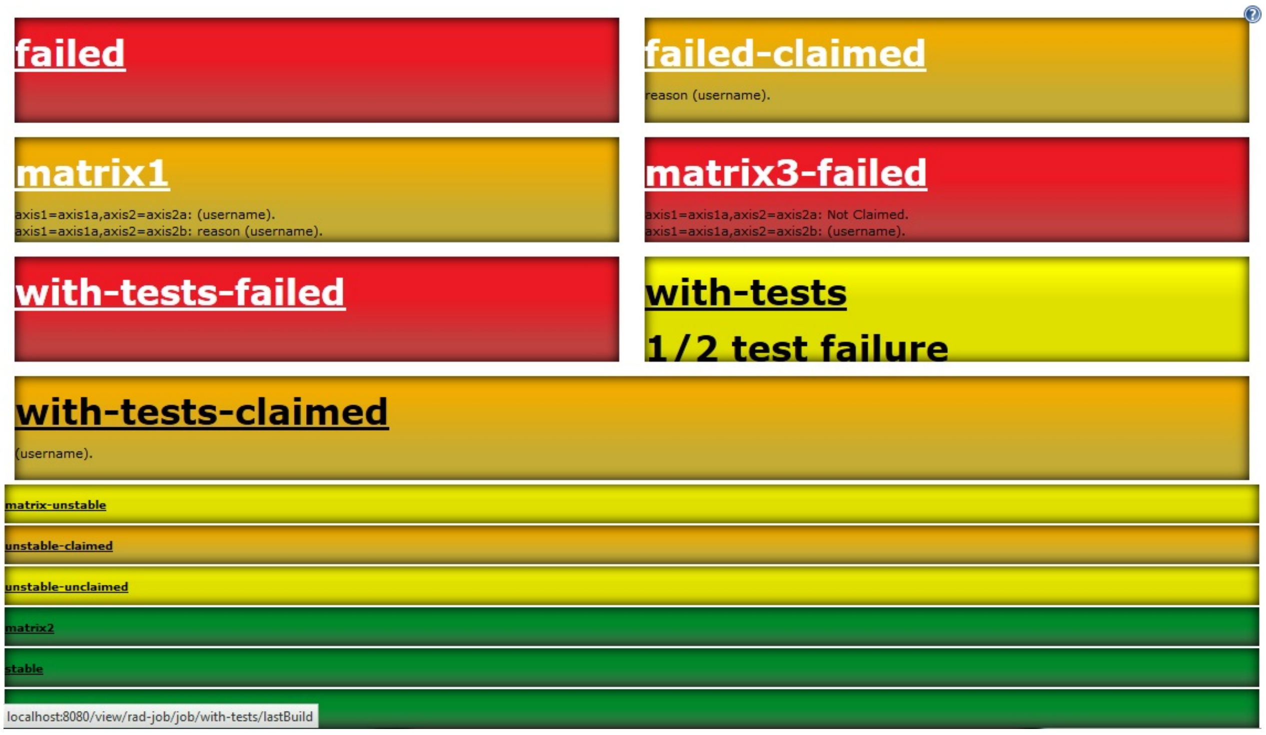
- `PROMOTED_USER_ID` - the user's id who triggered the promotion
- `PROMOTED_JOB_FULL_NAME` - the full name of the promoted job

Radiator View Plugin

Provides a job view displaying project status in a highly visible manner. This is ideal for displaying on a screen on the office wall as a form of Extreme Feedback Device.

Once the plugin is installed, click on the add view tab and select "Radiator View". The job selection options are the same as the standard list view -- either select projects to include or specify a regular expression to select the options. This plugin will be integrated with the claim plugin if it is installed - claimed failures are displayed in a column on the right.

Example


failed
failed-claimed
reason (username).
matrix1
axis1=axis1a,axis2=axis2a: (username).
axis1=axis1a,axis2=axis2b: reason (username).
matrix3-failed
axis1=axis1a,axis2=axis2a: Not Claimed.
axis1=axis1a,axis2=axis2b: (username).
with-tests-failed
with-tests
1/2 test failure
with-tests-claimed
(username).
matrix-unstable
unstable-claimed
unstable-unclaimed
matrix2
stable
localhost:8080/view/rad-job/job/with-tests/lastBuild

- Green boxes are shown when all jobs are successful.
- Red boxes are shown when any jobs fail (including test failures). Links to the failed builds and details of possible culprits if known are also shown.
- Amber boxes are shown when jobs fail, but all failing jobs are claimed. Details of the claims are also shown.

Hovering over the project name opens a list of all jobs relating to that project as in the top-left project. Hovering over the `?` button provides configuration options.

There are several different approaches to displaying the radiator, including non-project based and only showing failing builds - give it a try to see some of them.

SMS Notification Plugin

SMS Notification for failed Build, powered by Hoiio API (http://www.hoiio.com)

Script Security Plugin

Allows Jenkins administrators to control what in-process scripts can be run by less-privileged users.

User's guide

(adapted from information on template security in CloudBees Jenkins Enterprise)

Various Jenkins plugins require that users define custom scripts, most commonly in the Groovy language, to customize Jenkins's behavior. If everyone who writes these scripts is a Jenkins administrator—specifically if they have the Overall/RunScripts permission, used for example by the Script Console link—then they can write whatever scripts they like. These scripts may directly refer to internal Jenkins objects using the same API offered to plugins. Such users must be completely trusted, as they can do anything to Jenkins (even changing its security settings or running shell commands on the server).

However, if some script authors are "regular users" with only more limited permissions, such as Job/Configure, it is inappropriate to let them run arbitrary scripts. To support such a division of roles, the Script Security library plugin can be integrated into various feature plugins. It supports two related systems: script approval, and Groovy sandboxing.

Script Approval

The first, and simpler, security system is to allow any kind of script to be run, but only with an administrator's approval. There is a globally maintained list of approved scripts which are judged to not perform any malicious actions.

When an administrator saves some kind of configuration (for example, a job), any scripts it contains are automatically added to the approved list. They are ready to run with no further intervention. ("Saving" usually means from the web UI, but could also mean uploading a new XML configuration via REST or CLI.)

When a non-administrator saves a template configuration, a check is done whether any contained scripts have been edited from an approved text. (More precisely, whether the requested content has ever been approved before.) If it has not been approved, a request for approval of this script is added to a queue. (A warning is also displayed in the configuration screen UI when the current text of a script is not currently approved.)

An administrator may now go to Manage Jenkins » In-process Script Approval where a list of scripts pending approval will be shown. Assuming nothing dangerous-looking is being requested, just click Approve to let the script be run henceforth.

If you try to run an unapproved script, it will simply fail, typically with a message explaining that it is pending approval. You may retry once the script has been approved. The details of this behavior may vary according to the feature plugin integrating this library.

Groovy Sandboxing

Waiting for an administrator to approve every change to a script, no matter how seemingly trivial, could be unacceptable in a team spread across timezones or during tight deadlines. As an alternative option, the Script Security system lets Groovy scripts be run without approval so long as they limit themselves to operations considered inherently safe. This limited execution environment is called a sandbox. (Currently no sandbox implementations are available for other languages, so all such scripts must be approved if configured by non-administrators.)

To switch to this mode, simply check the box Use Groovy Sandbox below the Groovy script's entry field. Sandboxed scripts can be run immediately by anyone. (Even administrators, though the script is subject to the same restrictions regardless of who wrote it.) When the script is run, every method call, object construction, and field access is checked against a whitelist of approved operations. If an unapproved operation is attempted, the script is killed and the corresponding Jenkins feature cannot be used yet.

The Script Security plugin ships with a small default whitelist, and integrating plugins may add operations to that list (typically methods specific to that plugin).

But you are not limited to the default whitelist: every time a script fails before running an operation that is not yet whitelisted, that operation is automatically added to another approval queue. An administrator can go to the same page described above for approval of entire scripts, and see a list of pending operation approvals. If Approve is clicked next to the signature of an operation, it is immediately added to the whitelist and available for sandboxed scripts.

Most signatures be of the form method class.Name methodName arg1Type arg2Type…, indicating a Java method call with a specific “receiver” class (this), method name, and list of argument (or parameter) types. (The most general signature of an attempted method call will be offered for approval, even when the actual object it was to be called on was of a more specific type overriding that method.) You may also see staticMethod for static (class) methods, new for constructors, and field for field accesses (get or set).

Administrators in security-sensitive environments should carefully consider which operations to whitelist. Operations which change state of persisted objects (such as Jenkins jobs) should generally be denied. Most getSomething methods are harmless.

ACL-aware methods

Be aware however that even some “getter” methods are designed to check specific permissions (using an ACL: access control list), whereas scripts are often run by a system pseudo-user to whom all permissions are granted. So for example method hudson.model.AbstractItem getParent (which obtains the folder or Jenkins root containing a job) is in and of itself harmless, but the possible follow-up call method hudson.model.ItemGroup getItems (which lists jobs by name within a folder) checks Job/Read. This second call would be dangerous to whitelist unconditionally, since it would mean that a user who is granted Job/Create in a folder would be able to read at least some information from any jobs in that folder, even those which are supposed to be hidden according to a project-based authorization strategy; it would suffice to create a job in the folder which includes a Groovy script like this (details would vary according to the integrating plugin):

```
println("I sniffed ${thisjob.getParent().getItems()}!");
```

When run, the script output would display at least the names of supposedly secret projects. An administrator may instead click Approve assuming permission check for getItems; this will permit the call when run as an actual user (if the integrating plugin ever does so), while forbidding it when run as the system user (which is more typical). In this case, getItems is actually implemented to return only those jobs which the current user has access to, so if run in the former case (as a specific user), the description will show just those jobs they could see anyway. This more advanced button is shown only for method calls (and constructors), and should be used only where you know that Jenkins is doing a permission check.

Skype Plugin

Integrates Jenkins with Skype for instant messaging. Requires extra manual installation steps!!!

Note that you also need to install the instant-messaging plugin.

This plugin enables Jenkins to send build notifications via Skype, and to talk to Jenkins via a 'bot' to run commands, query build status etc..

Section 1: Key CI/CD/Jenkins concepts

This topic comprises approximately 27% of the exam. Questions cover the following topics:

Continuous Delivery/Continuous Integration Concepts

- Define continuous integration, continuous delivery, continuous deployment

Term	Definition
Continuous integration	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.
Continuous delivery	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.
Continuous deployment	Continuous deployment means that every change is automatically deployed to production. In order to do continuous deployment one must be doing continuous delivery.

- Difference between CI and CD

The main difference is that with CD, the software is ready for Release at any time. CD presupposes that CI is in effect

- Stages of CI and CD

CI: Commit, Integration, Build, Tests

CD: Build automation and continuous integration; test automation; and deployment automation

- Continuous delivery versus continuous deployment

With Continuous Delivery, the software is ready for deployment after each successfully integrated commit. With Continuous Deployment, the software gets released after each successfully integrated commit.

Jobs

- What are jobs in Jenkins?
- Types of jobs
- Scope of jobs

Builds

- What are builds in Jenkins?
- What are build steps, triggers, artifacts, and repositories?
- Build tools configuration

Source Code Management

- What are source code management systems and how are they used?
- Cloud-based SCMs
- Jenkins changelogs
- Incremental updates v clean check out
- Checking in code
- Infrastructure-as-Code
- Branch and Merge Strategies

Testing

- Benefits of testing with Jenkins
- Define unit test, smoke test, acceptance test, automated verification/functional tests

Notifications

- Types of notifications in Jenkins
- Importance of notifications

Distributed Builds

- What are distributed builds?
- Functions of masters and agents

Plugins

- What are plugins?
- What is the plugin manager?

Jenkins Rest API

- How to interact with it
- Why use it?

Security

- Authentication versus authorization
- Matrix security
- Definition of auditing, credentials, and other key security concepts

Fingerprints

- What are fingerprints?
- How do fingerprints work?

Artifacts

- How to use artifacts in Jenkins
- Storing artifacts

Configuration Management (Tools such as Chef, Puppet, etc.)

- Elements of software configuration management
- Importance of software configuration management

Using 3rd party tools

- How to use 3rd party tools with Jenkins

Various notes

Default shell environment valuables

see `<yourserver:port>/env-vars.html`

The following variables are available to shell scripts

- `BUILD_NUMBER` The current build number, such as "153"
- `BUILD_ID` The current build ID, identical to BUILD_NUMBER for builds created in 1.597+, but a YYYY-MM-DD_hh-mm-ss timestamp for older builds
- `BUILD_DISPLAY_NAME` The display name of the current build, which is something like "#153" by default.
- `JOB_NAME` Name of the project of this build, such as "foo" or "foo/bar". (To strip off folder paths from a Bourne shell script, try: ${JOB_NAME##*/})
- `BUILD_TAG` String of "jenkins-${JOB_NAME}-${BUILD_NUMBER}". Convenient to put into a resource file, a jar file, etc for easier identification.
- `EXECUTOR_NUMBER` The unique number that identifies the current executor (among executors of the same machine) that's carrying out this build. This is the number you see in the "build executor status", except that the number starts from 0, not 1.
- `NODE_NAME` Name of the slave if the build is on a slave, or "master" if run on master
- `NODE_LABELS` Whitespace-separated list of labels that the node is assigned.
- `WORKSPACE` The absolute path of the directory assigned to the build as a workspace.
- `JENKINS_HOME` The absolute path of the directory assigned on the master node for Jenkins to store data.
- `JENKINS_URL` Full URL of Jenkins, like http://server:port/jenkins/ (note: only available if Jenkins URL set in system configuration)
- `BUILD_URL` Full URL of this build, like http://server:port/jenkins/job/foo/15/ (Jenkins URL must be set)
- `JOB_URL` Full URL of this job, like http://server:port/jenkins/job/foo/ (Jenkins URL must be set)
- `SVN_REVISION` Subversion revision number that's currently checked out to the workspace, such as "12345"
- `SVN_URL` Subversion URL that's currently checked out to the workspace.

Section 1 Bibliography

These online resources provide entry points to understanding the above topics:

http://www.martinfowler.com

- Continuous Integration

  > [...] projects with Continuous Integration tend to have dramatically less bugs, both in production and in process. However I should stress that the degree of this benefit is directly tied to how good your test suite is. You should find that it's not too difficult to build a test suite that makes a noticeable difference. Usually, however, it takes a while before a team really gets to the low level of bugs that they have the potential to reach. Getting there means constantly working on and improving your tests. If you have continuous integration, it removes one of the biggest barriers to frequent deployment. Frequent deployment is valuable because it allows your users to get new features more rapidly, to give more rapid feedback on those features, and generally become more collaborative in the development cycle. This helps break down the barriers between customers and development - barriers which I believe are the biggest barriers to successful software development.

- Continuous Delivery

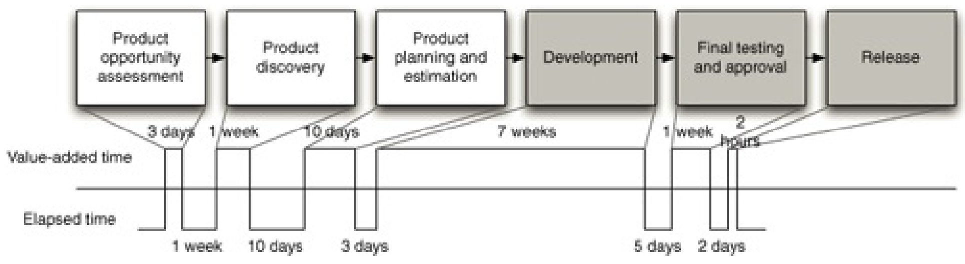
  > You achieve continuous delivery by continuously integrating the software done by the development team, building executables, and running automated tests on those executables to detect problems. Furthermore you push the executables into increasingly production-like environments to ensure the software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

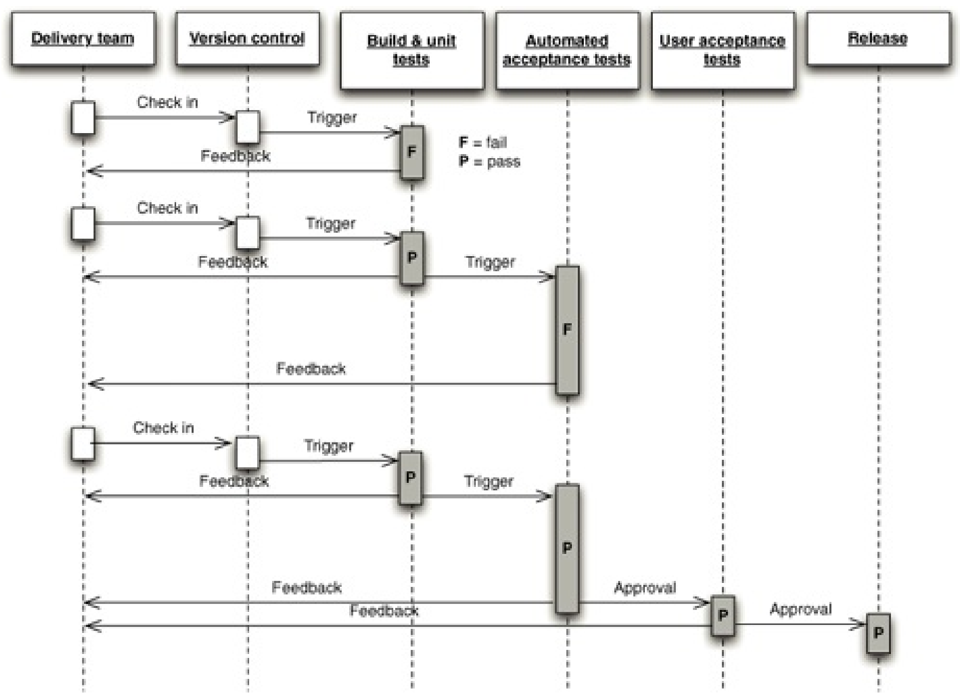
- Deployment Pipeline

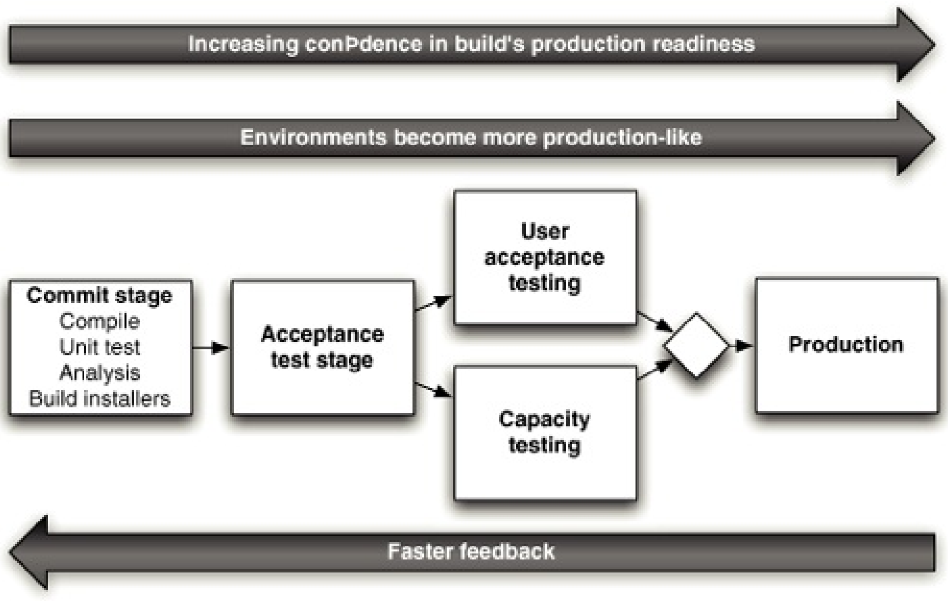
  > A deployment pipeline is a way to deal with this by breaking up your build into stages. Each stage provides increasing confidence, usually at the cost of extra time. Early stages can find most problems yielding faster feedback, while later stages provide slower and more through probing. Deployment pipelines are a central part of Continuous Delivery.

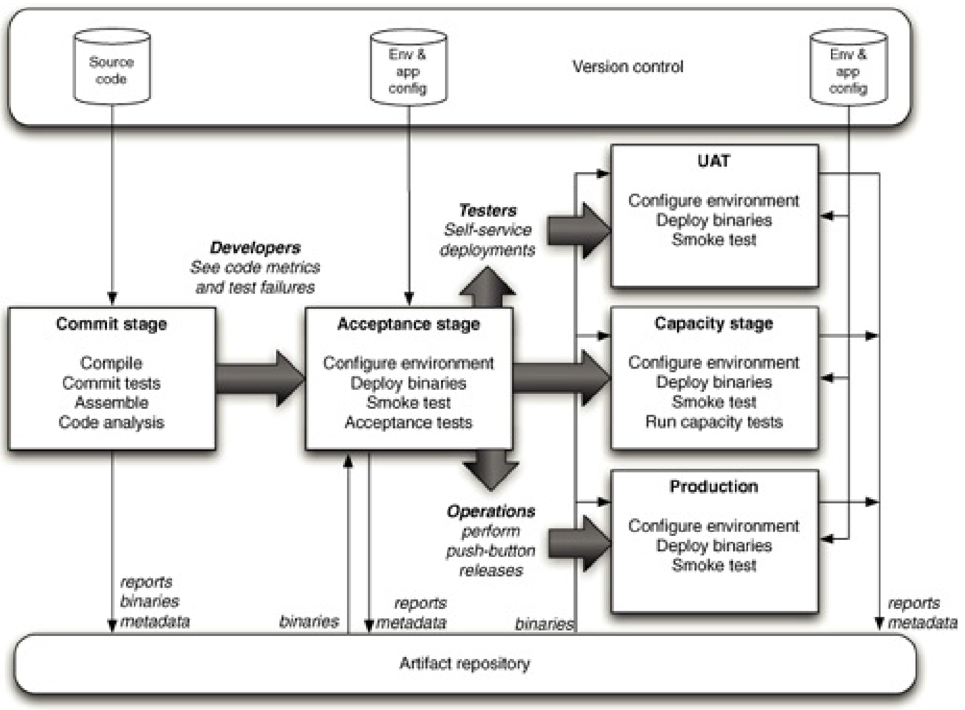
http://www.informit.com

- CD Pipeline Anatomy







> We must also automate deployment to testing, staging, and production environments to remove these manually intensive, error-prone steps. For many systems, other forms of testing and so other stages in the release process are also needed, but the subset that is common to all projects is as follows.
>
> - **The commit stage** asserts that the system works at the technical level. It compiles, passes a suite of (primarily unit-level) automated tests, and runs code analysis.
> - **Automated acceptance test stages** assert that the system works at the functional and nonfunctional level, that behaviorally it meets the needs of its users and the specifications of the customer.
> - **Manual test stages** assert that the system is usable and fulfills its requirements, detect any defects not caught by automated tests, and verify that it provides value to its users. These stages might typically include *exploratory testing environments, integration environments, and UAT (user acceptance testing)*.
> - **Release stage** delivers the system to users, either as packaged software or by deploying it into a production or staging environment (a staging environment is a testing environment identical to the production environment).
>
> We refer to these stages, and any additional ones that may be required to model your process for delivering software, as a deployment pipeline.
>
> This is not intended to imply that there is no human interaction with the system through this release process; rather, it ensures that error-prone and complex steps are automated, reliable, and repeatable in execution.
>
> 
>

http://devops.com

- What is a CD pipeline

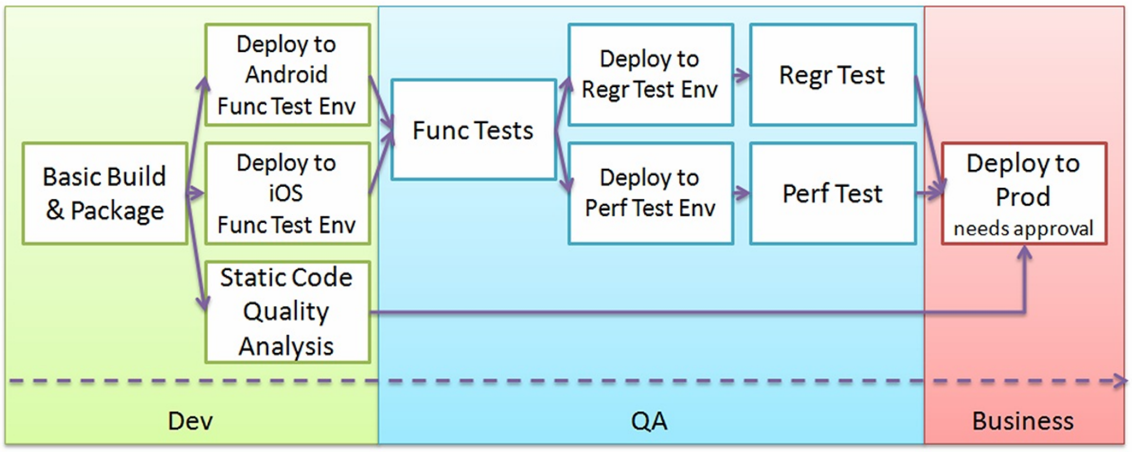
> A typical CD pipeline will include the following stages: build automation and continuous integration; test automation; and deployment automation. The deployment pipeline is supported by platform provisioning and system configuration management, which allow teams to create, maintain and tear down complete environments automatically or at the push of a button.

https://jaxenter.com

- Implementing Continuous Delivery
  > A key aspect of continuous delivery is the ability to automate your configuration. This configuration-as-code DevOps practice ensures consistency in your CD process, clearing away problems that could result from rebuilding your configuration – potentially inconsistently – every time you want to push a release to production.
  >
  > - Define a process
  > - Ensure a blameless culture
  > - Set metrics and measure your success
  > - Adopt configuration as code (Ansible+Docker?)
  > - Orchestrating a process

http://www.infoq.com/articles/

- Orchestrating Pipelines Jenkins


Basic Build & Package
Deploy to Android Func Test Env
Deploy to iOS Func Test Env
Static Code Quality Analysis
Func Tests
Deploy to Regr Test Env
Regr Test
Deploy to Perf Test Env
Perf Test
Deploy to Prod needs approval
Dev
QA
Business

> Setting up Continuous Delivery pipelines in Jenkins that are secure, efficient, and easy to use and manage can quickly become challenging. We've discussed important prerequisites, made a number of recommendations and introduced a set of freely available plugins which can make the process a whole lot easier. Hopefully, you are now in a better position to identify whether Jenkins is the right orchestrator for your current process, to build pipelines painlessly and to quickly start making life better for your teams and delivering business value to your customers!

http://technologyconversations.com

- Continuous Delivery Introduction to Concepts and Tools
  > 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.

https://en.wikipedia.org

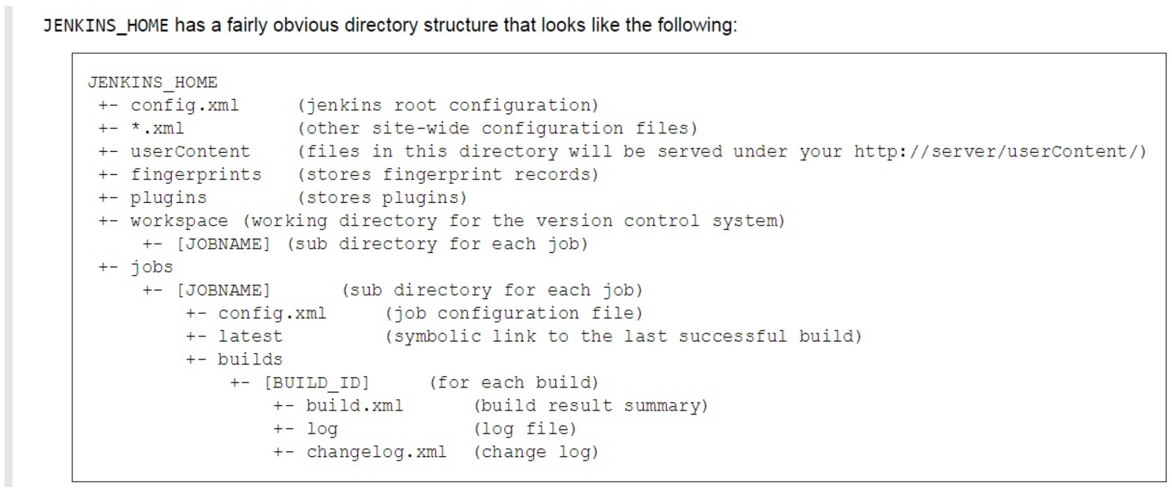
- Continuous delivery
- Artifact software development
- Build automation
- Distributed version control
- List of version control software
- Smoke testing (software)

https://www.safaribooksonline.com

- Jenkins the Definitive Guide

https://wiki.jenkins-ci.org

- Administering Jenkins



`JENKINS_HOME` has a fairly obvious directory structure that looks like the following:

```
JENKINS_HOME
 +- config.xml     (jenkins root configuration)
 +- *.xml          (other site-wide configuration files)
 +- userContent    (files in this directory will be served under your http://server/userContent/)
 +- fingerprints   (stores fingerprint records)
 +- plugins        (stores plugins)
 +- workspace (working directory for the version control system)
     +- [JOBNAME] (sub directory for each job)
 +- jobs
     +- [JOBNAME]      (sub directory for each job)
         +- config.xml     (job configuration file)
         +- latest         (symbolic link to the last successful build)
         +- builds
             +- [BUILD_ID]     (for each build)
                 +- build.xml      (build result summary)
                 +- log            (log file)
                 +- changelog.xml  (change log)
```

- Terminology

Contains a table of terms used in Jenkins..

- Extreme feedback lamp switch gear style

Just a goofy page about communicating build status with lamps

- Distributed builds: Offline status and retention strategy Useful page about setting up slave nodes

I set up a test slave node in one of our Test VMs

- Remoting issue
- Remote access API
- Matrix based security
- Securing Jenkins
- Quick and Simple Security

http://docs.openstack.org

- Jenkins job builder

https://www.simple-talk.com

- Branching and merging

http://stackoverflow.com

- What is unit test ,integration test, smoke test, regression test?

https://www.cloudbees.com

- Notifications

http://searchsecurity.techtarget.com/

- Authentication authorization and accounting