JENKINS COOKBOOK

Cutting-Edge Best Practices For Continuous Delivery

CloudBees®

The Enterprise Jenkins Company

Chapter 1. Hardware Recommendations

1.1. Introduction

Sizing a Jenkins environment depends on a number of factors, which makes it a very inexact science. Achieving an optimal configuration requires some experience and experimentation, but it is possible to make a smart approximation to start, especially when designing with Jenkins’ best practices in mind.

The following chapter will outline these factors and how you can account for them when sizing your configuration. It will also share some sample configurations and the hardwares behind some of the largest Jenkins installations presented in a Jenkins Scalability Summit.

1.2. Choosing the right hardware for masters

One of the greatest challenges of properly setting up a Jenkins instance is the fact that there is no "one size fits all" answer - the exact specifications of the hardware that you will need will depend heavily on your organization’s current and future needs.

Your Jenkins master will be serving HTTP requests and storing all of the important information for your Jenkins instance in its *$JENKINS_HOME* folder (configurations, build histories, plugins). Please note that for masters which will run in a highly available configuration using the CloudBees High Availability plugin, the *$JENKINS_HOME* will need to reside in a network-attached storage server, which then must be made accessible to the Jenkins servers using NFS or some other protocol.

More information on sizing masters based organizational needs can be found in Architecting for Scale[1].

1.2.1. Memory requirements for the master

The amount of memory Jenkins needs is largely dependent on many factors, which is why the RAM allotted for it can range from 200 MB for a small installation to 70+ GB for a single and massive Jenkins master. However, you should be able to estimate the RAM required based on your project build needs.

Each build node connection will take 2-3 threads, which equals about 2 MB or more of memory. You will also need to factor in CPU overhead for Jenkins if there are a lot of users who will be accessing the Jenkins user interface.

It is generally a bad practice to allocate executors on a master, as builds can quickly overload a master’s CPU/memory/etc and crash the instance, causing unnecessary

[1] http://jenkins-cookbook.cloudbees.com/docs/jenkins-cookbook/_right_sizing_jenkins_masters.html#_calculating_how_many_jobs_masters_and_executors_are_needed

downtime. Instead, it is advisable to set up slaves that the Jenkins master can delegate build jobs to, keeping the bulk of the work off of the master itself.

1.3. Choosing the right slave machines

The backbone of Jenkins is its ability to orchestrate builds, but installation which do not leverage Jenkins' distributed builds architecture are artificially limiting the number of builds that their masters can orchestrate. More information on the benefits of a distributed architecture[2] can be found in the Architecting for Scale section.

1.3.1. Requirements for a machine to be a slave

Slave machines are typically generic x86 machines with enough memory to run specific build types. The slave machine's configuration depends on the builds it will be used for and on the tools required by the same builds.

Configuring a machine to act as slave inside your infrastructure can be tedious and time consuming. This is especially true when the same set-up has to be replicated on a large pool of slaves. Because of this, is ideal to have fungible slaves, which are slaves that are easily replaceable. Slaves should be generic for all builds rather customized for a specific job or a set of jobs. The more generic the slaves, the more easily they are interchanged, which in turn allows for a better use of resources and a reduced impact on productivity if some slaves suffer an outage. Andrew Bayer introduced this concept of "fungibility" as applied to slaves during his presentation "Seven Habits of Highly Effective Jenkins Users" at the Jenkins User Conference (Europe, 2014)[3].

The more automated the environment configuration is, the easier it is to replicate a configuration onto a new slave machine. Tools for configuration management or a pre-baked image can be excellent solutions to this end. Containers and virtualization are also popular tools for creating generic slave environments.

More information on estimating the number of executors needed in a given environment can be found in the Architecting for Scale[4] section.

1.4. Sample use cases published on-line

For help with regards to architecture and best practices, it is always a good idea to refer to real-life examples to learn from others' experiences. All architecture decisions require some trade off, but you can save your organization time and money by anticipating and planning for these pains before you experience them.

[2] http://jenkins-cookbook.cloudbees.com/docs/jenkins-cookbook/_architecting_for_scale.html#_distributed_builds_architecture
[3] http://www.slideshare.net/andrewbayer/seven-habits-of-highly-effective-jenkins-users-2014-edition
[4] http://jenkins-cookbook.cloudbees.com/docs/jenkins-cookbook/_architecting_for_scale.html

In the Jenkins community, such examples can be found by presenters at the Jenkins User Conferences, scalability summits, and other such community events.

Here are a few case studies from the 2013 Jenkins Scalability Summit:

1.4.1. Netflix

At the Scalability Summit, Gareth Bowles, a Sr. Build/Tool engineer at Netflix, explained how Netflix created its large installation and discussed some of the pain they encountered.

In 2012, Netflix had the following configuration:

- 1 master
  - 700 engineers on one master
- Elastic slaves with Amazon EC2 + 40 ad-hoc slaves in Netflix's data center
- 1,600 jobs
  - 2,000 builds per day
  - 2 TB of builds
  - 15% build failures
- Hardware Configuration
  - 2x quad core x86_64 for the master
  - 26G memory
  - m1.xlarge slave machines on AWS

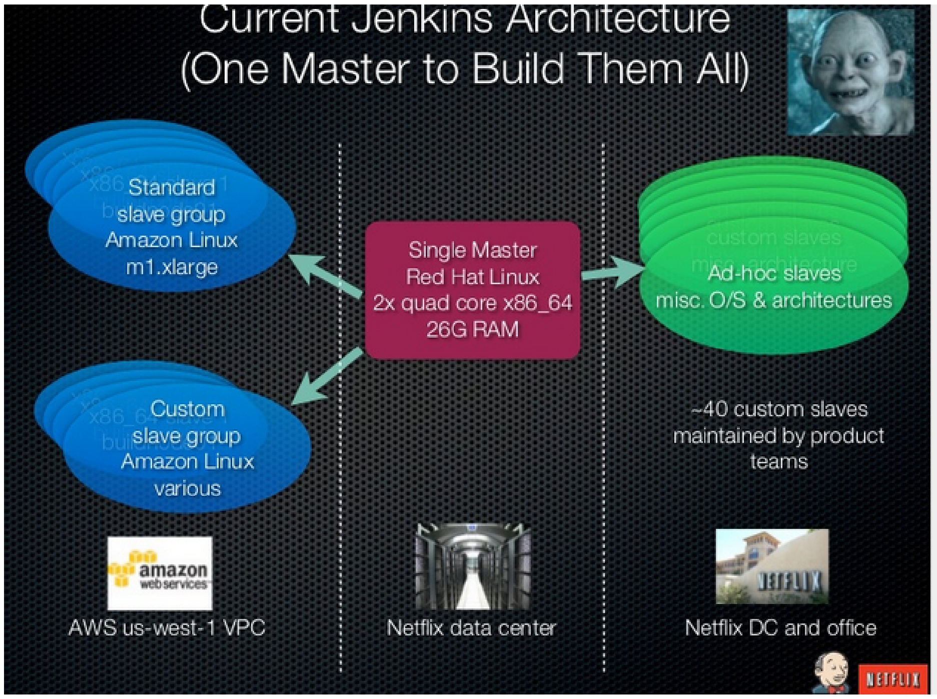

Current Jenkins Architecture
(One Master to Build Them All)
Standard slave group Amazon Linux m1.xlarge
Custom slave group Amazon Linux various
Single Master Red Hat Linux 2x quad core x86_64 26G RAM
Ad-hoc slaves misc. O/S & architectures
~40 custom slaves maintained by product teams
AWS us-west-1 VPC
Netflix data center
Netflix DC and office

Figure 1.1. Netflix's physical architecture

By 2013, Netflix split their monolithic master and significantly increased their number of jobs and builds:

- 6 masters
  - 700 engineers on each master
- 100 slaves
  - 250 executors, but generally 4 executors per slave
- 3,200 jobs
  - 3 TB of builds
  - 15% build failures

Scheduling and monitoring

Netflix was able to spin up dynamic slaves using the Dynaslave plugin[5], and configure Jenkins to automatically disable any jobs that hadn't run successfully in 30 days. They were also able to monitor their slaves' uptime using a Groovy script in a Jenkins job and which emailed the slaves' teams if there was downtime. Furthermore, this script

[5] http://www.slideshare.net/bmoyles/the-dynaslave-plugin

would also monitor slave connections for errors and would remove the offending slaves. Removed slaves were then de-provisioned.

1.4.2. Yahoo

Mujibur Wahab of Yahoo also presented Yahoo’s massive installation to the 2013 Scalability Summit. Their installation was:

- 1 primary master
  - 1,000 engineers rely on this Jenkins instance
  - 3 backup masters
  - *$JENKINS_HOME* lives on NetApp
- 50 Jenkins slaves in 3 data centers
  - 400+ executors
- 13,000 jobs
  - 8,000 builds/day
  - 20% build failure rate
  - 2 million builds/year and on target for 1 million/quarter
- Hardware Configuration
  - 2 x Xeon E5645 2.40GHz, 4.80GT QPI (HT enabled, 12 cores, 24 threads)
  - 96G memory
  - 1.2TB disk
  - 48GB max heap to JVM
  - 20TB Filer volume to store Jenkins job and build data
  - This volume stores 6TB of build data

Here is an overview of their architecture, as taken from Wahab’s slides:

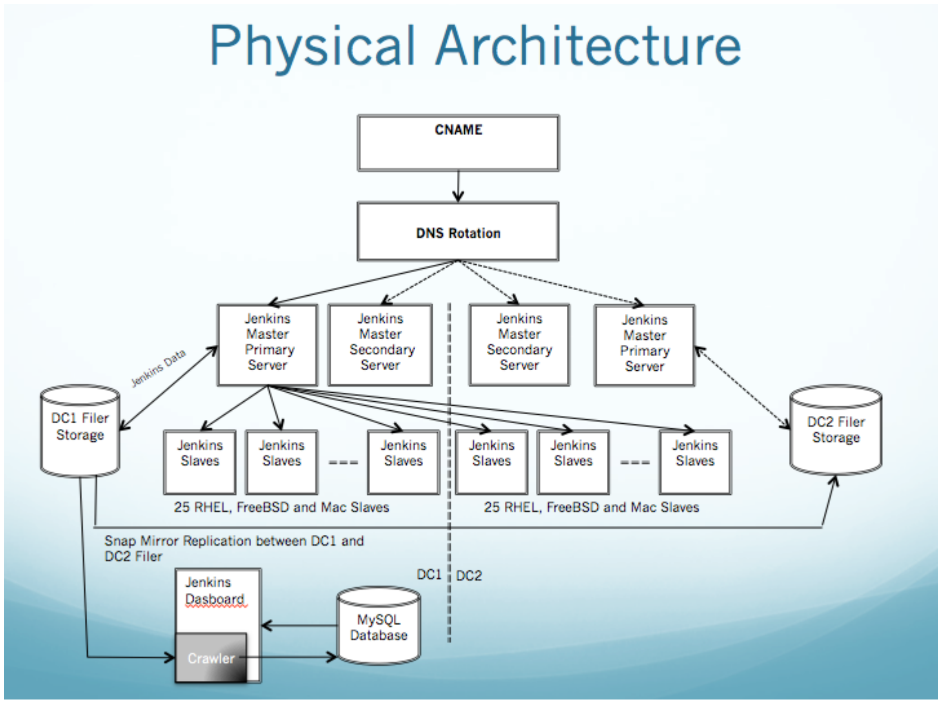

Physical Architecture
CNAME
DNS Rotation
Jenkins Master Primary Server
Jenkins Master Secondary Server
Jenkins Master Secondary Server
Jenkins Master Primary Server
Jenkins Data
DC1 Filer Storage
DC2 Filer Storage
Jenkins Slaves
Jenkins Slaves
Jenkins Slaves
Jenkins Slaves
Jenkins Slaves
Jenkins Slaves
25 RHEL, FreeBSD and Mac Slaves
25 RHEL, FreeBSD and Mac Slaves
Snap Mirror Replication between DC1 and DC2 Filer
DC1
DC2
Jenkins Dasboard
Crawler
MySQL Database

Figure 1.2. Yahoo’s physical architecture

Because continuous delivery is so critical to Yahoo, they created a Jenkins team to develop tools related to their pipeline and provide Jenkins-as-a-service to the internal Yahoo teams. The Jenkins team is not responsible for job configurations or creating the pipelines, just the uptime of the infrastructure. The health of their infrastructure is monitored by other existing mechanisms.

Yahoo quickly found that running only one build per slave was a problem because it would be impossible to continue adding new hardware to scale with their increasing build needs. To solve this, they started using an LXC-like chroot scheme to emulate virtualization. This light-weight container is a heavily augmented version of the standard UNIX command “chroot”. Their version installs all files need to create a functional, clean software environment and provides the ability to manage those virtual environments. Each VM gets 2 threads and as well as 4GB of memory to accommodate their Maven builds.

Chapter 2. Architecting for Scale

2.1. Introduction

As an organization matures from a continuous delivery standpoint, its Jenkins requirements will similarly grow. This growth is often reflected in the Jenkins master's architecture, whether that be "vertical" or "horizontal" growth.

Vertical growth is when a master's load is increased by having more configured jobs or orchestrating more frequent builds. This may also mean that more teams are depending on that one master.

Horizontal growth is the creation of additional masters within an organization to accommodate new teams or projects, rather than adding these things to an existing single master.

There are potential pitfalls associated with each approach to scaling Jenkins, but with careful planning, many of them can be avoided or managed. Here are some things to consider when choosing a strategy for scaling your organization's Jenkins instances:

- **Do you have the resources to run a distributed build system?** If possible, it is recommended set up dedicated build nodes that run separately from the Jenkins master. This frees up resources for the master to improve its scheduling performance and prevents builds from being able to modify any potentially sensitive data in the master's *$JENKINS_HOME*. This also allows for a single master to scale far more vertically than if that master were both the job builder and scheduler.

- **Do you have the resources to maintain multiple masters?** Jenkins masters will require regular plugin updates, semi-monthly core upgrades, and regular backups of configurations and build histories. Security settings and roles will have to be manually configured for each master. Downed masters will require manual restart of the Jenkins master and any jobs that were killed by the outage.

- **How mission critical are each team's projects?** Consider segregating the most vital projects to separate masters to minimize the impact of a single downed master. Also consider converting any mission-critical project pipelines to Workflow jobs, which have the ability to survive a master-slave connection interruptions.

- **How important is a fast start-up time for your Jenkins instance?** The more jobs a master has configured, the longer it takes to load Jenkins after an upgrade or a crash. The use of folders and views to organize jobs can limit the number of that need to be rendered on start up.

2.2. Distributed Builds Architecture

A Jenkins master can operate by itself both managing the build environment and executing the builds with its own executors and resources. If you stick with this “standalone” configuration you will most likely run out of resources when the number or the load of your projects increase.

To come back up and running with your Jenkins infrastructure you will need to enhance the master (increasing memory, number of CPUs, etc). The time it takes to maintain and upgrade the machine, the master together with all the build environment will be down, the jobs will be stopped and the whole Jenkins infrastructure will be unusable.

Scaling Jenkins in such a scenario would be extremely painful and would introduce many “idle” periods where all the resources assigned to your build environment are useless.

Moreover, executing jobs on the master’s executors introduces a “security” issue: the ”jenkins” user that Jenkins uses to run the jobs would have full permissions on all Jenkins resources on the master. This means that, with a simple script, a malicious user can have direct access to private information whose integrity and privacy could not be, thus, guaranteed.

For all these reasons Jenkins supports the "master/slave" mode, where the workload of building projects are delegated to multiple slave nodes.

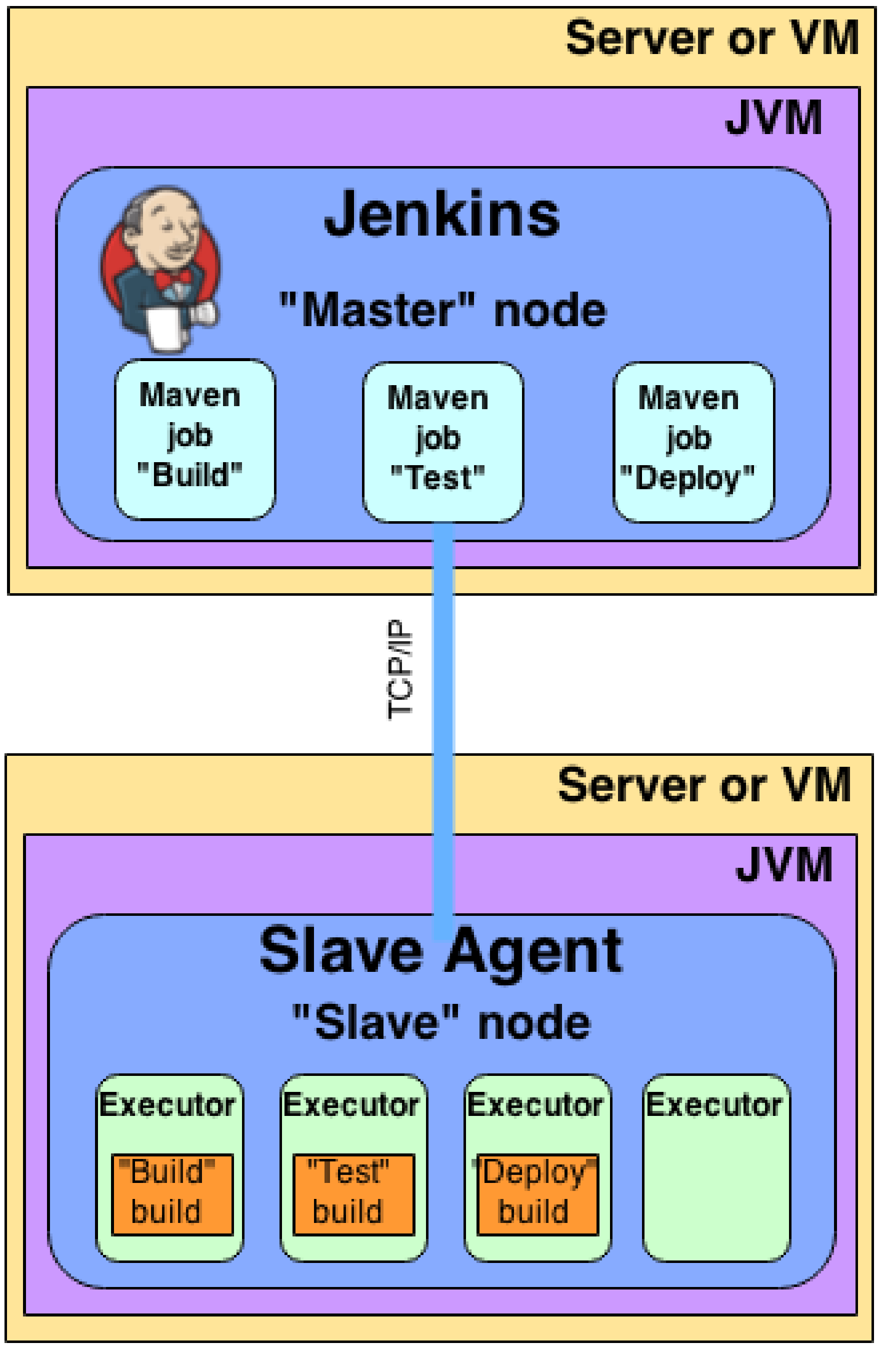


Figure 2.1. Master slave architecture

Note

An architecture with 1 Jenkins master connected to 1 slave over a TCP/IP socket. Slave has 4 execution slots and is building 3 of the master's configured jobs.

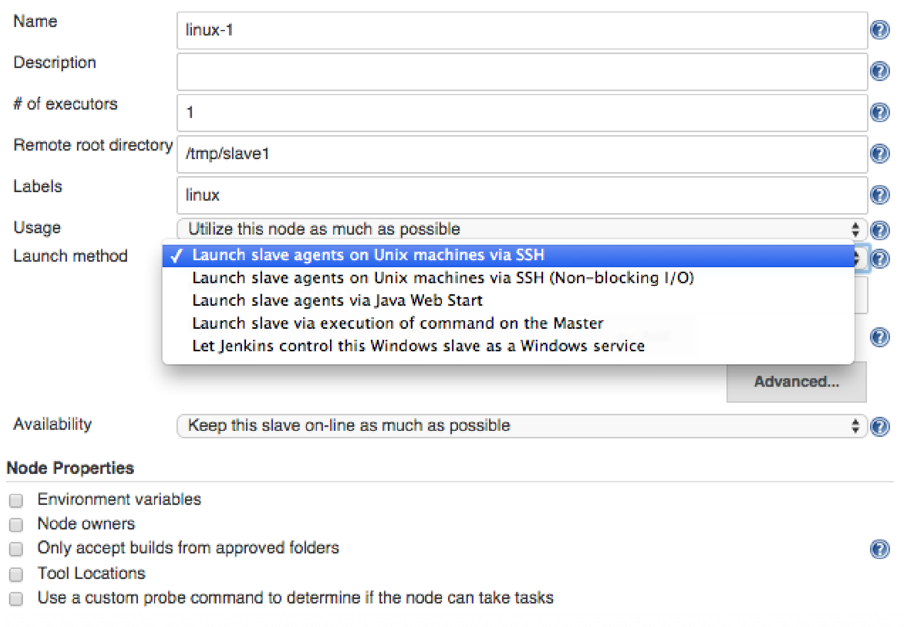
A slave is a machine set up to offload build projects from the master. The method for scheduling builds depends on the configuration given to a specific project: some projects may be configured to only use a specific slave while others can freely pick up slaves among the pool of slaves assigned to the master.

In a distributed builds environment, the Jenkins master will use its resources to only handle HTTP requests and manage the build environment. Actual execution of builds will be delegated to the slaves. With this configuration it is possible to horizontally scale an architecture, which allows a single Jenkins installation to host a large number of projects and build environments.

2.2.1. Master/slave communication protocols

In order for a machine to be recognized a slave, it needs to run a specific agent program that establish a bi-directional communication with the master.

There are different ways to establish a connection between master and slave:



Name	linux-1
Description	
# of executors	1
Remote root directory	/tmp/slave1
Labels	linux
Usage	Utilize this node as much as possible
Launch method	✓ Launch slave agents on Unix machines via SSH
	Launch slave agents on Unix machines via SSH (Non-blocking I/O)
	Launch slave agents via Java Web Start
	Launch slave via execution of command on the Master
	Let Jenkins control this Windows slave as a Windows service
Availability	Keep this slave on-line as much as possible

Node Properties

- ☐ Environment variables
- ☐ Node owners
- ☐ Only accept builds from approved folders
- ☐ Tool Locations
- ☐ Use a custom probe command to determine if the node can take tasks

Figure 2.2. New node configuration screen

- **The SSH connector**: Configuring a slave to use the SSH connector is the preferred and the most stable way to establish master-slave communication. Jenkins has a built-in SSH client implementation. This means that the Jenkins master can easily communicate with any machine with a SSHD server installed. The only requirements is that the public key of the master is part of the set of the authorized-keys of the slaves. Once defined the host and the ssh key to be used (in the Manage Node # Add New Node configuration page), Jenkins will manage the communication automatically by installing the binaries of the agent program and starting/stopping the slave program as needed.

- **The JNLP-TCP connector**: In this case the communication is established starting the slave agent through Java Web Start (JNLP). With this connector the

Java Web Start program has to be launched in the slave machine in 2 different ways:

- Manually by navigating to the Jenkins master URL in the slave's browser. Once the Java Web Start icon is clicked, the slave agent will be launched on the slave machine. The downside of this approach is that the slaves cannot be centrally managed by the Jenkins master and each/stop/start/update of the slave agent needs to be executed manually on the slave machine in versions of Jenkins older than 1.611. This approach is convenient when the master cannot instantiate the connection with the client (i.e. it runs outside a firewall whereas the slave runs inside).

- As a service, which allows for restarts to be automated. First you'll need to manually launch the slave using the above method. After manually launching it, *jenkins-slave.exe* and *jenkins-slave.xml* will be created in the slave's work directory. Now go to the command line to execute the following command:

```
sc.exe create "<serviceKey>" start= auto binPath= "<path to jenkins-slave.exe>" Dis
```

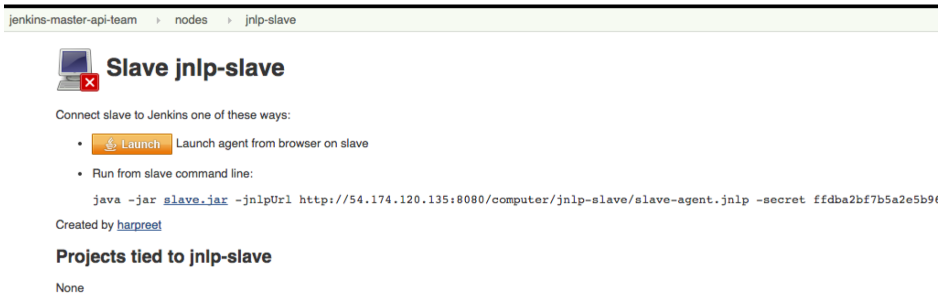
<serviceKey> is the name of the registry key to define this slave service and <service display name> is the label that will identify the service in the Service Manager interface.

To ensure that restarts are automated, you will need to download a slave jar newer than v 2.37 and copy it to a permanent location on the slave machine. The *slave.jar* can be found in:

```
http://<your-jenkins-host>/jnlpJars/slave.jar
```

If running a version of Jenkins newer than 1.559, the *slave.jar* will be kept up to date each time it connects to the master.

- **The JNLP-HTTP connector**: This approach is quite similar to the JNLP-TCP Java Web Start approach, with the difference in this case being that the slave agent is executed as headless and the connection can be tunneled via HTTP(s). The exact command can be found on your JNLP slave's configuration page:



Slave jnlp-slave

Connect slave to Jenkins one of these ways:

- Launch agent from browser on slave
- Run from slave command line:

```
java -jar slave.jar -jnlpUrl http://54.174.120.135:8080/computer/jnlp-slave/slave-agent.jnlp -secret ffdba2bf7b5a2e5b9
```

Created by harpreet

Projects tied to jnlp-slave

None

Figure 2.3. JNLP slave launch command

This approach is convenient for an execution as a daemon on Unix.

- **Custom-script**: It is also possible to create a custom script to initialize the communication between master and slave if the other solutions do not provide enough flexibility for a specific use-case. The only requirement is that the script runs the java program as a *java -jar slave.jar* on the slave.

Windows slaves set-up can either follow the standard SSH and JNLP approach or use a more Windows-specific configuration approach. Windows slaves have the following options:

- **SSH-connector approach with Putty**
- **SSH-connector approach with Cygwin and OpenSSH**: This[1] is the easiest to setup and recommended approach.
- **Remote management facilities (WMI + DCOM)**: With this approach, which utilizes the Windows Slave plugin[2]), the Jenkins master will register the slave agent on the windows slave machine creating a Windows service. The Jenkins master can control the slaves, issuing stops/restarts/updates of the same. However this is difficult to set-up and not recommended.
- **JNLP-connector approach**: With this approach[3] it is possible to manually register the slave as Windows service, but it will not be possible to centrally manage it from the master. Each stop/start/update of the slave agent needs to be executed manually on the slave machine, unless running Jenkins 1.611 or newer.

2.3. Creating fungible slaves

2.3.1. Configuring tools location on slaves

The Jenkins Global configuration page let you specify the tools needed during the builds (i.e. Ant, Maven, Java, etc).

When defining a tool, it is possible to create a pointer to an existing installation by giving the directory where the program is expected to be on the slave. Another option is to let Jenkins take care of the installation of a specific version in the given location. It is also possible to specify more than one installation for the same tool since different jobs may need different versions of the same tool.

The pre-compiled “Default” option calls whatever is already installed on the slave and exists in the machine PATH, but this will return a failure if the tool was not already installed and its location was not added to the PATH system variable.

[1] http://wiki.jenkins-ci.org/display/JENKINS/SSH+slaves+and+Cygwin
[2] http://wiki.jenkins-ci.org/display/JENKINS/Windows+Slaves+Plugin
[3] http://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+as+a+Windows+service

One best practice to avoid this failure is to configure a job with the assumption that the target slave does not have the necessary tools installed, and to include the tools' installation as part of the build process.

2.3.2. Define a policy to share slave machines

As mentioned previously, slaves should be interchangeable and standardized in order to make them sharable and to optimize resource usage. Slaves should not be customized for a particular set of jobs, nor for a particular team. CloudBees' recommendation is always to make slaves general enough to be shared among jobs and teams, but there are exceptions.

Lately Jenkins has become more and more popular not only in CI but also in CD, which means that it must orchestrate jobs and pipelines which involve different teams and technical profiles: developers, QA people and Dev-Ops people.

In such a scenario, it might make sense to create customized and dedicated slaves: different tools are usually required by different teams (i.e. Puppet/Chef for the Ops team) and teams' credentials are usually stored on the slave in order to ensure their protection and privacy.

In order to ensure the execution of a job on a single/group of slaves only (i.e. iOS builds on OSX slaves only), it is possible to tie the job to the slave by specifying the slave's label in the job configuration page. Note that the restriction has to be replicated in every single job to be tied and that the slave won't be protected from being used by other teams.

2.3.3. Setting up cloud slaves

Cloud build resources can be a solution for a case when it is necessary to maintain a reasonably small cluster of slaves on-premise while still providing new build resources when needed.

In particular it is possible to offload the execution of the jobs to slaves in the cloud thanks to ad-hoc plugins which will handle the creation of the cloud resources together with their destruction when they are not needed anymore:

- The EC2 Plugin[4] let Jenkins use AWS EC2 instances as cloud build resources when it runs out of on-premise slaves. The EC2 slaves will be dynamically created inside an AWS network and de-provisioned when they are not needed.

- The JCloud plugin[5] creates the possibility of executing the jobs on any cloud provider supported by JCloud libraries

[4] https://wiki.jenkins-ci.org/display/JENKINS/Amazon+EC2+Plugin
[5] https://wiki.jenkins-ci.org/display/JENKINS/JClouds+Plugin

- The CloudBees Cloud Connector Plugin[6] makes it possible to leverage a subscription to CloudBees' hosted Jenkins service, DEV@Cloud, to expand the build capabilities of an on-premise Jenkins installation. This plugin allows instances to offload some jobs to the CloudBees-managed cluster of Linux slaves in DEV@cloud.

2.4. Right-sizing Jenkins masters

2.4.1. Master division strategies

Designing the best Jenkins architecture for your organization is dependent on how you stratify the development of your projects and can be constrained by limitations of the existing Jenkins plugins.

The 3 most common forms of stratifying development by masters is:

1. **By environment (QA, DEV, etc)** - With this strategy, Jenkins masters are populated by jobs based on what environment they are deploying to.
   - **Pros**
     - Can tailor plugins on masters to be specific to that environment's needs
     - Can easily restrict access to an environment to only users who will be using that environment
   - **Cons**
     - Reduces ability to create workflows
     - No way to visualize the complete flow across masters
     - Outage of a master will block flow of all products
2. **By org chart** - This strategy is when masters are assigned to divisions within an organization.
   - **Pros**
     - Can tailor plugins on masters to be specific to that team's needs
     - Can easily restrict access to a division's projects to only users who are within that division
   - **Cons**

[6] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Cloud+Connector+Plugin

    - Reduces ability to create cross-division workflows
    - No way to visualize the complete flow across masters
    - Outage of a master will block flow of all products

3. **Group masters by product lines** - When a group of products, with on only critical product in each group, gets its own Jenkins masters.
    - **Pros**
        - Entire flows can be visualized because all steps are on one master
        - Reduces the impact of one master’s downtime on only affects a small subset of products
    - **Cons**
        - A strategy for restricting permissions must be devised to keep all users from having access to all items on a master.

When evaluating these strategies, it is important to weigh them against the vertical and horizontal scaling pitfalls discussed in the introduction.

Another note is that a smaller number of jobs translates to faster recovery from failures and more importantly a higher mean time between failures.

2.4.2. Calculating how many jobs, masters, and executors are needed

Having the best possible estimate of necessary configurations for a Jenkins installation allows an organization to get started on the right foot with Jenkins and reduces the number of configuration iterations needed to achieve an optimal installation. The challenge for Jenkins architects is that true limit of vertical scaling on a Jenkins master is constrained by whatever hardware is in place for the master, as well as harder to quantify pieces like the types of builds and tests that will be run on the build nodes.

There is a way to estimate roughly how many masters, jobs and executors will be needed based on build needs and number of developers served. These equations assume that the Jenkins master will have 5 cores with one core per 100 jobs (500 total jobs/master) and that teams will be divided into groups of 40.

If you have information on the actual number of available cores on your planned master, you can make adjustments to the “number of masters” equations accordingly.

The equation for **estimating the number of masters and executors needed** when the number of configured jobs is known is as follows:

```
masters = number of jobs/500
```

```
executors = number of jobs * 0.03
```

The equation for **estimating the maximum number of jobs, masters, and executors needed** for an organization based on the number of developers is as follows:

```
number of jobs = number of developers * 3.333
number of masters = number of jobs/500
number of executors = number of jobs * 0.03
```

These numbers will provide a good starting point for a Jenkins installation, but adjustments to actual installation size may be needed based on the types of builds and tests that an installation runs.

2.4.3. Scalable storage for masters

It is also recommended to choose a master with consideration for future growth in the number of plugins or jobs stored in your master’s *$JENKINS_HOME*. Storage is cheap and Jenkins does not require fast disk access to run well, so it is more advantageous to invest in a larger machine for your master over a faster one.

Different operating systems for the Jenkins master will also allow for different approaches to expandable storage:

- **Spanned Volumes on Windows** - On NTFS devices like Windows, you can create a spanned volume that allows you to add new volumes to an existing one, but have them behave as a single volume. To do this, you will have to ensure that Jenkins is installed on a separate partition so that it can be converted to a spanned volume later.

- **Logical Volume Manager for Linux** - LVM manages disk drives and allows logical volumes to be resized on the fly. Many distributions of Linux use LVM when they are installed, but Jenkins should have its our LVM setup.

- **ZFS for Solaris** - ZFS is even more flexible than LVM and spanned volumes and just requires that the *$JENKINS_HOME* be on its own filesystem. This makes it easier to create snapshots, backups, etc.

- **Symbolic Links -** For systems with existing Jenkins installations and who cannot use any of the above-mentioned methods, symbolic links (symlinks) may be used instead to store job folders on separate volumes with symlinks to those directories.

Additionally, to easily prevent a *$JENKINS_HOME* folder from becoming bloated, make it mandatory for jobs to discard build records after a specific time period has passed and/or after a specific number of builds have been run. This policy can be set on a job’s configuration page.

2.5. Setting up a backup policy

It is a best practice to take regular backups of your $JENKINS_HOME. A backup ensures that your Jenkins instance can be restored despite a misconfiguration, accidental job deletion, or data corruption.

CloudBees offers a plugin that allows backups to be taken using a Jenkins job, which you can read more about here[7]

2.5.1. Finding your $JENKINS_HOME

Windows

If you install Jenkins with the Windows installer, Jenkins will be installed as a service and the default *$JENKINS_HOME* will be "C:\Program Files (x86)\jenkins".

You can edit the location of your *$JENKINS_HOME* by opening the jenkins.xml file and editing the *$JENKINS_HOME* variable, or going to the "Manage Jenkins" screen, clicking on the "Install as Windows Service" option in the menu, and then editing the "Installation Directory" field to point to another existing directory.

Mac OSX

If you install Jenkins with the OS X installer, you can find and edit the location of your *$JENKINS_HOME* by editing the "Macintosh HD/Library/LaunchDaemons" file's *$JENKINS_HOME* property.

By default, the *$JENKINS_HOME* will be set to "Macintosh HD/Users/Shared/Jenkins".

Ubuntu/Debian

If you install Jenkins using a Debian package, you can find and edit the location of your *$JENKINS_HOME* by editing your "/etc/default/jenkins" file.

By default, the *$JENKINS_HOME* will set to "/var/lib/jenkins" and your $JENKINS_WAR will point to "/usr/share/jenkins/jenkins.war".

Red Hat/CentOS/Fedora

If you install Jenkins as a RPM package, the default *$JENKINS_HOME* will be "/var/lib/jenkins".

You can edit the location of your *$JENKINS_HOME* by editing the "/etc/sysconfig/jenkins" file.

openSUSE

If installing Jenkins as a package using zypper, you'll be able to edit the *$JENKINS_HOME* by editing the "/etc/sysconfig/jenkins" file.

[7] http://jenkins-cookbook.cloudbees.com/docs/jenkins-cookbook/_backing_up_with_cloudbees_backup_plugin.html

The default location for your *$JENKINS_HOME* will be set to “/var/lib/jenkins” and the $JENKINS_WAR home will be in “/usr/lib/jenkins”.

FreeBSD

If installing Jenkins using a port, the *$JENKINS_HOME* will be located in whichever directory you run the “make” command in. It is recommended to create a “/usr/ports/devel/jenkins” folder and compile Jenkins in that directory.

You will be able to edit the *$JENKINS_HOME* by editing the “/usr/local/etc/jenkins”.

OpenBSD

If installing Jenkins using a package,the *$JENKINS_HOME* is set by default to “/var/jenkins”.

If installing Jenkins using a port, the *$JENKINS_HOME* will be located in whichever directory you run the “make” command in. It is recommended to create a “/usr/ports/devel/jenkins” folder and compile Jenkins in that directory.

You will be able to edit the *$JENKINS_HOME* by editing the “/usr/local/etc/jenkins” file.

Solaris/OpenIndiana

The Jenkins project voted on September 17, 2014 to discontinue Solaris packages.

2.5.2. Anatomy of a $JENKINS_HOME

The folder structure for a *$JENKINS_HOME* directory is as follows:

```
JENKINS_HOME
 +- config.xml     (Jenkins root configuration file)
 +- *.xml          (other site-wide configuration files)
 +- identity.key   (RSA key pair that identifies an instance)
 +- secret.key     (deprecated key used for some plugins’ secure operations)
 +- secret.key.not-so-secret  (used for validating _$JENKINS_HOME_ creation date)
 +- userContent    (files served under your http://server/userContent/)
 +- secrets        (root directory for the secret+key for credential decryption)
     +- hudson.util.Secret   (used for encrypting some Jenkins data)
     +- master.key           (used for encrypting the hudson.util.Secret key)
     +- InstanceIdentity.KEY (used to identity this instance)
 +- fingerprints   (stores fingerprint records, if any)
 +- plugins        (root directory for all Jenkins plugins)
     +- [PLUGINNAME]   (sub directory for each plugin)
         +- META-INF       (subdirectory for plugin manifest + pom.xml)
         +- WEB-INF        (subdirectory for plugin jar(s) and licenses.xml)
     +- [PLUGINNAME].jpi   (.jpi or .hpi file for the plugin)
 +- jobs           (root directory for all Jenkins jobs)
     +- [JOBNAME]      (sub directory for each job)
         +- config.xml     (job configuration file)
         +- workspace      (working directory for the version control system)
         +- latest         (symbolic link to the last successful build)
         +- builds         (stores past build records)
```

```
            +- [BUILD_ID]     (subdirectory for each build)
                +- build.xml      (build result summary)
                +- log            (log file)
                +- changelog.xml  (change log)
    +- [FOLDERNAME]   (sub directory for each folder)
        +- config.xml     (folder configuration file)
        +- jobs           (sub directory for all nested jobs)
```

2.5.3. Choosing a backup strategy

All of your Jenkins-specific configurations that need to be backed up will live in the *$JENKINS_HOME*, but it is a best practice to back up only a subset of those files and folders.

Below are a few guidelines to consider when planning your backup strategy.

Exclusions. When it comes to creating a backup, it is recommended to exclude archiving the following folders to reduce the size of your backup:

```
/war      (the exploded Jenkins war directory)
/cache    (downloaded tools)
/tools    (extracted tools)
```

These folders will automatically be recreated the next time a build runs or Jenkins is launched.

Jobs and Folders. Your job or folder configurations, build histories, archived artifacts, and workspace will exist entirely within the *jobs* folder.

The *jobs* directory, whether nested within a folder or at the root level is as follows:

```
 +- jobs           (root directory for all Jenkins jobs)
    +- [JOBNAME]      (sub directory for each job)
        +- config.xml     (job configuration file)
        +- workspace      (working directory for the version control system)
        +- latest         (symbolic link to the last successful build)
        +- builds         (stores past build records)
            +- [BUILD_ID]     (subdirectory for each build)
                +- build.xml      (build result summary)
                +- log            (log file)
                +- changelog.xml  (change log)
```

If you only need to backup your job configurations, you can opt to only backup the *config.xml* for each job. Generally build records and workspaces do not need to be backed up, as workspaces will be re-created when a job is run and build records are only as important as your organizations deems them.

System configurations. Your instance’s system configurations exist in the root level of the *$JENKINS_HOME* folder:

```
 +- config.xml     (Jenkins root configuration file)
 +- *.xml           (other site-wide configuration files)
```

The *config.xml* is the root configuration file for your Jenkins. It includes configurations for the paths of installed tools, workspace directory, and slave agent port.

Any .xml other than that *config.xml* in the root Jenkins folder is a global configuration file for an installed tool or plugin (i.e. Maven, Git, Ant, etc). This includes the *credentials.xml* if the Credentials plugin is installed.

If you only want to backup your core Jenkins configuration, you only need to back up the *config.xml*.

Plugins. Your instance's plugin files (.hpi and .jpi) and any of their dependent resources (help files, *pom.xml* files, etc) will exist in the *plugins* folder in $JENKINS_HOME.

```
 +- plugins         (root directory for all Jenkins plugins)
     +- [PLUGINNAME]     (sub directory for each plugin)
         +- META-INF       (subdirectory for plugin manifest + pom.xml)
         +- WEB-INF        (subdirectory for plugin jar(s) and licenses.xml)
     +- [PLUGINNAME].jpi (.jpi or .hpi file for the plugin)
```

It is recommended to back up the entirety of the plugins folder (.hpi/.jpis + folders).

Other data. Other data that you are recommended to back up include the contents of your *secrets* folder, your *identity.key*, your *secret.key*, and your *secret.key.not-so-secret* file.

```
+- identity.key   (RSA key pair that identifies an instance)
 +- secret.key     (used for various secure Jenkins operations)
 +- secret.key.not-so-secret  (used for validating _$JENKINS_HOME_ creation date)
 +- userContent    (files served in http://server/userContent/)
 +- secrets        (directory for the secret+key decryption)
     +- hudson.util.Secret   (used for encrypting some Jenkins data)
     +- master.key           (used for encrypting the hudson.util.Secret key)
     +- InstanceIdentity.KEY (used to identity this instance)
```

The *identity.key* is an RSA key pair that identifies and authenticates the current Jenkins instance.

The *secret.key* is used to encrypt plugin and other Jenkins data, and to establish a secure connection between a master and slave.

The *secret.key.not-so-secret* file is used to validate when the *$JENKINS_HOME* was created. It is also meant to be a flag that the secret.key file is a deprecated way of encrypting information.

The files in the secrets folder are used by Jenkins to encrypt and decrypt your instance's stored credentials, if any exist. Loss of these files will prevent recovery of any stored credentials. *hudson.util.Secret* is used for encrypting some Jenkins data like the credentials.xml, while the *master.key* is used for encrypting the hudson.util.Secret key. Finally, the *InstanceIdentity.KEY* is used to identity this instance and for producing digital signatures.

2.5.4. Define a Jenkins instance to rollback to

In the case of a total machine failure, it is important to ensure that there is a plan in place to get Jenkins both back online and in its last good state.

If a high availability set up has not been enabled and no back up of that master’s filesystem has been taken, then an corruption of a machine running Jenkins means that all historical build data and artifacts, job and system configurations, etc. will be lost and the lost configurations will need to be recreated on a new instance.

1. Backup policy - In addition to creating backups using the previous section’s backup guide, it is important to establish a policy for selecting which backup should be used when restoring a downed master.

2. Restoring from a backup - A plan must be put in place on whether the backup should be restored manually or with scripts when the primary goes down.

2.6. Resilient Jenkins Architecture

Administrators are constantly adding more and more teams to the software factory, making administrators in the business of making their instances resilient to failures and scaling them in order to onboard more teams.

Adding build nodes to a Jenkins instance while beefing up the machine that runs the Jenkins master is the typical way to scale Jenkins. Said differently, administrators scale their Jenkins master vertically. However, there is a limit to how much an instance can be scaled. These limitations are covered in the introduction to this chapter.

Ideally, masters will be set up to automatically recover from failures without human intervention. There are proxy servers monitoring active masters and re-routing requests to backup masters if the active master goes down. There are additional factors that should be reviewed on the path to continuous delivery. These factors include componetizing the application under development, automating the entire pipeline (within reasonable limits) and freeing up contentious resources.

Step 1: Make each master highly available. Each Jenkins master needs to be set up such that it is part of a Jenkins cluster. One way to do this is to upgrade the master to run CloudBees Jenkins Enterprise, which includes a High Availability plugin[8] that allows for an automatic failover when a master goes down and to notify other nodes in the cluster of the failure.

[8] http://jenkins-cookbook.cloudbees.com/docs/jenkins-cookbook/_setting_up_high_availability_with_cloudbees_high_availability_plugin.html

A proxy (typically HAProxy or F5) then fronts the primary master. The proxy’s job is to continuously monitor the primary master and route requests to the backup if the primary goes down. To make the infrastructure more resilient, you can have multiple backup masters configured.

Step 2: Enable security. Set up an authentication realm that Jenkins will use for its user database. If using CloudBees Jenkins Enterprise, enable the Role-based Access Control plugin that comes bundled with it.

Tip

If you are trying to set up a proof-of-concept, it is recommended to use the Mock Security Realm plugin[9] for authentication.

Step 3: Add build nodes (slaves) to master. Add build servers to your master to ensure you are conducting actual build execution off of the master, which is meant to be an orchestration hub, and onto a “dumb” machine with sufficient memory and I/O for a given job or test.

Step 4: Setup a test instance[10]. A test instance is typically used to test new plugin updates. When a plugin is ready to be used, it should be installed into the main production update center. :imagesdir: ../resources/

[9] https://wiki.jenkins-ci.org/display/JENKINS/Mock+Security+Realm+Plugin

[10] http://jenkins-cookbook.apps.cloudbees.com/docs/jenkins-cookbook/_test_instances.html

Chapter 3. Architecting for Manageability

3.1. Introduction

With over 1,000 plugins and countless versions of said plugins in the open-source community, testing for all possible conflicts before upgrading one or more production Jenkins instances is simply not feasible. While Jenkins itself will warn of potential incompatibility if it detects that a plugin requires a newer version of the Jenkins core, there is no automatic way to detect conflicts between plugins or to automatically quantifying the impact of upgrading a plugin before doing the upgrade.

Instead, Jenkins administrators should be made responsible for manually testing their own instance’s plugin and core version updates before performing them on the production instance. This kind of testing requires a copy or “test instance” of the production server to act as the sandbox for such tests and can prevent production master downtime.

3.2. Test Instances

A test master is a Jenkins master used solely for testing configurations and plugins in a non-production environment. For organizations with a mission-critical Jenkins instance, having a test master is highly recommended.

Upgrading or downgrading either the Jenkins core or any plugins can sometimes have the unintended side effect of crippling another plugin’s functionality or even crashing a master. As of today, there is no better way to pre-test for such catastrophic conflicts than with a test master.

Test masters should have identical configurations, jobs, and plugins as the production master so that test upgrades will most closely resemble the outcomes of a similar change on your production master. For example, installing the Folders plugin while running a version of the Jenkins core older than 1.554.1 will cause the instance crash and be inaccessible until the plugin is manually uninstalled from the *plugin* folder.

3.2.1. Setting up a test instance

There are many methods for setting up a test instance, but the commonality between them all is that the *$JENKINS_HOME* between them is nearly identical. Whether this means that most all of the production masters’ *$JENKINS_HOME* folder is version controlled in a service like GitHub and mounted manually or programmatically to a test server or Docker container, or whether the settings are auto-pushed or the test instance automatically created by CloudBees Jenkins Operations Center, the result is nearly the same.

It is ideal to first ensure sure the master is idle (no running or queued jobs) before attempting to create a test master.

If you are running CloudBees Jenkins Enterprise, there are also a few files specific to CJE which you should not copy over into your alternate installation:

- *identity.key* should not be copied to the test master, as this identifies the installation generally. For the new installation you will need a distinct test license.
- The *jgroups* subdirectory should not be copied. Otherwise the new installation may attempt to join an HA cluster with the old installation.
- *cluster-identity.txt*, if present (only created if you have a custom jgroups.xml), should not be copied.

With GitHub + manual commands

You will simply need to open up your command-line interface and “cd” to the folder that contains the *$JENKINS_HOME* directory for your production master and run the “git init” command. For the location of this folder, please refer to section 3.

It is recommended that before running the “git add” command that you create a good *.gitignore* file. This file will prevent you from accidentally version-controlling a file that is too large to be stored in GitHub (anything >50 MB).

Here is an example *.gitignore* file for a Jenkins master running on OS X:

LSOverride.

```
Icon
._*
.Spotlight-V100
.Trashes
.AppleDB
.AppleDesktop
Network Trash Folder
Temporary Items
.apdisk
*.log
*.tmp
*.old
*.jar
*.son
.Xauthority
.bash_history
.bash_profile
.fontconfig
.gitconfig
.gem
.lesshst
.mysql_history
```

```
.owner
.ri
.rvm
.ssh
.viminfo
.vnc
bin/
tools/
**/.owner
**/queue.xml
**/fingerprints/
**/shelvedProjects/
**/updates/
**/jobs/*/workspace/
**/war/
/tools/
**/custom_deps/
**/slave/workspace/
**/slave-slave.log.*
cache/
fingerprints/
**/wars/jenkins*.war
*.log
*.zip
*.rrd
*.gz
```

Once you have a good *.gitignore* file, you can run the follow git commands to commit your *$JENKINS_HOME* to a git repository like GitHub:

```
git add -—all
git commit -m “first commit”
git push
```

Now you can install Jenkins to a fresh instance and “git clone” this *$JENKINS_HOME* from the git repository to your new instance. You will need to replace the files in the new instance with your version-controlled files to complete the migration, whether through scripts or through a drag-and-drop process.

Once this is done, you will need to restart the new test master’s Jenkins service or reload its configuration from the Jenkins UI (“Manage Jenkins” >> “Reload Configuration”).

With GitHub + Docker (Linux-only)

When it comes to version controlling your $JENKINS_HOME, just follow the instructions in the previous section.

The next step will be to create a Docker image with identical configurations to your production instance’s - operating system (Linux-only), installed libraries/tools, and open ports. This can be accomplished through Dockerfiles.

You will then just need to create mounted storage on your Docker server with a clone of your version-controlled *$JENKINS_HOME* home and a simple image to clone the *$JENKINS_HOME* into.

For example, we can create a Docker image called *jenkins-storage* and version control our *$JENKINS_HOME* in a Github repository known as “demo-joc”. The “jenkins-storage” Docker image can be built from a Dockerfile similar to this:

```
FROM debian:jessie
RUN apt-get update && apt-get -y upgrade
RUN apt-get install -y --no-install-recommends \
    openjdk-7-jdk \
    openssh-server \
    curl \
    ntp \
    ntpdate  \
    git  \
    maven  \
    less  \
    vim
RUN printf "AddressFamily inet" >> /etc/ssh/ssh_config
ENV MAVEN_HOME /usr/bin/mvn
ENV GIT_HOME /usr/bin/git
# Install Docker client
RUN curl https://get.docker.io/builds/Linux/x86_64/docker-latest -o /usr/local/bin/
RUN chmod +x /usr/local/bin/docker
RUN groupadd docker
# Create Jenkins user
RUN useradd jenkins -d /home/jenkins
RUN echo "jenkins:jenkins" | chpasswd
RUN usermod -a -G docker jenkins
# Make directories for [masters] JENKINS_HOME, jenkins.war lib and [slaves] remote
RUN mkdir /usr/lib/jenkins /var/lib/jenkins /home/jenkins /var/run/sshd
# Set permissions
RUN chown -R jenkins:jenkins /usr/lib/jenkins /var/lib/jenkins /home/jenkins
#create data folder for cloning
RUN ["mkdir", "/data"]
RUN ["chown", "-R", "jenkins:jenkins", "/data"]
USER jenkins
VOLUME ["/data"]
WORKDIR /data
# USER jenkins
CMD ["git", "clone", "https://github.com/[your-github-id]/docker-jenkins-storage.gi
```

Creating mounted storage for containers would just require something similar to the following command:

```
docker run --name storage [your-dockerhub-id]/jenkins-storage git clone https://git
```

And Jenkins images that rely on the mounted storage for their *$JENKNIS_HOME* will then need to point to the mounted volume:

```
docker run -d --dns=172.17.42.1 --name joc-1 --volumes-from storage -e JENKINS_HOME
```

Note that Docker only supports one mounted volume at a time, so if you are planning on running multiple test instances on Docker, all of their _$JENKINS_HOME_s will need to be version controlled in the same GitHub repo.

With Jenkins Operations Center

CloudBees Jenkins Operations Center can push plugins, core Jenkins versions, and security configurations to any Jenkins master that is managed by it. This makes it possible to attach a test server instance (whether that be a Docker container, EC2 instance, vSphere VM, etc) and CJOC will automatically push to the master those pre-configured settings once a connection is established.

To keep both masters in sync plugin/core-wise, 3 Custom Update Centers would need to be set up using the Custom Update Center plugin by CloudBees. The update center would need to be hosted on CloudBees Jenkins Operations Center and maintained by the Jenkins administrator.

Update centers can take several different upstream resources:

1. The **open source update center** - https://updates.jenkins-ci.org/current/update-center.json - this is the default update center that ships with open-source Jenkins instances.

2. The **experimental open source update center** - http://updates.jenkins-ci.org/experimental/update-center.json - this update center only contains experimental plugins which are not yet stable or ready for production, but which represent the most cutting edge version of Jenkins' feature set.

3. The **CloudBees update center** - http://jenkins-updates.cloudbees.com/updateCenter/WwwvhPlQ/update-center.json - this update center includes CloudBees' proprietary plugins and their dependencies.

4. The **CloudBees experimental update center** - http://jenkins-updates.cloudbees.com/updateCenter/HcEXz-Ow/update-center.json - like its open source equivalent, this update center contains the most experimental and cutting edge versions of the CloudBees plugin set, but which are not necessarily ready for production.

5. The **CloudBees Jenkins Operations Center update center** - http://jenkins-updates.cloudbees.com/update-center/operations-center/update-center.json - This update center only contains plugins for the CloudBees Jenkins Operations Center product.

6. The **CloudBees Jenkins Operations Center experimental update center** - http://jenkins-updates.cloudbees.com/updateCenter/xcdPj_ZA/update-center.json

- Like the previous experimental update centers, this UC contains the most cutting edge versions of the CloudBees Jenkins Operations Center product’s feature set.

With this in mind, here is how a test instance could be set up with a combination of the above update centers:

- **Main update center** - This custom update center would only take the OSS plugins as its upstream plugins. The administrator would need to select which plugins to store and which versions to push to any downstream update center. This update center should be configured to automatically promote the latest plugin.

- **Production update center** - This custom update center would need to take the main update center as its upstream and be configured to **not** automatically take the latest version. This allows the administrator more control over what plugins will be available to the downstream master, in this case the production master. This will in turn prevent users of the downstream master from being able to upgrade beyond an approved version of a plugin of the Jenkins core.

- **Test update center** - This customer update center would need to take the main update center as its upstream and be configured to automatically take the latest version of its plugins. This allows the test environment to always have access to the latest plugins to be tested against your environment. The test master will be the downstream master for this update center.

The only further configuration that would need to be duplicated would be the jobs, which can be accomplished by copy/pasting the jobs folder from the production master to the target test master or by a script that is run by a Cluster Operation on CJOC. Such a custom script can be configured to run after certain triggers or at certain intervals.

Test master slaves. Test masters can be connected to test slaves, but this will require further configurations. Depending on your implementation of a test instance, you will either need to create a Jenkins Docker slave image or a slave VM. Of course, open-source plugins like the EC2 plugin also the option of spinning up new slaves on-demand.

If you are not using CloudBees Jenkins Operations Center, the slave connection information will then need to be edited in the config.xml located in your test master’s *$JENKINS_HOME*.

If using Jenkins Operations Center, no further configuration is required so long as the test master has been added as a client master to CJOC.

Rolling back plugins that cause failures. If you discover that a plugin update is causing conflict within the test master, you can rollback in several ways:

- For bad plugins, you can rollback the plugin from the UI by going to the plugin manager (“Manage Jenkins” >> “Manage Plugins”) and going to the “Available” tab. Jenkins will show a “downgrade” button next to any plugins that can be downgraded.

- If the UI is unavailable, then enter your *$JENKINS_HOME* folder and go to the plugins folder. From there, delete the .hpi or .jpi file for the offending plugin, then restart Jenkins. If you need to rollback to an older version, you will need to manually copy in an older version of that .jpi or .hpi. To do this, go to the plugin’s page on the Jenkins wiki[1] and download one of its archived versions. :imagesdir: ../resources/

[1] http://updates.jenkins-ci.org/download/plugins

Chapter 4. Continuous Delivery with Jenkins Workflow

4.1. Introduction

Continuous delivery allows organizations to deliver software with lower risk. The path to continuous delivery starts by modeling the software delivery pipeline used within the organization and then focusing on the automation of it all. Early, directed feedback, enabled by pipeline automation enables software delivery more quickly over traditional methods of delivery.

Jenkins is the Swiss army knife in the software delivery toolchain. Developers and operations (DevOps) personnel have different mindsets and use different tools to get their respective jobs done. Since Jenkins integrates with a huge variety of toolsets, it serves as the intersection point between development and operations teams.

Many organizations have been orchestrating pipelines with existing Jenkins plugins for several years. As their automation sophistication and their own Jenkins experience increases, organizations inevitably want to move beyond simple pipelines and create complex flows specific to their delivery process.

These Jenkins users require a feature that treats complex pipelines as a first-class object, and so CloudBees engineers developed and contributed the new Jenkins Workflow feature[1] to the Jenkins open source project. This Workflow plugin was built with the community's requirements for a flexible, extensible, and script-based pipeline in mind.

4.2. Prerequisites

Continuous delivery is a process - rather than a tool - and requires a mindset and culture that must percolate from the top-down within an organization. Once the organization has bought into the philosophy, the next and most difficult part is mapping the flow of software as it makes its way from development to production.

The root of such a pipeline will always be an orchestration tool like a Jenkins, but there are some key requirements that such an integral part of the pipeline must satisfy before it can be tasked with enterprise-critical processes:

- **Zero or low downtime disaster recovery**: A commit, just as a mythical hero, encounters harder and longer challenges as it makes its way down the pipeline. It is not unusual to see pipeline executions that last days. A hardware or a Jenkins

[1] https://wiki.jenkins-ci.org/display/JENKINS/Workflow+Plugin

failure on day six of a seven-day pipeline has serious consequences for on-time delivery of a product.

- **Audit runs and debug ability**: Build managers like to see the exact execution flow through the pipeline, so they can easily debug issues.

To ensure a tool can scale with an organization and suitably automate existing delivery pipelines without changing them, the tool should also support:

- **Complex pipelines**: Delivery pipelines are typically more complex than canonical examples (linear process: Dev#Test#Deploy, with a couple of operations at each stage). Build managers want constructs that help parallelize parts of the flow, run loops, perform retries and so forth. Stated differently, build managers want programming constructs to define pipelines.

- **Manual interventions**: Pipelines cross intra-organizational boundaries necessitating manual handoffs and interventions. Build managers seek capabilities such as being able to pause a pipeline for a human to intervene and make manual decisions.

The Workflow plugin allows users to create such a pipeline through a new job type called Workflow. The flow definition is captured in a Groovy script, thus adding control flow capabilities such as loops, forks and retries. Workflow allows for stages with the option to set concurrencies, preventing multiple builds of the same workflow from trying to access the same resource at the same time.

4.3. Concepts

Workflow Job Type. There is just one job to capture the entire software delivery pipeline in an organization. Of course, you can still connect two workflow job types together if you want. A Workflow job type uses a Groovy-based DSL for job definitions. The DSL affords the advantage of defining jobs programmatically:

```
node(‘linux’){
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def mvnHome = tool 'M3'
  env.PATH = "${mvnHome}/bin:${env.PATH}"
  sh 'mvn -B clean verify'
}
```

Stages. Intra-organizational (or conceptual) boundaries are captured through a primitive called “stages.” A deployment pipeline consists of various stages, where each subsequent stage builds on the previous one. The idea is to spend as few resources as possible early in the pipeline and find obvious issues, rather than spend a lot of computing resources for something that is ultimately discovered to be broken.

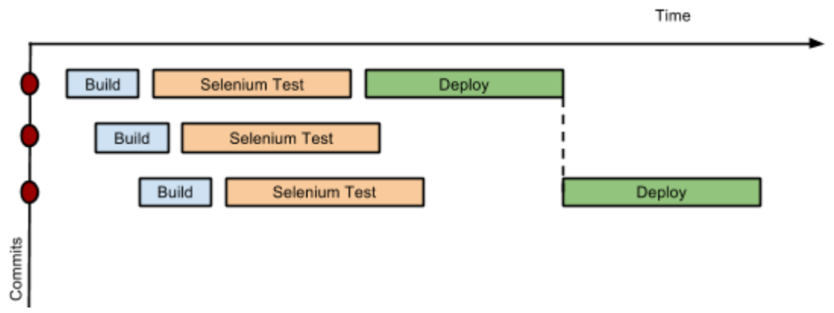


Figure 4.1. Throttled stage concurrency with Workflow

Consider a simple pipeline with three stages. A naive implementation of this pipeline can sequentially trigger each stage on every commit. Thus, the deployment step is triggered immediately after the Selenium test steps are complete. However, this would mean that the deployment from commit two overrides the last deployment in motion from commit one. The right approach is for commits two and three to wait for the deployment from commit one to complete, consolidate all the changes that have happened since commit one and trigger the deployment. If there is an issue, developers can easily figure out if the issue was introduced in commit two or commit three.

Jenkins Workflow provides this functionality by enhancing the stage primitive. For example, a stage can have a concurrency level of one defined to indicate that at any point only one thread should be running through the stage. This achieves the desired state of running a deployment as fast as it should run.

```
stage name: 'Production', concurrency: 1
   node {
       unarchive mapping: ['target/x.war' : 'x.war']
       deploy 'target/x.war', 'production'
       echo 'Deployed to http://localhost:8888/production/'
   }
```

Gates and Approvals. Continuous delivery means having binaries in a release ready state whereas continuous deployment means pushing the binaries to production - or automated deployments. Although continuous deployment is a sexy term and a desired state, in reality organizations still want a human to give the final approval before bits are pushed to production. This is captured through the “input” primitive in Workflow. The input step can wait indefinitely for a human to intervene.

```
input message: "Does http://localhost:8888/staging/ look good?"
```

Deployment of Artifacts to Staging/Production. Deployment of binaries is the last mile in a pipeline. The numerous servers employed within the organization and available in the market make it difficult to employ a uniform deployment step. Today, these are solved by third-party deployer products whose job it is to focus on deployment of a particular stack to a data center. Teams can also write their own extensions to hook into the Workflow job type and make the deployment easier.

Meanwhile, job creators can write a plain old Groovy function to define any custom steps that can deploy (or undeploy) artifacts from production.

```
def deploy(war, id) {
    sh "cp ${war} /tmp/webapps/${id}.war"
}
```

Restartable flows. All workflows are resumable, so if Jenkins needs to be restarted while a flow is running, it should resume at the same point in its execution after Jenkins starts back up. Similarly, if a flow is running a lengthy sh or bat step when a slave unexpectedly disconnects, no progress should be lost when connectivity is restored.

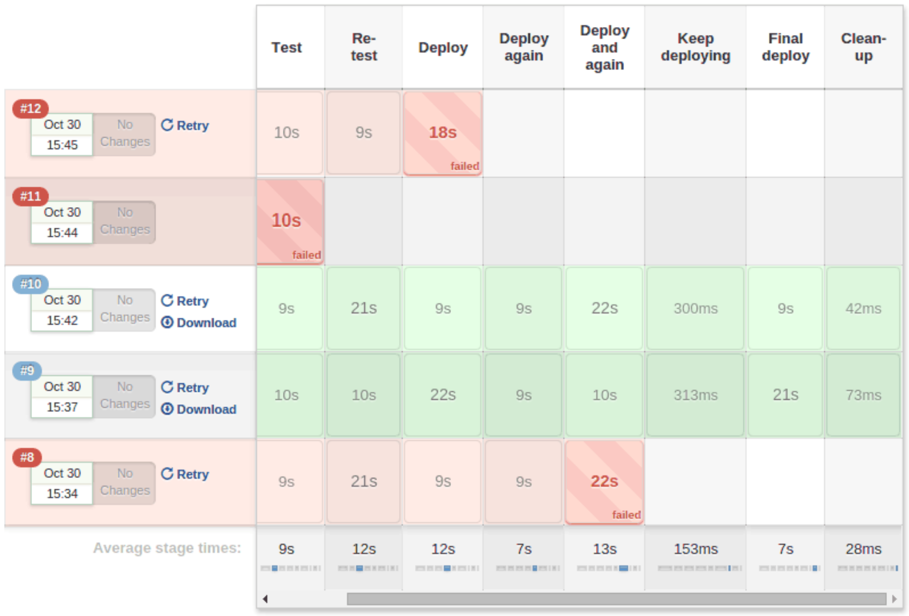
There are some cases when a flow build will have done a great deal of work and proceeded to a point where a transient error occurred: one which does not reflect the inputs to this build, such as source code changes. For example, after completing a lengthy build and test of a software component, final deployment to a server might fail because of network problems.

Instead of needed to rebuild the entire flow, the Checkpoints Workflow feature allows users to restart just the last portion of the flow, essentially running a new build with the tested artifacts of the last build. This feature is part of the CloudBees Jenkins Enterprise product.

```
node {
    sh './build-and-test'
}
checkpoint 'Completed tests'
node {
    sh './deploy'
}
```

Workflow Stage View. When you have complex builds pipelines, it is useful to see the progress of each stage and to see where build failures are occurring in the pipeline. This can enable users to debug which tests are failing at which stage or if there are other problems in their pipeline. Many organization also want to make their pipelines user-friendly for non-developers without having to develop a homegrown UI, which can prove to be a lengthy and ongoing development effort.

The Workflow Stage View feature offers extended visualization of workflow build history on the index page of a flow project. This visualization also includes helpful metrics like average run time by stage and by build, and a user-friendly interface for interacting with input steps.



	Test	Re-test	Deploy	Deploy again	Deploy and again	Keep deploying	Final deploy	Clean-up
#12 Oct 30 15:45 No Changes Retry	10s	9s	18s failed					
#11 Oct 30 15:44 No Changes	10s failed							
#10 Oct 30 15:42 No Changes Retry Download	9s	21s	9s	9s	22s	300ms	9s	42ms
#9 Oct 30 15:37 No Changes Retry Download	10s	10s	22s	9s	10s	313ms	21s	73ms
#8 Oct 30 15:34 No Changes Retry	9s	21s	9s	9s	22s failed			
Average stage times:	9s	12s	12s	7s	13s	153ms	7s	28ms

Figure 4.2. CloudBees Workflow View

The only prerequisite for this plugin is a workflow with defined stages in the flow. There can be as many stages as you desired and they can be in a linear sequence, and the stage names will be displayed as columns in the Stage View UI. This feature is part of the CloudBees Jenkins Enterprise product.

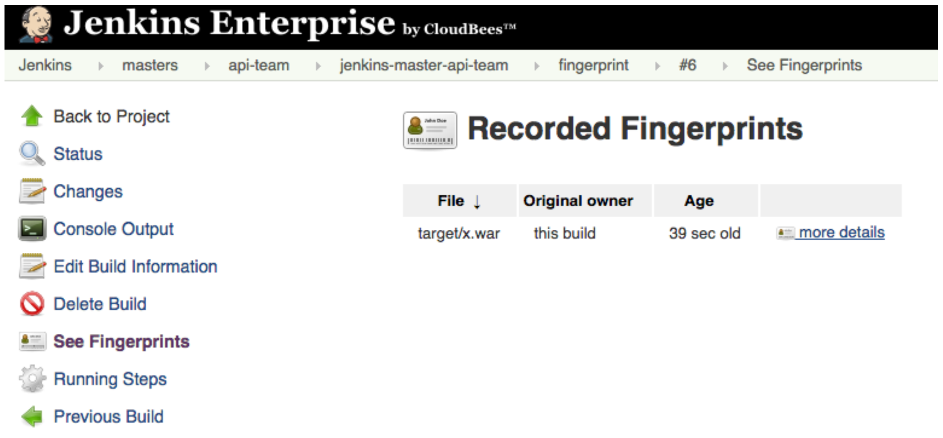
4.4. Artifact traceability with fingerprinting

Traceability is important for DevOps teams who need to be able to trace code from commit to deployment. It enables impact analysis by showing relationships between artifacts and allows for visibility into the full lifecycle of an artifact, from its code repository to where the artifact is eventually deployed in production.

Jenkins and the Workflow feature support tracking versions of artifacts using file fingerprinting, which allows users to trace which downstream builds are using any given artifact. To fingerprint with workflow, simply add a “fingerprint: true” argument to any artifact archiving step. For example:

```
step([$class: 'ArtifactArchiver', artifacts: '**/target/*.war’, fingerprint: true])
```

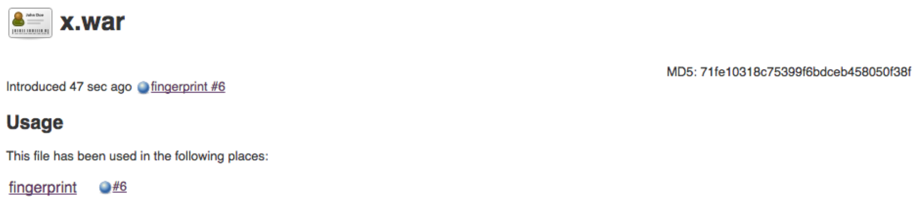
will archive any WAR artifacts created in the Workflow and fingerprint them for traceability. This trace log of this artifact and a list of all fingerprinted artifacts in a build will then be available in the left-hand menu of Jenkins:



File ↓	Original owner	Age	
target/x.war	this build	39 sec old	more details

Figure 4.3. List of fingerprinted files

To find where an artifact is used and deployed to, simply follow the “more details” link through the artifact’s name and view the entires for the artifact in its “Usage” list.



MD5: 71fe10318c75399f6bdceb458050f38f

Introduced 47 sec ago fingerprint #6

Usage

This file has been used in the following places:

fingerprint #6

Figure 4.4. Fingerprint of a WAR

For more information, visit the Jenkins community’s wiki[2] on how fingerprints work and their usage.

4.5. Workflow DSL Keywords

Workflows are defined as scripts that tell Jenkins what to do within the workflow, what tools to use, and which slave to use for the builds. These actions are pre-defined as steps and new steps can be created using plugins (see the “Plugin Compatibility” section for more). Below are some of the key steps in a workflow:

4.5.1. `node`: Allocate node

- This step schedules the tasks in its nested block on whatever node with the label specified in its argument. This step is required for all workflows so that Jenkins knows which slave should run the workflow steps.

[2] https://wiki.jenkins-ci.org/display/JENKINS/Fingerprint

- Params:
- `label`: String - label or name of a slave
- Nested block

```
node('master') {}
```

4.5.2. `stage`: Stage

- The “stage” command allows sections of the build to be constrained by a limited or unlimited concurrency, which is useful when preventing slow build stages like integration tests from overloading the build system.
- Params:
- `name`: String, mandatory
- `concurrency`: Integer that is greater than 0

```
stage 'Dev'
```

4.5.3. `input`: Input

- This step allows
- Params:
- `message` : String, default "Workflow has paused and needs your input before proceeding"
- `id`: Optional ID that uniquely identifies this input from all others.
- `submitter`: Optional user/group name who can approve this.
- `ok`: Optional caption of the OK button
- `parameters`: List<ParameterDefinition>

```
input ‘hello world’
```

4.5.4. `parallel`: Execute sub-workflows in parallel

- This step allows multiple actions to be performed at once, like integration tests or builds of multiple branches of a project.
- Params: Map, mandatory

```
def branches = [:]
```

```
parallel branches
```

4.5.5. `bat`: Windows Batch Script

- This step allows a Jenkins server or slave running on a Windows machine to execute a batch script.
- Params:
- `script`: String - the script to execute

```
bat "${mvnHome}\\bin\\mvn -B verify"
```

4.5.6. `sh`: Shell Script

- This step allows a Jenkins server or slave running on Linux or a Unix-like machine to execute a shell script.
- Params:
- `script`: String - the script to execute

```
sh "${mvnHome}/bin/mvn -B verify"
```

More information on the Workflow DSL can be found in the Workflow tutorial[3].

4.6. Setting up a basic build pipeline with Jenkins

Creating a pipeline in Jenkins with Workflow is as simple as a basic script:

```
node(‘linux’){
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def mvnHome = tool 'M3'
  env.PATH = "${mvnHome}/bin:${env.PATH}"
  sh 'mvn -B clean verify'
}
```

Where **node** is the step that schedules the tasks in the following block on whatever node with the label specified in its argument. In this case, the block’s tasks will only be run on a node with the label ‘linux’. The node block is required to tell Jenkins what system to run the commands.

git is the step that specifies what source code repository code should be checked from and does the checkout at this point.

The **tool** step makes sure a tool with the given name, in this case, a specific version of the Maven build tool, is installed on the current node. Merely running this step does

[3] https://github.com/jenkinsci/workflow-plugin/blob/master/TUTORIAL.md

not do much good; the script needs to know where it was installed, so the tool can be run later. For this, we need a variable.

The **def** keyword in Groovy is the quickest way to define a new variable, hence the “def mvnHome”. The return value of the **tool ‘M3’** check is assigned to the **mvnHome** variable.

Workflow also allows for the creation of very complex pipelines, with parallel stages, conditional logic gates, and for definitions to be loaded from version control and shared between workflows. This allows for workflows and certain standardized scripts to be shared between teams and changes to these scripts to be protected and reviewed by an administrator.

Here is an example script for such a scenario, where the bulk of the script is version controlled in GitHub:

```
def flow
node('master') {
    git branch: 'master', changelog: false, poll: true, url: 'https://github.com/la
    flow = load 'flow.groovy'
    flow.devQAStaging()
}
flow.production()
```

And here is the example script’s GitHub counterpart:

```
def devQAStaging() {
    env.PATH="${tool 'Maven 3.x'}/bin:${env.PATH}"
    stage 'Dev'
    sh 'mvn clean install package'
    archive 'target/x.war'
  try {
        checkpoint('Archived war')
    } catch (NoSuchMethodError _) {
        echo 'Checkpoint feature available in Jenkins Enterprise by CloudBees.'
    }
    stage 'QA'
    parallel(longerTests: {
        runWithServer {url ->
            sh "mvn -f sometests/pom.xml test -Durl=${url} -Dduration=30"
        }
    }, quickerTests: {
        runWithServer {url ->
            sh "mvn -f sometests/pom.xml test -Durl=${url} -Dduration=20"
        }
    })
    stage name: 'Staging', concurrency: 1
    deploy 'target/x.war', 'staging'
}
def production() {
    input message: "Does http://localhost:8888/staging/ look good?"
    try {
        checkpoint('Before production')
    } catch (NoSuchMethodError _) {
        echo 'Checkpoint feature available in Jenkins Enterprise by CloudBees.'
    }
    stage name: 'Production', concurrency: 1
    node {
        unarchive mapping: ['target/x.war' : 'x.war']
        deploy 'target/x.war', 'production'
        echo 'Deployed to http://localhost:8888/production/'
    }
}
def deploy(war, id) {
    sh "cp ${war} /tmp/webapps/${id}.war"
}
def undeploy(id) {
    sh "rm /tmp/webapps/${id}.war"
}
return this;
```

More information on complex pipelines can be found in the CloudBees Jenkins Enterprise documentation[4] and in the Workflow repository[5].

4.7. Plugin compatibility

Many source plugins have already added support for the Workflow feature, while others can be upgraded to support workflow through some code changes to utilize some newer APIs.

4.7.1. Supported SCMs

- `GitSCM` (`git`): supported as of 2.3; native `git` step also bundled
- `SubversionSCM` (`subversion`): supported as of the Subversion Plugin’s v 2.5; native `svn` step also bundled
- `MercurialSCM` (`mercurial`): supported as of 1.51
- `PerforceScm` (`p4`, not the older `perforce`): supported as of 1.2.0

4.7.2. Build steps and post-build actions

- `ArtifactArchiver` (core)
- `Fingerprinter` (core)

[4] http://jenkins-enterprise.cloudbees.com/docs/user-guide-docs/workflow.html
[5] https://github.com/jenkinsci/workflow-plugin/blob/master/TUTORIAL.md

- `JUnitResultArchiver` (`junit`)
- `JavadocArchiver` (`javadoc`)
- `Mailer` (`mailer`)
- `CopyArtifact` (`copyartifact`): JENKINS-24887[6] in 1.34

4.7.3. Build wrappers

- API to integrate build wrappers: JENKINS-24673[7]

4.7.4. Clouds

- `docker`: supported as of 0.8
- `nectar-vmware` (CloudBees Jenkins Enterprise): supported as of 4.3.2
- `ec2`: known to work as is

4.7.5. Miscellaneous

- `rebuild`: JENKINS-26024[8]
- `build-token-root`: JENKINS-26693[9]
- `credentials-binding`: `withCredentials` step as of 1.3
- `job-dsl`: implemented in 1.29

4.7.6. Custom steps

- `parallel-test-executor`: supported with `splitTests` step since 1.6
- `mailer`: JENKINS-26104[10] in Workflow 1.2

The most up to date list on available steps can always be found in the Workflow snippet generator. The snippet generator is located at the bottom of the configuration page for every Workflow job. Steps offered by a compatible plugin will appear in the snippet generator once the compatible plugin is installed.

[6] https://issues.jenkins-ci.org/browse/JENKINS-24887
[7] https://issues.jenkins-ci.org/browse/JENKINS-24673
[8] https://issues.jenkins-ci.org/browse/JENKINS-26024
[9] https://issues.jenkins-ci.org/browse/JENKINS-26693
[10] https://issues.jenkins-ci.org/browse/JENKINS-26104

There are some plugins which have not yet incorporated support for workflow. Guides on how to add this support, as well as their relevant Jenkins JIRA tickets can be found in the Workflow Plugin GitHub documentation[11]. :imagesdir: ../resources/

[11] https://github.com/jenkinsci/workflow-plugin/blob/master/COMPATIBILITY.md

Chapter 5. Virtualization

In general, bare-metal servers tend to perform faster than virtualized servers, which suffer from a reported 1-5% of CPU overhead and 5-10% of memory overhead on older hardware and 2-3% on newer hardware. This overhead can be accounted for when architecting an installation, but even with such overhead, virtualization still makes sense if scalability trumps performance or virtualization is being used for only certain components of a Jenkins installation.

5.1. Masters

In general, it is not recommended to virtualize the Jenkins master. Reasons include the fact that the number of jobs tend to increase over time, while the master itself is the orchestrator of these jobs' builds. At the same time, the number of executors connecting to the master similarly increases. This ever increasing load, combined with the loads of other VMs on the same machine invites performance problems on a virtualized Jenkins master. When the Jenkins masters are throttled, slaves connections will be dropped, killing all in-flight builds. To prevent this, masters that are set up on VMs will need to be regularly resized as the installation grows organically.

An exception to the general recommendation is when certain features exclusive to VMs are being leveraged, like VMware's live migration for vSphere. Such live migration allows for VMs to be copied between physical servers without downtime, which confers obvious and VM-exclusive benefits.

5.2. Slaves

In general, slaves are simply "dumb" machines that only need enough memory to run a few builds. This makes them suitable for being virtualized and even being ephemeral if utilizing a VM cloud, like VMware's vSphere.

Various open source plugins exist to provide integrations between various VM providers — CloudBees in particular offers a vSphere plugin which allows for a true cloud integration with vSphere. In this case, you can simply install a separate agent on a VM box and the CloudBees vSphere plugin to Jenkins, which is covered in Chapter 7.

It must be noted however, that running builds on virtualized slaves versus "bare metal" slaves may suffer from slower build times. Whether or not this hit will happen is based on many factors, but Grid Dynamics' Kirill Evstigneev had the following configuration and saw the average build time increase 25% when switching to virtualized slaves:

- Original "bare metal" slave

    - OS: CentOs 6 Linux
    - CPU: HT CPU with 4 cores
    - RAM: 8 GB RAM
    - Processor: i7
    - Storage: RAID0 with 2 rotating disks
- Xen 4 virtual machine
    - CPU: 7 logical cores
    - Storage: local disk

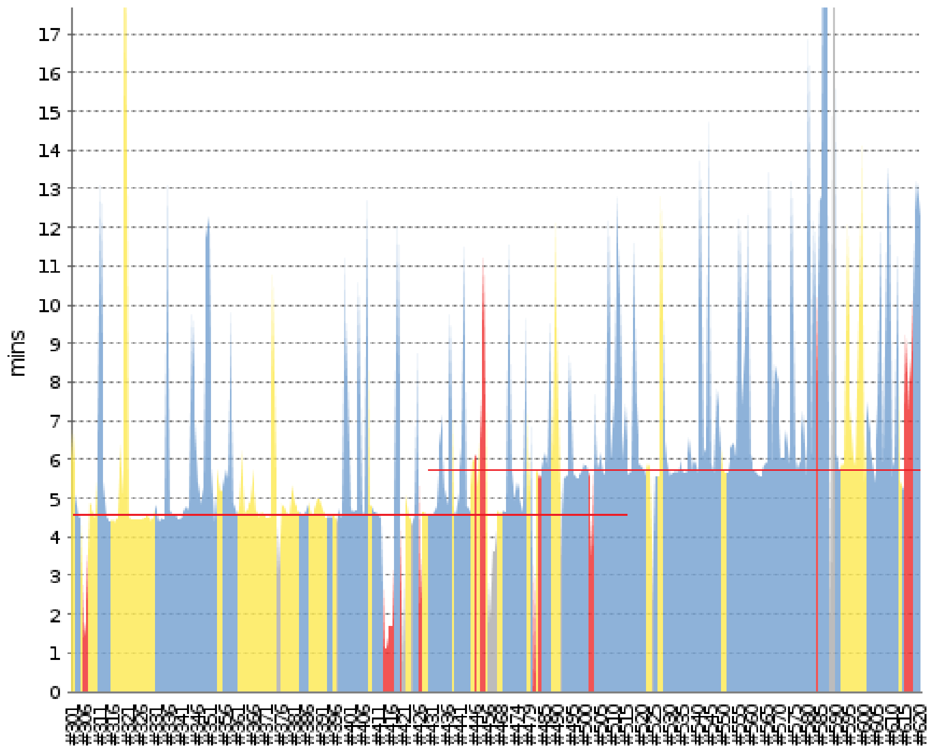


Figure 5.1. Build time comparison

Note

Metrics from soldering-iron.blogspot.com

However, there are many cases where any possible performance hit is dwarfed by a larger increase in productivity due to the elasticity and scalability of a virtualized environment.

Docker and other containerization technologies do not suffer from these performance hits due to an elimination of hardware virtualization layers and direct execution “on the metal”. scaledwidth=90%:imagesdir: ../resources/

Chapter 6. Containers

Containerization is an alternative approach to virtualization, such that the elimination of hardware virtualization layers and direct execution “on the metal” allow containers to be “light-weight” and not suffer from some of the performance issues of traditional VMs.

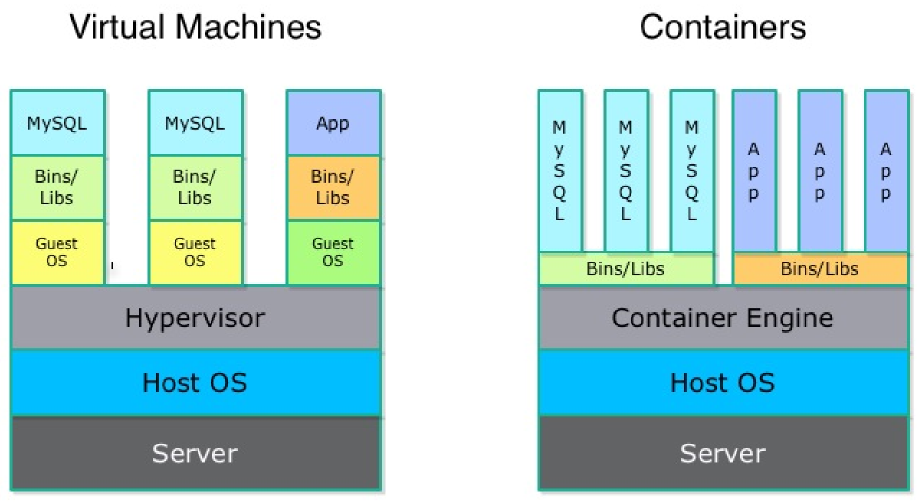


Figure 6.1. Container versus Virtual Machine

NOTES: image source is patg.net

This low overhead makes containers ideal for use as slaves and many open source Jenkins plugins support this use-case, so implementation is quite simple. Containers may also be used to containerize masters, though this can be a bit more difficult depending on your organizations’ requirements.

6.1. Docker

Docker is an open-source project that provides a platform for building and shipping applications using containers. This platform enables developers to easily create standardized environments that ensure that a testing environment is the same as the production environment, as well as providing a lightweight solution for virtualizing applications. Docker has inspired a wave of microservice architectures and is supported by tech giants like Google, Microsoft, IBM and Amazon.

The versatility and usability of Docker has made it a popular choice among DevOps-driven organizations. It has also made Docker an ideal choice for creating the standardized and repeatable environments.

6.1.1. Docker Concepts

Docker containers are lightweight runtime environments that consist of an application and its dependencies. These containers run “on the metal” of a machine, allowing them to avoid the 1-5% of CPU overhead and 5-10% of memory overhead associated with traditional virtualization technologies. They can also be created from a read-only template called a Docker image.

Docker images can be created from an environment definition called a Dockerfile or from a running Docker container which has been committed as an image. Once a Docker image exists, it can be pushed to a registry like Docker Hub and a container can be created from that image, creating a runtime environment with a guaranteed set of tools and applications installed to it. Similarly, containers can be committed to images which are then committed to Docker Hub.

6.1.2. Using Jenkins and Docker for End-to-End CD

Docker containers allow runtimes to be captured as an image and run in a lightweight way, making it an excellent choice for both building and shipping applications. If a change needs to be pushed to the container’s runtime, whether that be a new version of the application or a new tool installation, the container will have to be rebuilt using a Docker image. Just as with traditional continuous delivery, all such changes should be tested for regressions before being committed or new containers are created from it.

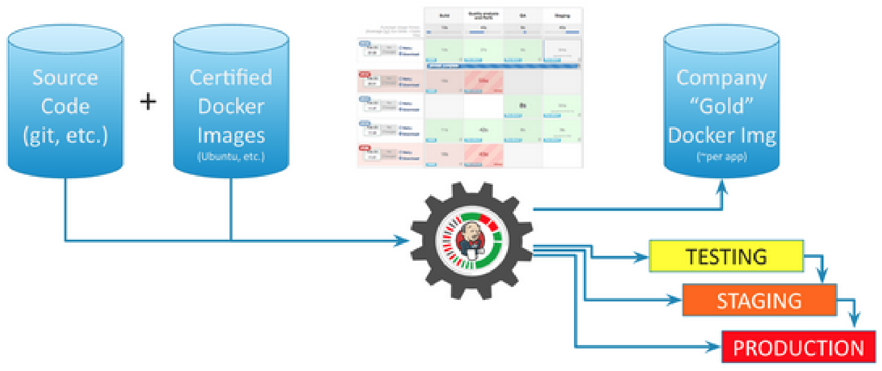


Figure 6.2. Automating application releases with Docker

This testing and validation is possible with “end-to-end CD”, where the environments captured by Docker containers are also subject to a pipeline of builds and tests, as well as human-approved pushes back to an image repository like Docker hub. This approach to continuously managing containers’ quality ensures that testing and production environments are always stable and adaptable to changes in an organization’s needs. To further enhance this end-to-end pipeline, Docker images can also be tracked, allowing individual images to be tied to any deployed containers and for easy rollbacks should a change in one image break another.

Because Docker offers the ability to quickly create multiple environments from one definition, Docker containers are also ideal for serving as standardized build environments for an organization.

The Jenkins open source community has plugins which enable all of these use cases for Docker and Jenkins, allowing Docker to be leveraged for continuous delivery and creating a Docker-based Jenkins architecture. The recommended open-source plugins for these use cases are:

- Custom Build Environments for Jenkins with the CloudBees Docker Custom Builds Environment Plugin[1] (formerly the Oki Docki plugin)
- Traceability of Docker images and containers in Jenkins with the CloudBees Docker Traceability Plugin[2]
- Triggering Jenkins pipelines with Docker Hub with the CloudBees Docker Hub Notification Plugin[3]
- Building and publishing Docker images with the CloudBees Docker Build and Publish Plugin[4]
- Orchestrating Workflows with Jenkins and Docker with the CloudBees Workflow Docker Plugin[5]

CloudBees also offers a way for sharing Docker slave configurations using Docker Swarm and the CloudBees Jenkins Operations Center product[6], which is part of the CloudBees Jenkins Platform.

6.1.3. Prerequisites

Most all plugins focused on Docker also require the installation of **Docker** to any slaves that will use the Docker CLI, like the CloudBees Docker Build and Publish Plugin.

Docker is an open source project and supports installation to a wide range of operating systems and platforms[7]. Though non-Linux operating systems will also require the installation of another open source project known as **Boot2Docker**. Boot2Docker allows non-Linux operating systems to run the Docker host in a **VirtualBox** environment.

[1] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Custom+Build+Environment+Plugin
[2] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Traceability
[3] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Hub+Notification
[4] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Build+and+Publish+plugin
[5] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Workflow+Plugin
[6] http://operations-center.cloudbees.com/docs
[7] https://docs.docker.com/installation

6.1.4. Custom Build Environments for Jenkins with the CloudBees Docker Custom Builds Environment Plugin[8] (formerly the Oki Docki plugin)

Builds often require that credentials or tooling be available to the slave node which runs it. For a small installation with few specialized jobs, this may be manageable using generic slaves, but when these requirements are multiplied by the thousands of jobs that many organizations running per day, managing and standardizing these slave environments becomes more challenging.

Docker has established itself as a popular and convenient way to bootstrap isolated and reproducible environments, which allows Docker containers serve as the easiest to maintain slave environments. Docker containers' tooling and other configurations can be version controlled in an environment definition a Dockerfile, and multiple identical containers can be created quickly using this definition.

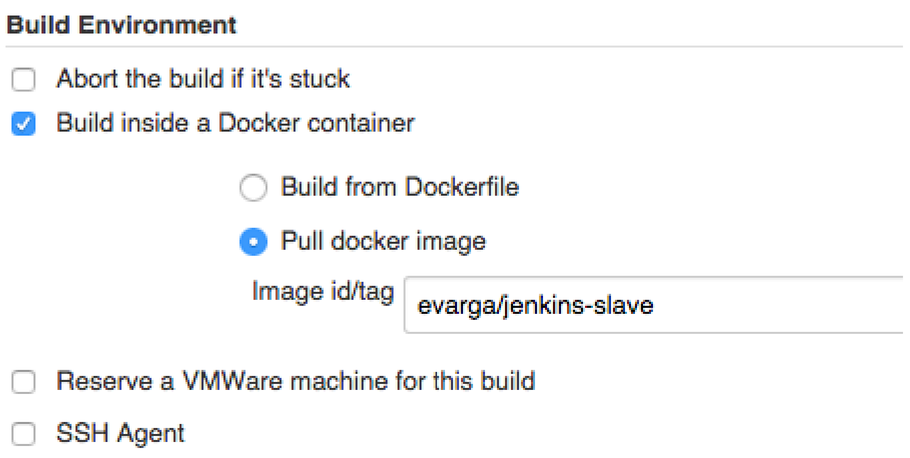
The CloudBees Custom Builds Environment Plugin allows Docker images and files to serve as template for Jenkins slaves, reducing the administrative overhead of a slave installation to only updating a few lines in a handful of environment definitions for potentially thousands of slaves.

Prerequisites. Machines which will host the Docker containers must have Docker both installed and running. It is highly recommended that such slaves be labeled "docker". Also note that Docker only supports Linux runtimes, so .NET, OS X, and other OS-dependent builds will not run in a Dockerized Linux environment.

The Docker build environment can be used to build any type of Jenkins job.

Configuration. This plugin adds the option "Build inside a Docker container" in the build environment configuration of a job. To enable it, simply scroll to the "Build Environment" section of any Jenkins job and select the "Build inside a Docker container" option. You will then be able to specify whether a slave container should be created from a Dockerfile checked into the workspace (e.g. "the file was in the root of the project to be built") or whether to pull an explicit image from a Docker registry to use as the slave container.

[8] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Custom+Build+Environment+Plugin



Build Environment

- [ ] Abort the build if it's stuck
- [✓] Build inside a Docker container
  - ○ Build from Dockerfile
  - ● Pull docker image
  - Image id/tag: evarga/jenkins-slave
- [ ] Reserve a VMWare machine for this build
- [ ] SSH Agent

Figure 6.3. Using Docker containers as slaves

For the latter option, you can leverage the most popular Docker slave image in Docker Hub called evarga/jenkins-slave[9] or create a new image with a custom Dockerfile. To create a new Dockerfile, you can fork the "evarga/jenkins-slave" or edit the below copy of it using the Dockerfile guidelines[10] and reference[11]:

```
FROM ubuntu:trusty
MAINTAINER Ervin Varga <ervin.varga@gmail.com>
RUN apt-get update
RUN apt-get -y upgrade
RUN apt-get install -y openssh-server
RUN sed -i 's|session    required     pam_loginuid.so|session    optional     pam_l
RUN mkdir -p /var/run/sshd
RUN apt-get install -y openjdk-7-jdk
RUN adduser --quiet jenkins
RUN echo "jenkins:jenkins" | chpasswd
EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

Builds which are built within a Docker container will display a Docker icon in a job's build history:

[9] https://registry.hub.docker.com/u/evarga/jenkins-slave/dockerfile
[10] https://docs.docker.com/articles/dockerfile_best-practices
[11] https://docs.docker.com/reference/builder

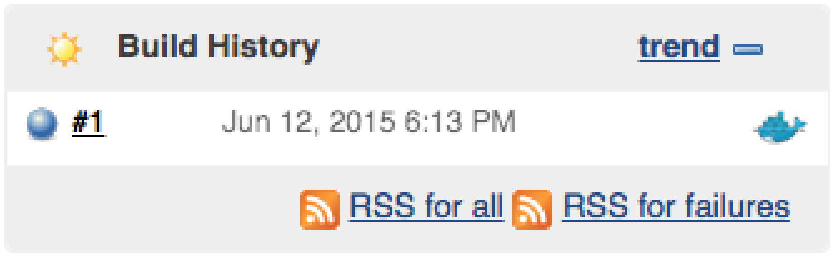


Figure 6.4. Docker builds indicator

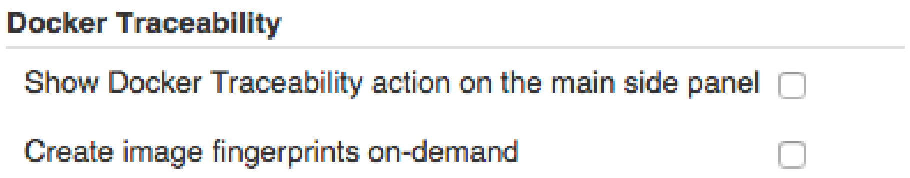
6.1.5. Traceability of Docker images and containers in Jenkins with the CloudBees Docker Traceability Plugin[12]

Traceability of artifacts used in a build is a requirement for many organizations due to audit requirements or for impact analysis. The best time to ensure traceability is when the artifacts are created and the cost of doing so is lowest.

CloudBees has created and contributed to the community the CloudBees Docker Traceability plugin, which traces the build and deployment history of Docker images and renders this information to the build page and an overall view. If there are images that are related to a build, this plugin will also display a list of “Docker fingerprints” for each image. If a base image has been used by a Jenkins build, the base image will also be linked to the build.

Prerequisites. This plugin requires installation of the “Docker Commons” plugin, which will provide the fingerprints of the Docker images.

Configuration. To begin tracing images, you will need to go to the Global Configuration page of your Jenkins instance (http://jenkins-url/configure) and scroll to a “Docker Traceability” section.


Docker Traceability

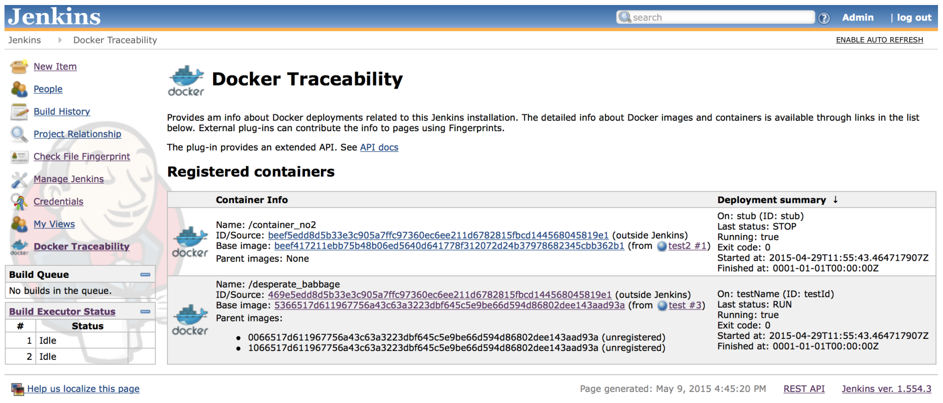
Show Docker Traceability action on the main side panel ☐

Create image fingerprints on-demand ☐

Figure 6.5. Configuring Jenkins to track Docker images

This section will allow a user to optionally create a link in the left-hand menu to an overall view of all tracked Docker images (http://jenkins-url/docker-traceability), though the direct link to this overall view will always be available.

[12] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Traceability



Jenkins | search | Admin | log out

Jenkins ▸ Docker Traceability | ENABLE AUTO REFRESH

- New Item
- People
- Build History
- Project Relationship
- Check File Fingerprint
- Manage Jenkins
- Credentials
- My Views
- Docker Traceability

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Docker Traceability

Provides am info about Docker deployments related to this Jenkins installation. The detailed info about Docker images and containers is available through links in the list below. External plug-ins can contribute the info to pages using Fingerprints.

The plug-in provides an extended API. See API docs

Registered containers

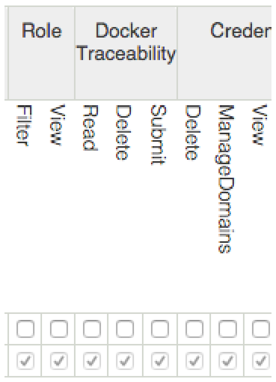
Container Info	Deployment summary ↓
Name: /container_no2 ID/Source: beef5edd8d5b33e3c905a7ffc97360ec6ee211d6782815fbcd144568045819e1 (outside Jenkins) Base image: beef417211ebb75b48b06ed5640d641778f312072d24b37978682345cbb362b1 (from test2 #1) Parent images: None	On: stub (ID: stub) Last status: STOP Running: true Exit code: 0 Started at: 2015-04-29T11:55:43.464717907Z Finished at: 0001-01-01T00:00:00Z
Name: /desperate_babbage ID/Source: 469e5edd8d5b33e3c905a7ffc97360ec6ee211d6782815fbcd144568045819e1 (outside Jenkins) Base image: 5366517d611967756a43c63a3223dbf645c5e9be66d594d86802dee143aad93a (from test #3) Parent images: • 0066517d611967756a43c63a3223dbf645c5e9be66d594d86802dee143aad93a (unregistered) • 1066517d611967756a43c63a3223dbf645c5e9be66d594d86802dee143aad93a (unregistered)	On: testName (ID: testId) Last status: RUN Running: true Exit code: 0 Started at: 2015-04-29T11:55:43.464717907Z Finished at: 0001-01-01T00:00:00Z

Help us localize this page | Page generated: May 9, 2015 4:45:20 PM | REST API | Jenkins ver. 1.554.3

Figure 6.6. A Docker image's build history and deployment locations

The other configuration option allows fingerprints to be created for docker images that have been created outside of Jenkins (e.g. with the Docker CLI). By default, the CloudBees Docker Traceability plugin will only collect traceability reports for images created within Jenkins and with plugins that utilize the Docker Commons Plugin.

Security. It is highly recommended to only use this plugin on a master with security enabled. This is because the JSON stored by this plugin may contain sensitive information. In light of this, this plugin also adds an extra set of security permissions that are compatible with the CloudBees Role-Based Access Control plugin:



Role		Docker Traceability			Creder		
Filter	View	Read	Delete	Submit	Delete	ManageDomains	View
☐	☐	☐	☐	☐	☐	☐	☐
✓	✓	✓	✓	✓	✓	✓	✓

Figure 6.7. Configuring Docker Traceability permissions

6.1.6. Triggering Jenkins pipelines with Docker Hub with the CloudBees Docker Hub Notification Plugin[13]

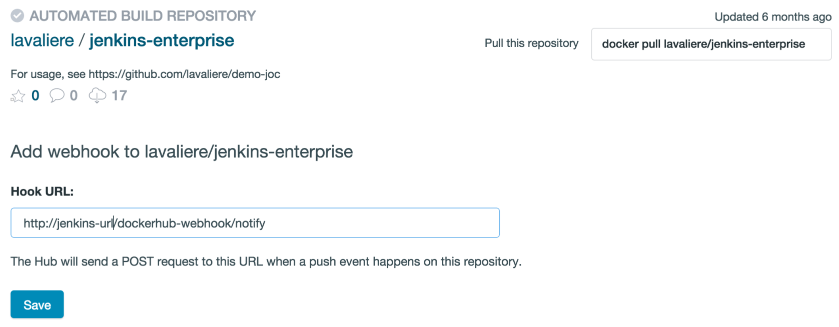
Any organization using Docker will also need to leverage a Docker registry to store any images they create for re-use in other images or for deployment, and Docker Hub is a Docker image registry which is offered by Docker Inc. as both a hosted service and a software for on-premise installations. Docker Hub allows images to be shared

[13] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Hub+Notification

and pulled for use as containers or as dependencies for other Docker images. Docker containers can also be committed to Docker Hub as images to save them in their current state. Docker Hub is to Docker images what GitHub has become for many developers' code — an essential tool for version and access control.

To allow Docker Hub to play as pivotal a role in continuous delivery as any source code repository, the CloudBees Docker Hub Notification plugin gives Docker Hub the ability to trigger application and slave environment builds, as well as application packaging, application releases via images, and application deployments via Docker containers.

Prerequisites. This plugin requires configuring a webhook in Docker Hub that points to the target Jenkins instance's Docker Hub endpoint (http://jenkins-url/dockerhub-webhook/notify):



AUTOMATED BUILD REPOSITORY

lavaliere / jenkins-enterprise

Updated 6 months ago

Pull this repository: docker pull lavaliere/jenkins-enterprise

For usage, see https://github.com/lavaliere/demo-joc

0 0 17

Add webhook to lavaliere/jenkins-enterprise

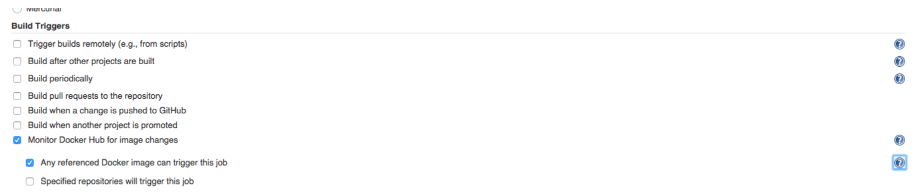
Hook URL: http://jenkins-url/dockerhub-webhook/notify

The Hub will send a POST request to this URL when a push event happens on this repository.

Figure 6.8. Docker webhook configuration

This is a generic hook for all Docker Hub repos that allows them to be linked to all downstream jobs in a master. You can configure up to 15 different jobs on the same webhook, and all jobs with this repository configured will be triggered once a notification is received.

Configuration. This plugin adds new a build trigger to both standard Jenkins jobs and Jenkins Workflows. This trigger is called "Monitor Docker Hub for image changes" and allows Jenkins to track when a given Docker image is rebuilt, whether that image is simply referenced by the job or is in a given repository.



Mercurial

Build Triggers

- ☐ Trigger builds remotely (e.g., from scripts)
- ☐ Build after other projects are built
- ☐ Build periodically
- ☐ Build pull requests to the repository
- ☐ Build when a change is pushed to GitHub
- ☐ Build when another project is promoted
- ☑ Monitor Docker Hub for image changes
  - ☑ Any referenced Docker image can trigger this job
  - ☐ Specified repositories will trigger this job

Figure 6.9. Configuring a Docker Hub trigger

Once a job has been triggered, the build's log will state what the trigger was (e.g. "triggered by push to <Docker Hub repo name>"). Triggers can be set by two parameters that have have been created by the plugin - the Docker Hub repository name and Docker Hub host.

Docker Hub pipelines. Docker Hub itself supports webhook chains, which you can read more about in Docker's webhook documentation[14]. If you have added several webhooks for different operations, the callback that each service is doing is done in a chain. If one hook fails higher up in the chain, then any following webhooks will not be run. This can be useful if using the downstream Jenkins job as a QA check before performing any other operations based on the pushed image.

<jenkins-url>/jenkins/dockerhub-webhook/details will list all events triggered by all hook events, and you will be linked directly to the build, while Docker Hub's webhook will link back to the Jenkins instance. You can also push tags to the Docker Hub repository, but build status is not pulled by default. The Docker Hub webhook will show result of builds in the "history" of the image on Docker Hub.

- **Other operations**

This plugin also adds a build step for pulling images from Docker Hub. This is a simple build step that does a "docker pull" using the specified ID, credentials, and registry URL.

6.1.7. Building and publishing Docker images with the CloudBees Docker Build and Publish Plugin[15]

Many organizations struggle with releasing their applications and this struggle has birthed an industry of tools designed to simplify the process. Release management tools allow a release process to be defined as stages in a pipeline, and stages themselves contain sequential steps to be performed before the next begins. Stages are segmented using approval gates to ensure that QA and release managers get the final say on whether are artifact is ready for the next stage in the release pipeline, and the entire process is tracked for reporting purposes.

The goal of such processes is to ensure that only high quality releases are deployed into production and are released on time, and the release manager is responsible for it all.

An obstacle to a smooth release is the structural challenge of maintaining identical testing and production environments. When these environments differ, it allows an opportunity for unexpected regressions to slip through testing and botch a release. Ideally, all environments will be identical and contain the same dependent libraries and tooling for the application, as well as the same network configurations.

The versatility and usability of Docker has made it a popular choice among DevOps-driven organizations. It has also made Docker an ideal choice for creating the

[14] https://docs.docker.com/docker-hub/builds/#webhooks
[15] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Build+and+Publish+plugin

standardized and repeatable environments that an organization needs for both creating identical testing and production environments as well as for packaging portable applications.

If an application is packaged in a Docker image, testing and deploying is a matter of creating a container from that image and running tests against the application inside. If the application passes the tests, then they should be stored in a registry and eventually deployed to production.

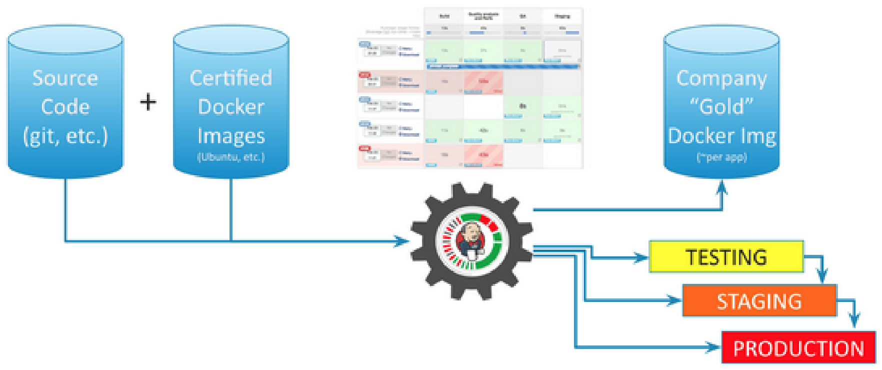

Source Code (git, etc.)
+
Certified Docker Images (Ubuntu, etc.)
Company "Gold" Docker Img (~per app)
TESTING
STAGING
PRODUCTION

Figure 6.10. Automating application releases with Docker

Automating the release. According to Forrester Research, the top pain of release management is a lack of visibility into the release management process and their process' lack of automation.

However, the testing, deploying, and releasing stages of a release pipeline can be orchestrated by Jenkins using the CloudBees Docker Build and Publish plugin. This plugin creates a new build step for building and packaging applications into Docker containers, as well as publishing them a images to both private and public Docker registries like Docker Hub.

Testing and QA. Applications packaged in Docker images can be tested by running them as a container. Docker allows containers to be linked, granting the linked container shell access and allowing it to run scripts against the application's container. This link can also be made between the Docker application container and another container which is packaging a service the application needs to run against for a true integration test, such as a test database.

Promotion and Release. Jenkins supports the concept of promotion, where tested and approved artifacts are promoted to the next stage in a pipeline. Promotion is compatible with both traditional Jenkins jobs and the new Jenkins Workflow, and promotions can be set to only trigger if manually approved by particular users or team members.

Once an artifact is promoted, it can be manually or automatically moved to the next stage of its pipeline, whether that is a pre-production staging area or a registry like

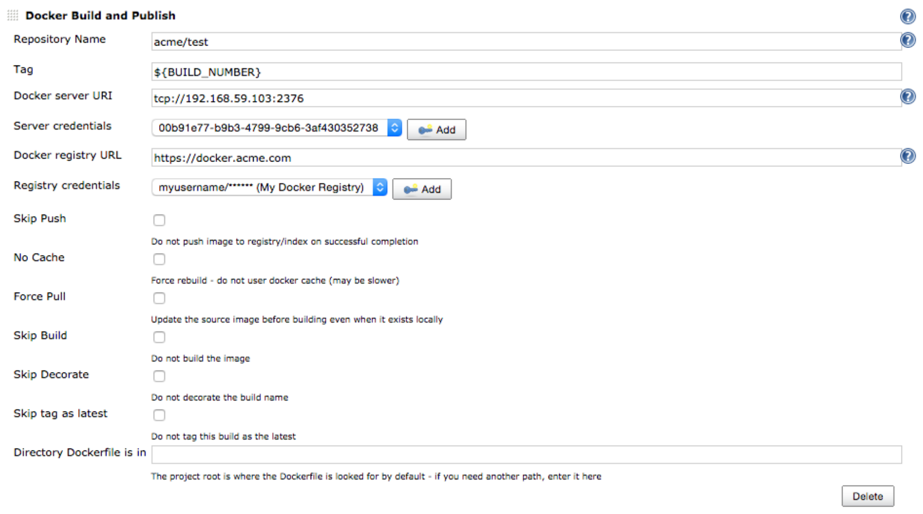
Docker Hub for later use as a dependency for another artifact's build. The promotion can also trigger any other number of pre-release actions, such as notifications and sending data about the artifact to a company dashboard.

Prerequisites. Unlike the CloudBees Docker Hub Notifications plugin, this plugin is only running the build of Docker images and pushing them to a specified Docker registry - no Docker registry webhook setup is required.

To use this plugin, the Docker CLI must be available on the slave machine that will be running the build.

Credentials for the Docker server will also need to be configured to push to a registry, and this can either be configured in a *.dockercfg*[16] file or within the Jenkins job. The target Docker server's URI can also be configured in this file or in the Jenkins job.

Configuration. To use this plugin, users will need to point Jenkins to where the Dockerfile for the target application is located. This can be a Dockerfile that is been checked into the slave's workspace or within the root folder of a project.



Docker Build and Publish

Repository Name	acme/test
Tag	${BUILD_NUMBER}
Docker server URI	tcp://192.168.59.103:2376
Server credentials	00b91e77-b9b3-4799-9cb6-3af430352738
Docker registry URL	https://docker.acme.com
Registry credentials	myusername/****** (My Docker Registry)
Skip Push	☐ Do not push image to registry/index on successful completion
No Cache	☐ Force rebuild - do not user docker cache (may be slower)
Force Pull	☐ Update the source image before building even when it exists locally
Skip Build	☐ Do not build the image
Skip Decorate	☐ Do not decorate the build name
Skip tag as latest	☐ Do not tag this build as the latest
Directory Dockerfile is in	The project root is where the Dockerfile is looked for by default - if you need another path, enter it here

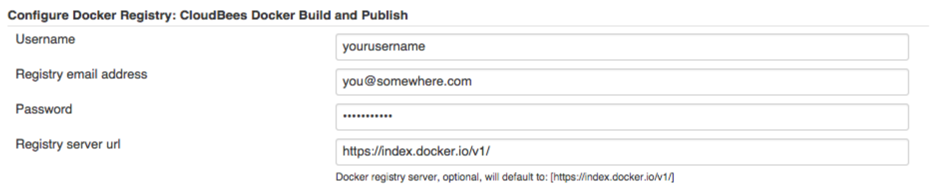
Figure 6.11. Configuring the Docker Build and Publish plugin

This plugin provides the following configuration options for the build/publish build step:

- **Repository name** - The name of the image to build, e.g. "csanchez/test"
- **Tag** - Whatever tag the build should take. This field supports using environment variables, such as the ${BUILD_NUMBER}, to tag the image. — **Docker server URI** - If left blank, this plugin will rely on the installed Docker server's the environment variables. The credentials for the Docker server will be stored in

[16] https://coreos.com/docs/launching-containers/building/customizing-docker

Jenkins as a Docker Server Certificate Authentication type, provided by the Docker Commons Plugin and the Credentials Plugin.



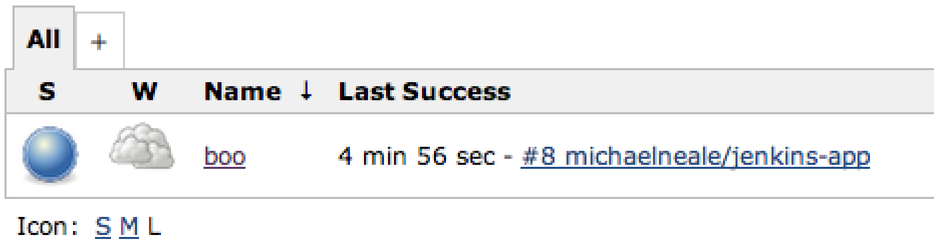
Configure Docker Registry: CloudBees Docker Build and Publish

Username	yourusername
Registry email address	you@somewhere.com
Password	•••••••••••
Registry server url	https://index.docker.io/v1/

Docker registry server, optional, will default to: [https://index.docker.io/v1/]

Figure 6.12. Configuring credentials for the Docker Build and Publish plugin

- **Server credentials** - Credentials needed for connecting to the Docker server. If no credentials are entered, Jenkins will default to using whatever is configured in the - Docker environment variables for your slave machine’s Docker CLI.
- **Registry credentials** - Typical credentials/user password.
- **Skip push** - Opt to just build, not push to specified Docker registry. A use case for this is to build an image locally for testing purposes before pushing it.
- **No cache** - An option to discard the old cache for this image completely during the build and do the build from scratch.
- **Force Pull** - An option to pull all dependent images, even if they are cached locally.
- **Skip Decorate** - An option to not show the repository name and tag in the Jenkins build history.



S	W	Name ↓	Last Success
●	☁	boo	4 min 56 sec - #8 michaelneale/jenkins-app

Figure 6.13. Image build decorations with the Docker Build and Publish plugin

- **Skip tag as latest** - By default, every time an image is built, it gets the default latest tag.

Tips and Tricks. **Explicit v. Default Docker Configurations**

If you have multiple slaves and each have Docker installed, you should not set credentials for each job, but would instead leave the configuration options blank so that the job's builds rely on the Docker CLI to be correctly configured on each slave machine. One set of credentials would not work for all of the build/publish jobs' steps unless explicitly configured.

OSX/Windows Docker Compatibility

If a server or registry is not configured, it will use the default of the Docker command line. This means that the Docker command line must be installed to the machine that will run this build.

This plugin can support boot2docker on OSX/Windows as long as Boot2Docker is up and the Docker Server URI for the OSX/Windows machine points to some Docker server or the Linux container running Docker. The Docker Server Certificate Authentication needs to be configured as a credential in Jenkins, and for boot2docker these keys/certificates are generated by boot2docker itself.

6.1.8. Orchestrating Workflows with Jenkins and Docker with the CloudBees Workflow Docker Plugin[17]

Most real world pipelines are more complex than the canonical BUILD#TEST#STAGE#PRODUCTION flow. These pipelines often have stages which should not be triggered unless certain conditions are met, while other stages should trigger if the conditions are not.

Jenkins Workflow[18] helps writes these pipelines, and the Jenkins Workflow Docker plugin extends on it to provide first class support for Docker as well. The CloudBees Workflow Docker plugin offers a domain-specific language (DSL) for performing some of the most commonly needed Docker operations in a continuous-deployment pipeline within a workflow script.

Prerequisites. This plugin is dependent on the Jenkins Workflow plugin and requires installation of the Docker CLI to nodes performing any Docker-related steps.

Docker Workflow DSL Keywords. Workflows are defined as scripts that tell Jenkins what to do within the workflow, what tools to use, and which slave to use for the builds. These actions are pre-defined as steps and new steps can be created using plugins. Below are some of the key Docker-related steps for a workflow:

`docker`: Docker related steps

- This variable allows access to Docker-related functions for use in the Workflow script.
- Params:
- Global variable

[17] https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Workflow+Plugin
[18] http://jenkins-cookbook.cloudbees.com/docs/jenkins-cookbook/_continuous_delivery_with_jenkins_workflow.html

```
docker.node() {}
```

`inside`: Allocate container

- This step schedules the tasks in its nested block on whatever Docker container was defined in the image that this command is being run from (e.g. “maven32.inside(){}” means that all commands in the curly brace will be run inside the maven32 container).

- Params:

- `label`: String - name of a Docker image to be used as a container

- Nested block

```
docker.image(‘maven:3.3.3-jdk-8’).inside {}
```

`push`: Publish to registry

- This step publishes to a registry whatever Docker image this command is being run from (e.g. “newApp.push()” will publish newApp’s image to a registry).

```
docker.image(‘maven:3.3.3-jdk-8’).push()
```

`withRegistry`: Interact with a registry

- This step selects a Docker registry which all tasks in the nested block will run against.

- Params:

- `url`: String - URL of Docker registry

- `credentials`: String -

- Nested block

```
docker.withRegistry(‘https://docker.mycorp.com/', ‘docker-login) { }
```

More information on syntax and advanced usages can be found in the CloudBees Docker Workflow documentation[19]. A Docker-based demonstration environment can also be found here[20].

scaledwidth=90%:imagesdir: ../resources/

[19] http://jenkins-enterprise.cloudbees.com/docs/user-guide-docs/docker-workflow.html
[20] https://github.com/jenkinsci/docker-workflow-plugin/blob/master/demo/README.md

Chapter 7. CloudBees Services

CloudBees is a company which offers a Jenkins-as-a-service, proprietary enhancements, and support for Jenkins and open-source plugins. These enhancements are in the form of plugins and a separate, Jenkins-based software for managing Jenkins masters.

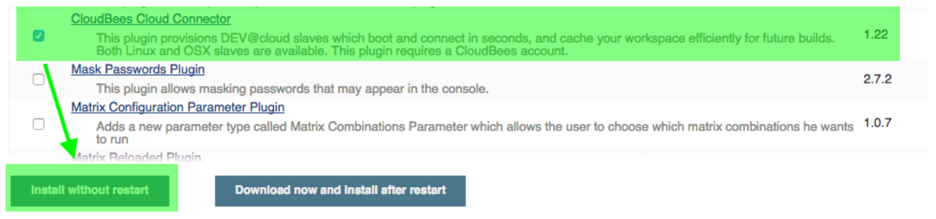
7.1. Cloud Slaves with CloudBees Cloud Connector

As part of CloudBees’ Jenkins-as-a-service, known as DEV@cloud, open source Jenkins users can offload certain build jobs to CloudBees’ slave pool of Linux build machines.

This is possible using the CloudBees Cloud Connector Plugin, which connects to your CloudBees DEV@cloud account and uses DEV@cloud slaves to build jobs. This plugin is available in the open-source update center.

7.1.1. Installing CloudBees Cloud Connector

In the "Manage Jenkins > Plugins > Install" windows, check the plugin "CloudBees Cloud Connector" and then click on "Install without Restart".



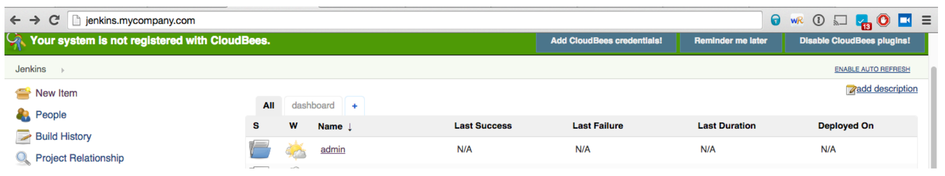
✓	CloudBees Cloud Connector This plugin provisions DEV@cloud slaves which boot and connect in seconds, and cache your workspace efficiently for future builds. Both Linux and OSX slaves are available. This plugin requires a CloudBees account.	1.22
☐	Mask Passwords Plugin This plugin allows masking passwords that may appear in the console.	2.7.2
☐	Matrix Configuration Parameter Plugin Adds a new parameter type called Matrix Combinations Parameter which allows the user to choose which matrix combinations he wants to run	1.0.7
	Matrix Reloaded Plugin	

Figure 7.1. Cloud Connector plugin installation

7.1.2. Configuring CloudBees Cloud Connector

After installing the plugin, a green banner "Your system is not registered with CloudBees" appears on Jenkins screens (if it doesn’t appear, restart Jenkins).

Click on "Add CloudBees Credentials", then select "Add Credentials" and "CloudBees User Account" and enter your credentials.



jenkins.mycompany.com

Your system is not registered with CloudBees.

S	W	Name ↓	Last Success	Last Failure	Last Duration	Deployed On
		admin	N/A	N/A	N/A	N/A

Figure 7.2. Adding CloudBees Credentials, #1

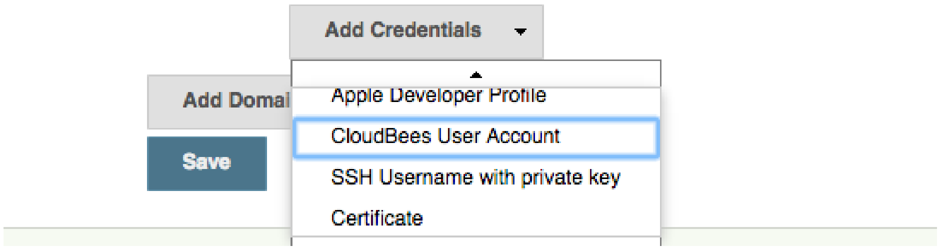
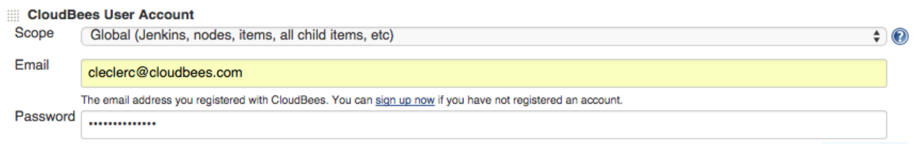


Figure 7.3. Adding CloudBees Credentials, #2



CloudBees User Account

Scope: Global (Jenkins, nodes, items, all child items, etc)

Email: cleclerc@cloudbees.com

The email address you registered with CloudBees. You can sign up now if you have not registered an account.

Password: ••••••••••••••

Figure 7.4. Adding CloudBees Credentials, #3

Go to the Manage Manage "Jenkins Jenkins > Configure System" screen.

In the "Cloud" section at the bottom of the screen, click on "Add a new Cloud" and select "CloudBees DEV@cloud Slaves".

In the Account drop down list, select the DEV@cloud account that you want to use if you have several accounts.

Then click on "Save". You can now use DEV@cloud slaves to build your applications.

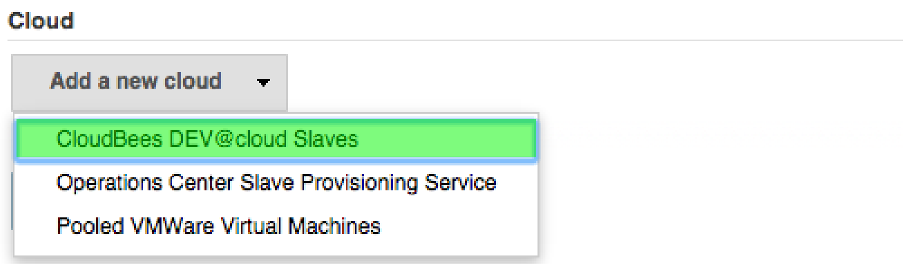
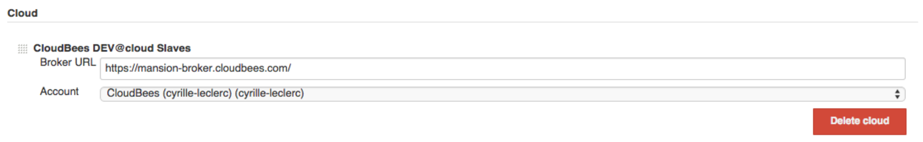

Cloud
Add a new cloud
CloudBees DEV@cloud Slaves
Operations Center Slave Provisioning Service
Pooled VMWare Virtual Machines

Figure 7.5. Connecting a DEV@cloud account, #1



Cloud

CloudBees DEV@cloud Slaves

Broker URL: https://mansion-broker.cloudbees.com/

Account: CloudBees (cyrille-leclerc) (cyrille-leclerc)

Delete cloud

Figure 7.6. Connecting a DEV@cloud account, #2

DEV@cloud slave templates. The "DEV@cloud slave templates" configuration screen lists the types of slaves available on the DEV@cloud platform (Linux Fedora slaves, OSX slaves…).

Each template is identified by a Jenkins Slave label, you can choose to use a template specifying this label (see below).

Click on the "DEV@cloud slave templates" menu on the left pane of Jenkins home page to display the list of available templates.

You can then display the details of each template. The screenshot below shows that the "DEV@cloud Standard Linux" is identified by the Jenkins label "lxc-fedora17".

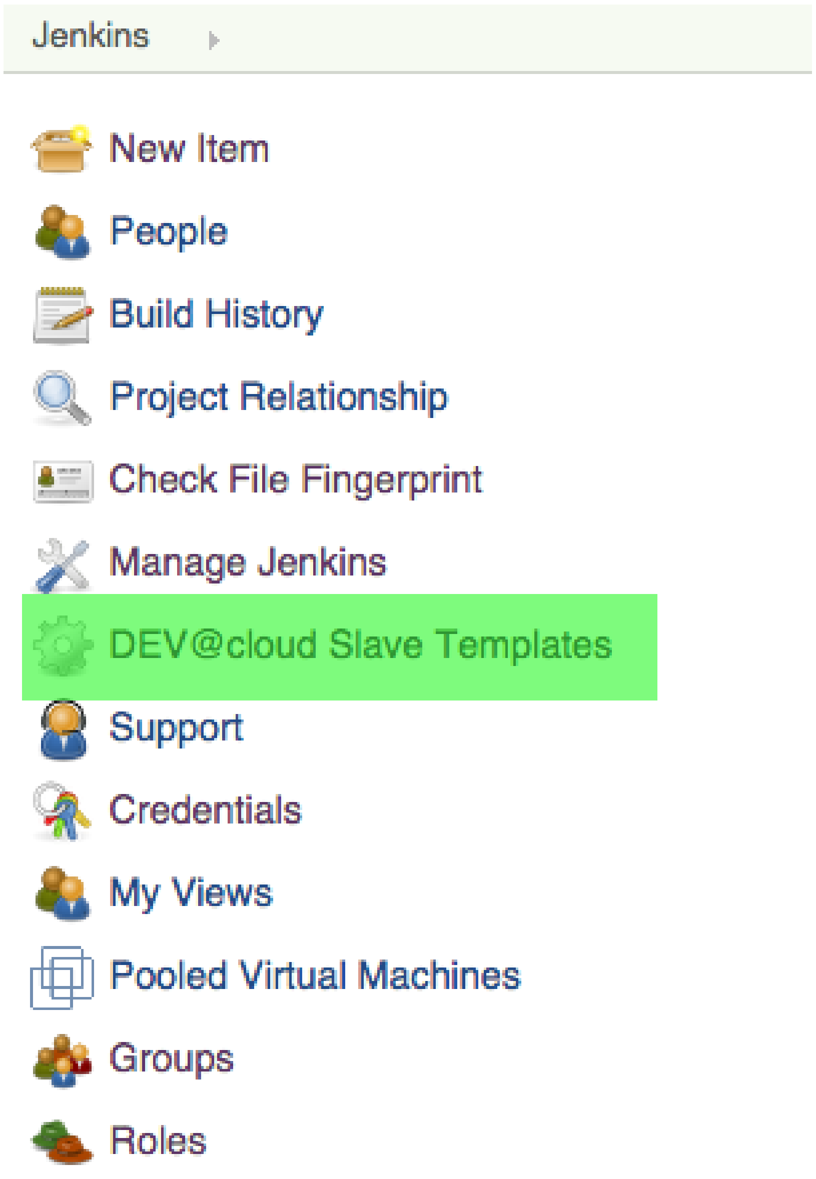
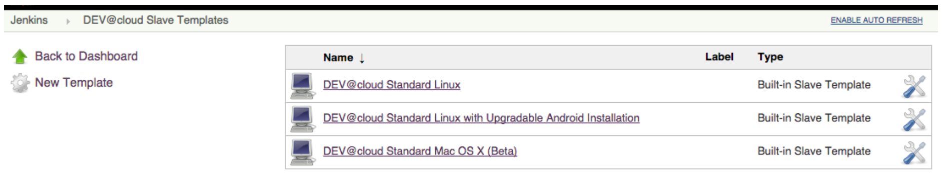


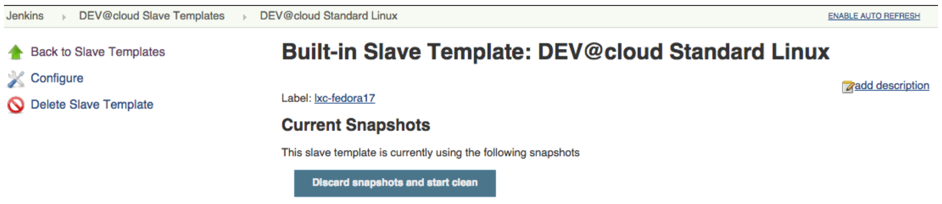
Figure 7.7. Utilizing DEV@cloud’s labeled slaves, #1


Jenkins › DEV@cloud Slave Templates
ENABLE AUTO REFRESH

Back to Dashboard
New Template

Name ↓	Label	Type
DEV@cloud Standard Linux		Built-in Slave Template
DEV@cloud Standard Linux with Upgradable Android Installation		Built-in Slave Template
DEV@cloud Standard Mac OS X (Beta)		Built-in Slave Template

Figure 7.8. Utilizing DEV@cloud's labeled slaves, #2


Jenkins › DEV@cloud Slave Templates › DEV@cloud Standard Linux
ENABLE AUTO REFRESH

Back to Slave Templates
Configure
Delete Slave Template

Built-in Slave Template: DEV@cloud Standard Linux

add description

Label: lxc-fedora17

Current Snapshots

This slave template is currently using the following snapshots

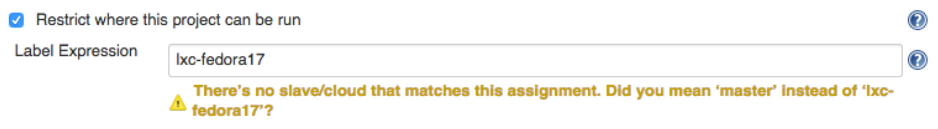
Discard snapshots and start clean

Figure 7.9. Utilizing DEV@cloud's labeled slaves, #3

7.1.3. Using DEV@cloud slaves to build

There are two ways to use CloudBees DEV@cloud slaves.

You can either choose to build on a given CloudBees DEV@cloud slave specifying its label in the "Restrict where this project can be run" section of the job.


☑ Restrict where this project can be run

Label Expression: lxc-fedora17

There's no slave/cloud that matches this assignment. Did you mean 'master' instead of 'lxc-fedora17'?

Figure 7.10. Restricting jobs to certain slaves

The alternate approach to use CloudBees DEV@cloud slaves is to use them as offload capacity when your slaves are busy. Jenkins will transparently offload to DEV@cloud slaves if all your local slaves are busy as long as the label restriction is compatible.

7.2. Sharing slaves on VMware vSphere

7.2.1. Description

The VMWare Pool Auto-Scaling Plugin connects to one or more VMware vSphere installations and uses virtual machines on those installations for better resource utilization when building jobs.

Virtual machines on a vSphere installation are grouped into named pools. A virtual machine may be acquired from a pool when a job needs to be built on that machine, or when that machine is assigned to a build of a job for additional build resources.

When such a build has completed, the acquired machine(s) are released back to the pool for use by the same or other jobs. When a machine is acquired it may be powered on, and when a machine is released it may be powered off.

7.2.2. Configuring VMware vSphere

The first step is to configure in Jenkins the configuration to access to the VMware vCenter(s) that will provide virtual machines.

On the left menu, select "Pooled Virtual Machines"

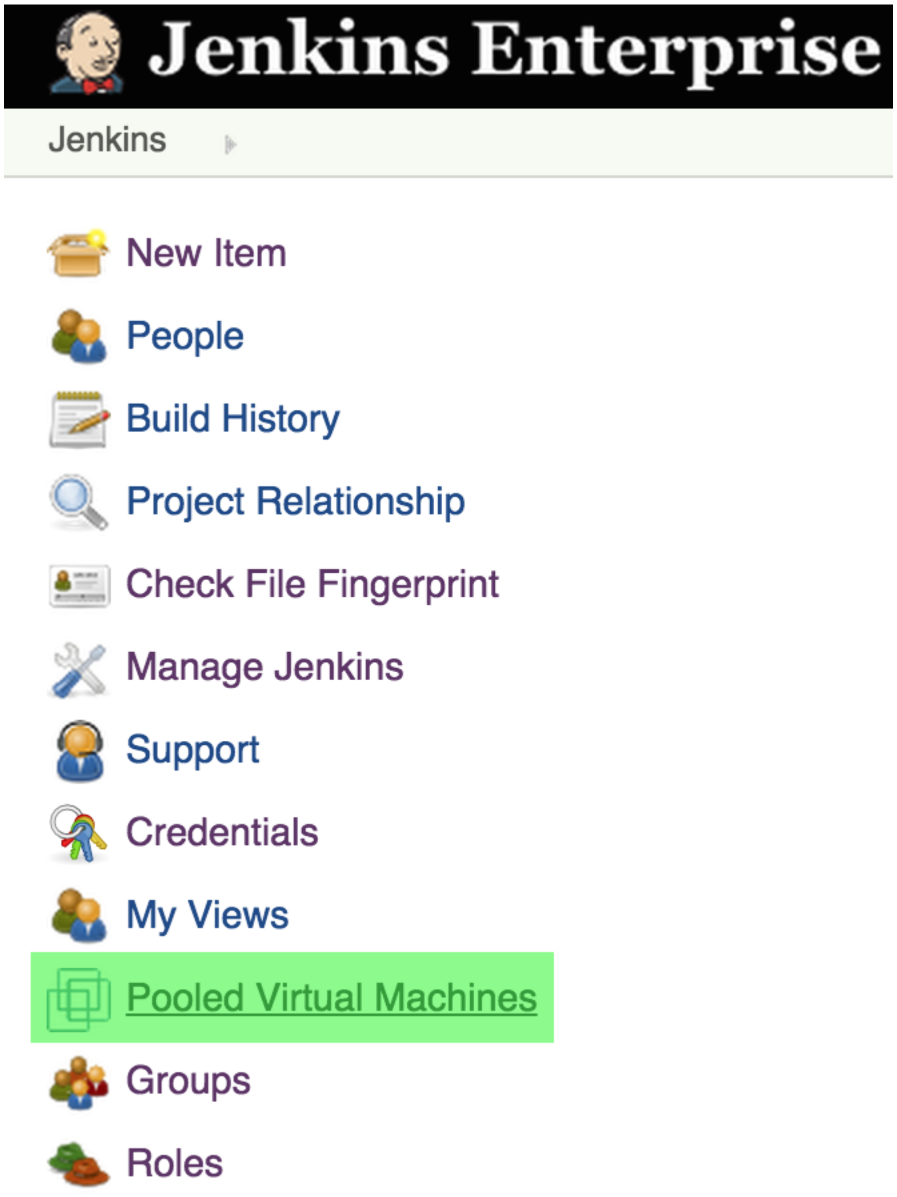
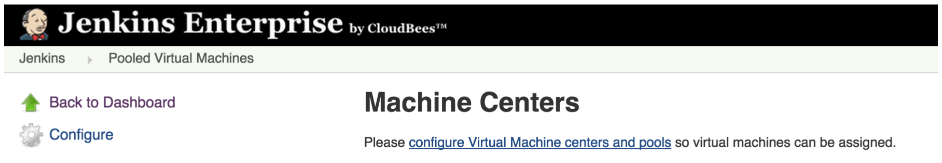


Figure 7.11. Viewing Pooled Virtual Machines

Then click on "Configure"

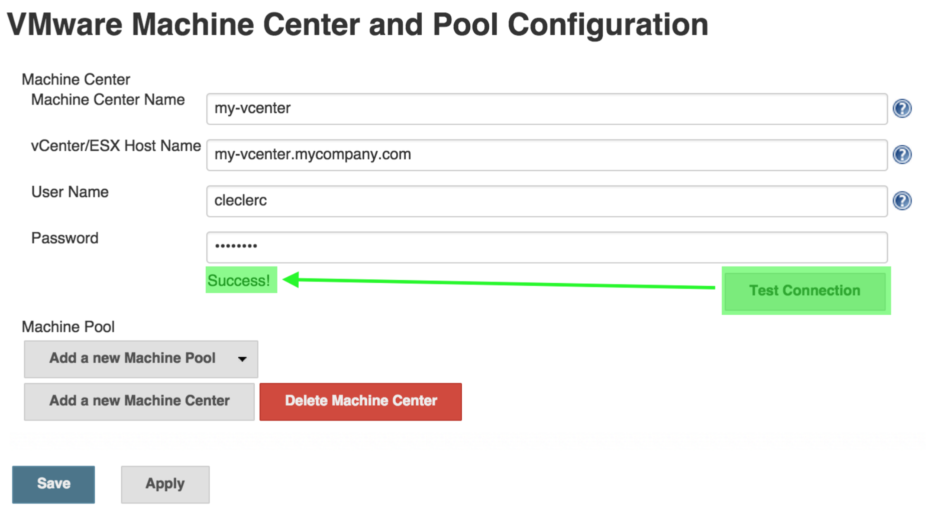


Machine Centers

Please configure Virtual Machine centers and pools so virtual machines can be assigned.

Figure 7.12. Configuring Pooled Virtual Machines

You can then enter the connection details of the vCenter and verify them clicking on "Test Connection"



VMware Machine Center and Pool Configuration

Machine Center

Machine Center Name	my-vcenter
vCenter/ESX Host Name	my-vcenter.mycompany.com
User Name	cleclerc
Password	••••••••

Success!

Figure 7.13. Testing vCenter VM connections

You can now define pools of virtual machines that will be used during Jenkins builds as slaves or as ephemeral servers used by builds.

Static Pool

A static pool is a pool of virtual machines enumerated one by one.

Click on "Add a new Machine Pool" then select "Static Machine Pool".

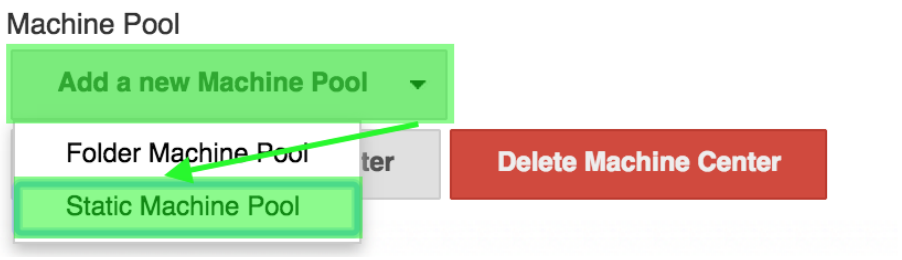
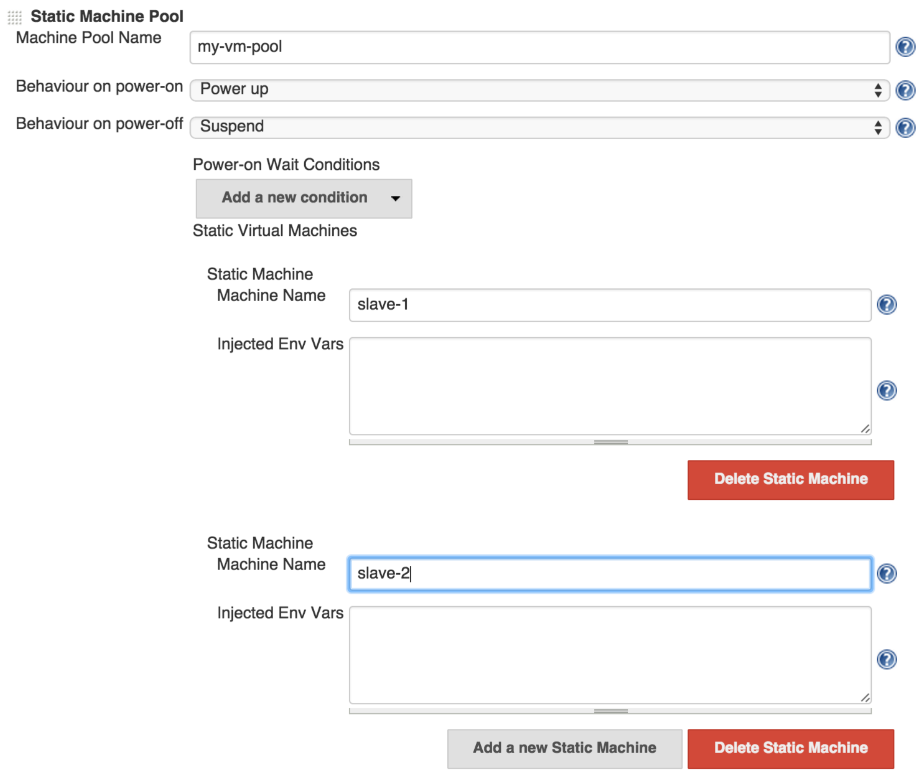


Figure 7.14. Adding a new vCenter pool

You can now enumerate all the machines of your pool. The "Machine Name" must match the name of a virtual machine defined in vCenter.



Static Machine Pool

Machine Pool Name: my-vm-pool

Behaviour on power-on: Power up

Behaviour on power-off: Suspend

Power-on Wait Conditions

Static Virtual Machines

Static Machine

Machine Name: slave-1

Injected Env Vars:

Static Machine

Machine Name: slave-2

Injected Env Vars:

Figure 7.15. Machine Names

For each declared virtual machine, you can define environment variables than can help to describe the capabilities of the VM.

"Behaviour on power-on" and "Behaviour on power-off" will be described later in this chapter.

Folder Pool

A "Folder Pool" is more flexible than a "Static Pool" as it allows you to specify a vCenter folder path or a vApp name and Jenkins will use all the VMs of the folder / vApp.

Click on "Add a new Machine Pool" then select "Folder Machine Pool".

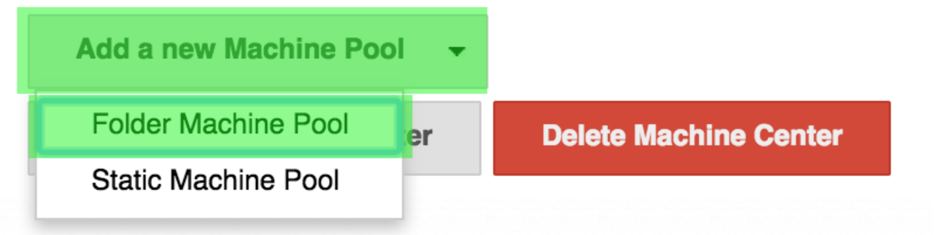
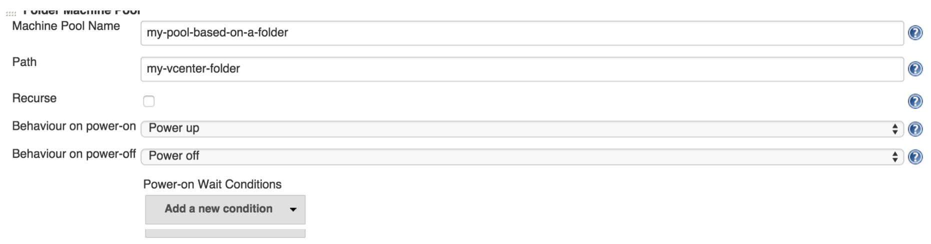


Figure 7.16. Adding a new folder pool

You can now enter the name of the folder of the vCenter Data Center or the name of the vApp.



Folder Machine Pool

Machine Pool Name	my-pool-based-on-a-folder
Path	my-vcenter-folder
Recurse	☐
Behaviour on power-on	Power up
Behaviour on power-off	Power off

Power-on Wait Conditions

Figure 7.17. Naming vCenter folder

You can decide to recurse in the sub folders selecting the "Recurse" checkbox.

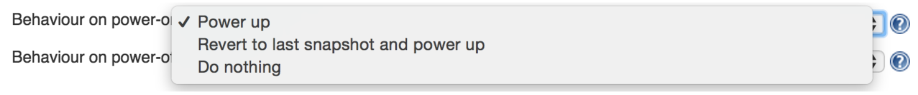
7.2.3. Power-on and power-off strategies

Power-on and power-off strategies let you define the action performed on the vSphere VMs when a VM is borrowed by Jenkins and then when it is returned to the pool.

Power strategies allows you to reset the build environment and/or to optimize resource utilization releasing CPU and RAM when the build slaves are not used.

Power-on. The following power-on strategies are available:

- Power up: starts the VM if it was suspended or powered-off, do nothing if the VM was already running.
- Revert to the last snapshot and power up: revert the state of the VM to the last snapshot and power-on the VM.
- Do nothing: do nothing, this implies that the VM is already running.



Behaviour on power-o

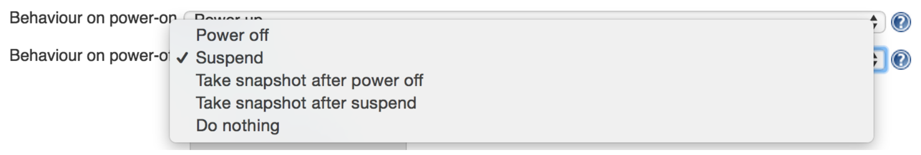
Behaviour on power-o

Figure 7.18. Configuring vSphere power strategy

Power-off. The following power-off strategies are available:

- Power off: shutdown the vSphere VM.
- Suspend: shutdown the vSphere VM.
- Take a snapshot after power off: take a snapshot of the state of the VM after powered off.

- Take a snapshot after suspend: take a snapshot of the state of the VM after suspended.
- Do nothing: keep the VM running.

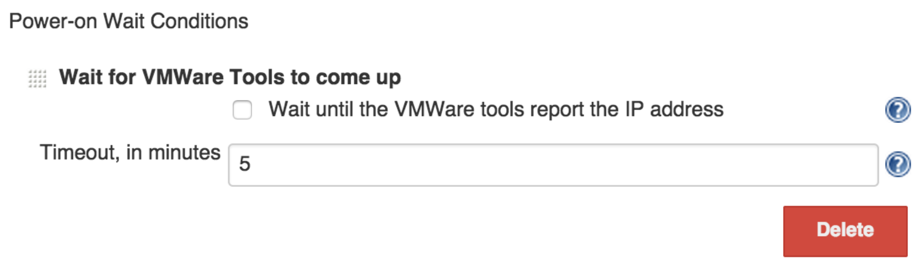

Behaviour on power-on
Behaviour on power-off

- Power off
- ✓ Suspend
- Take snapshot after power off
- Take snapshot after suspend
- Do nothing

Figure 7.19. Configuring vSphere power strategy, #2

Power-on Wait Conditions. Wait conditions allows to ensure that the slave is properly started before triggering builds on it.

Wait for VMWare Tools to come up. Wait for the VMWare Tools service installed on the guest to be started. You can optionally wait for the VMWare Tools to report the IP address of the guest.


Power-on Wait Conditions

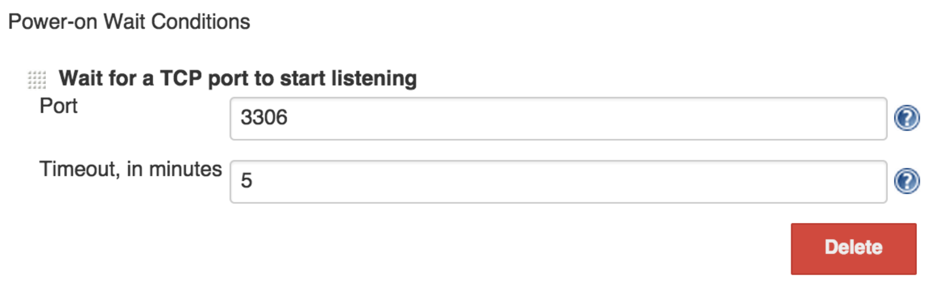
Wait for VMWare Tools to come up

☐ Wait until the VMWare tools report the IP address

Timeout, in minutes: 5

Figure 7.20. Configuring vSphere wait strategy, #3

Wait for a TCP port to start listening. Wait for the guest to start listening on a given TCP port such as the SSH port or the listen port of a database server (e.g. 3306 with MySQL).


Power-on Wait Conditions

Wait for a TCP port to start listening

Port: 3306

Timeout, in minutes: 5

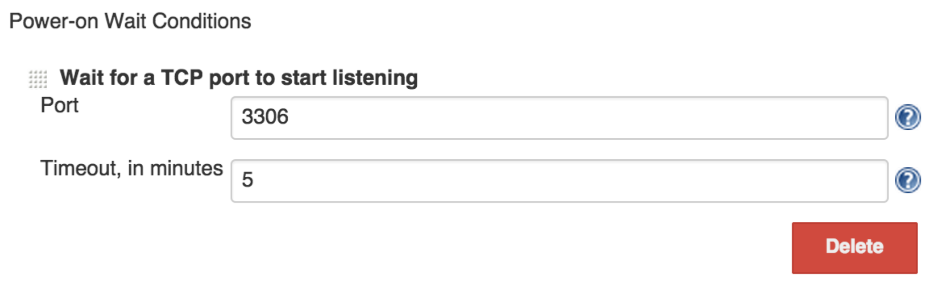
Figure 7.21. Configuring vSphere wait strategy, #4

7.2.4. Auto scaling slaves with VMware vSphere

Once the pool(s) of vSphere virtual machines are declared, you can add a new cloud of slaves in Jenkins configuration.

Go to "Manage Jenkins / Configure System" then in the "Cloud" section at the bottom of the page, add a cloud of type "Pooled VMWare Virtual Machines".

The configuration of the slaves of a cloud is very similar to the configuration of "Dumb Slaves" (labels, remote FS root, connector, etc).


Power-on Wait Conditions

Wait for a TCP port to start listening

Port: 3306

Timeout, in minutes: 5

Figure 7.22. Configuring vSphere autoscale strategy

7.2.5. Using VMWare vSphere slaves to build

VMWare vSphere slaves are similar to standard to other slaves, you can use them restricting a job to a label that is specific to this pool of slaves or using the usage strategy "use this node as much as possible".

7.2.6. Managing VMs of a Pool of VMWare Virtual Machines

You can take offline a VM of a pool of VMWare virtual machines.

On the left menu, select "Pooled Virtual Machines":

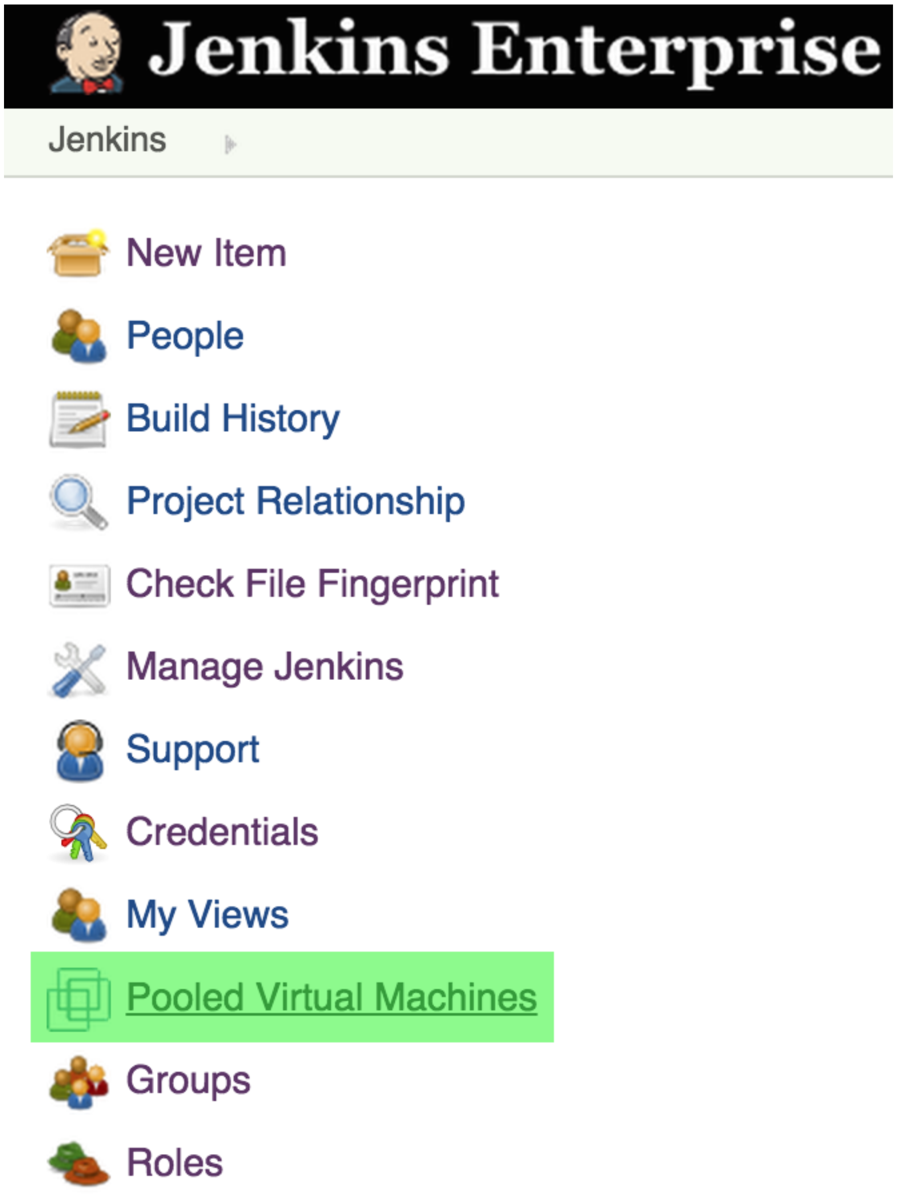
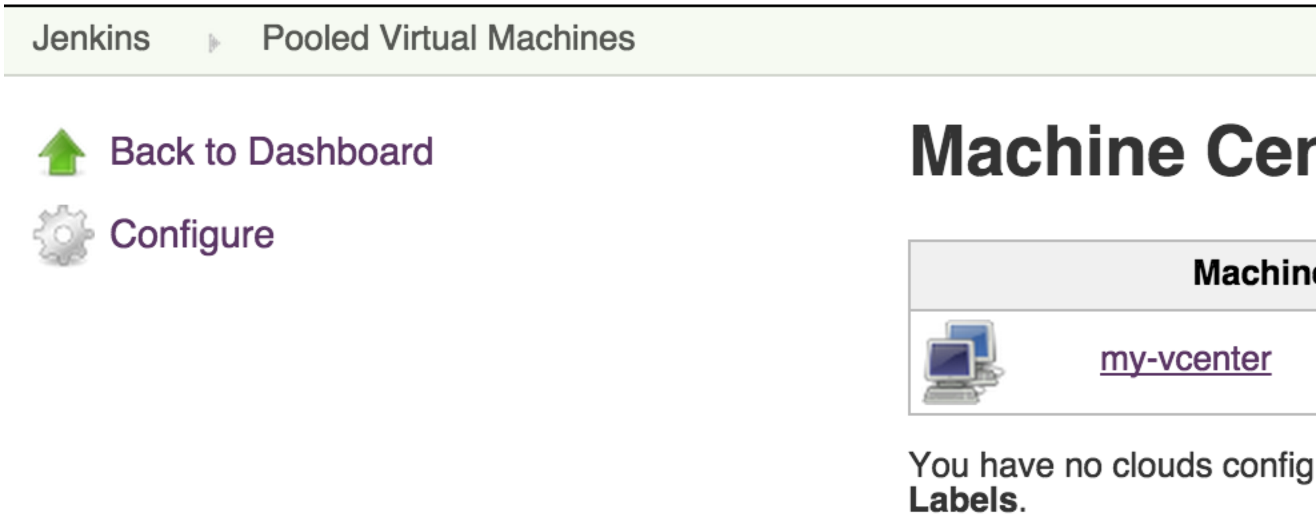


Figure 7.23. Managing vSphere machines

Then click on the desired "Machine Center" (e.g. "my-vcenter"):



Jenkins › Pooled Virtual Machines

Back to Dashboard

Configure

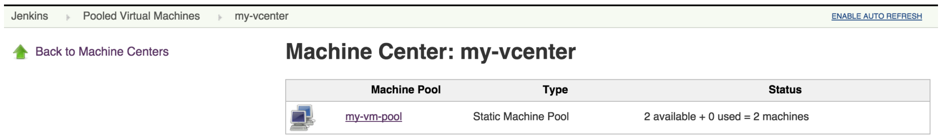
Machine Cer

Machin
my-vcenter

You have no clouds config
Labels.

Figure 7.24. Managing vSphere machines

Then click on the desired "Machine Pool" (e.g. "my-vm-pool"):



Jenkins › Pooled Virtual Machines › my-vcenter — ENABLE AUTO REFRESH

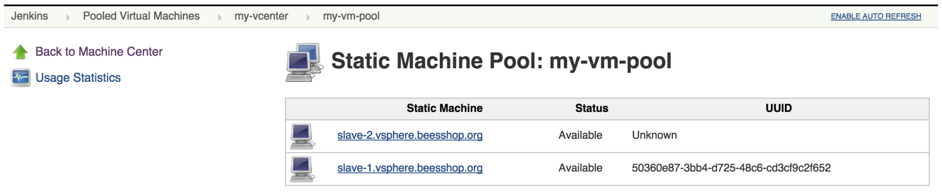
Back to Machine Centers

Machine Center: my-vcenter

Machine Pool	Type	Status
my-vm-pool	Static Machine Pool	2 available + 0 used = 2 machines

Figure 7.25. Managing vSphere machines, #2

The status screen list the virtual machines indicating their status and their vSphere UUID:



Jenkins › Pooled Virtual Machines › my-vcenter › my-vm-pool — ENABLE AUTO REFRESH

Back to Machine Center

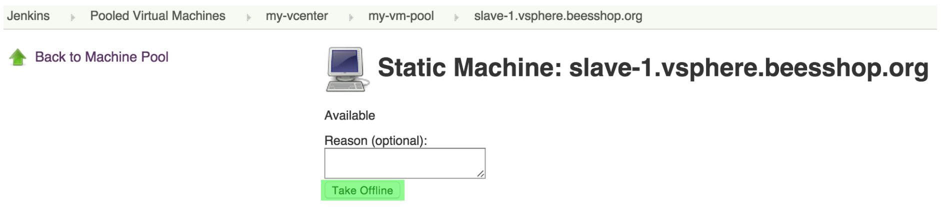
Usage Statistics

Static Machine Pool: my-vm-pool

Static Machine	Status	UUID
slave-2.vsphere.beesshop.org	Available	Unknown
slave-1.vsphere.beesshop.org	Available	50360e87-3bb4-d725-48c6-cd3cf9c2f652

Figure 7.26. Managing vSphere machines, #3

Click on the desired Virtual Machine and you can take if off-line:



Jenkins › Pooled Virtual Machines › my-vcenter › my-vm-pool › slave-1.vsphere.beesshop.org

Back to Machine Pool

Static Machine: slave-1.vsphere.beesshop.org

Available

Reason (optional):

Take Offline

Figure 7.27. Managing vSphere machines, #4

7.2.7. Monitoring the status of a vSphere Pool of virtual machines

You can monitor the pools of VMWare virtual machines. On the left menu, select "Pooled Virtual Machines":

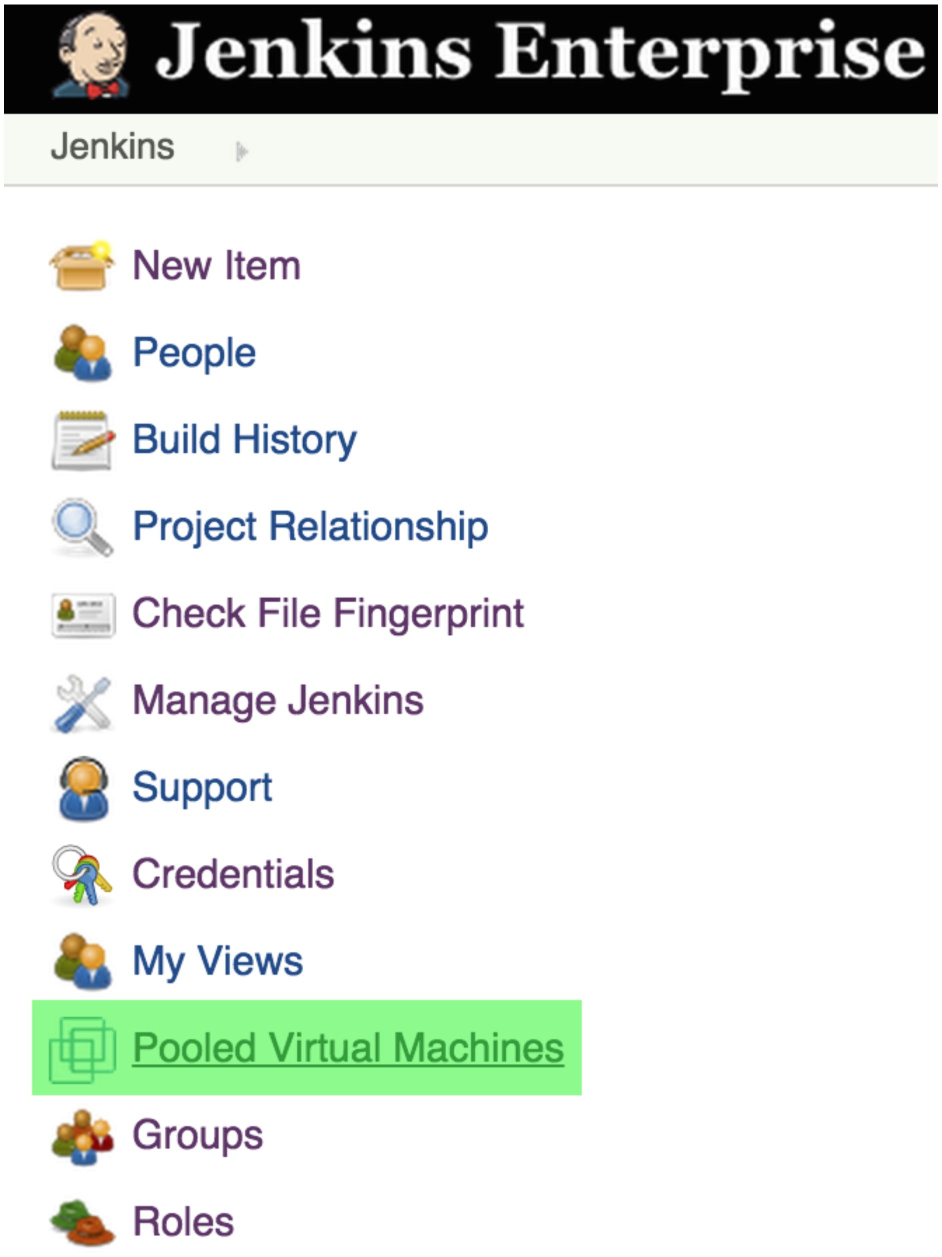
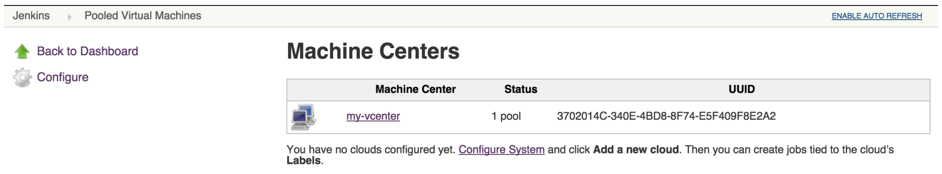


Figure 7.28. Monitoring vSphere machines

Then click on the desired "Machine Center" (e.g. "my-vcenter"):



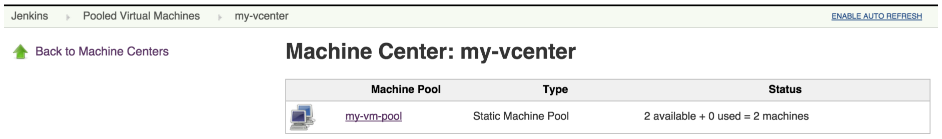
Machine Centers

Machine Center	Status	UUID
my-vcenter	1 pool	3702014C-340E-4BD8-8F74-E5F409F8E2A2

You have no clouds configured yet. Configure System and click **Add a new cloud**. Then you can create jobs tied to the cloud's **Labels**.

Figure 7.29. Monitoring vSphere machines, #2

Then click on the desired "Machine Pool" (e.g. "my-vm-pool"):

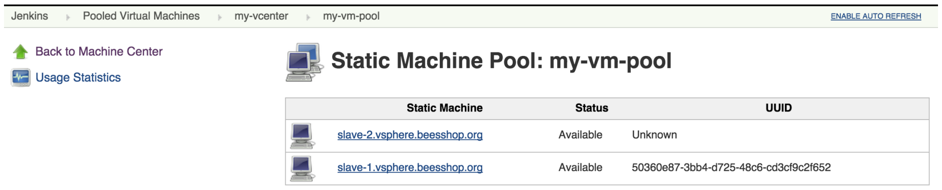


Machine Center: my-vcenter

Machine Pool	Type	Status
my-vm-pool	Static Machine Pool	2 available + 0 used = 2 machines

Figure 7.30. Monitoring vSphere machines, #3

The status screen list the virtual machines indicating their status and their vSphere UUID.

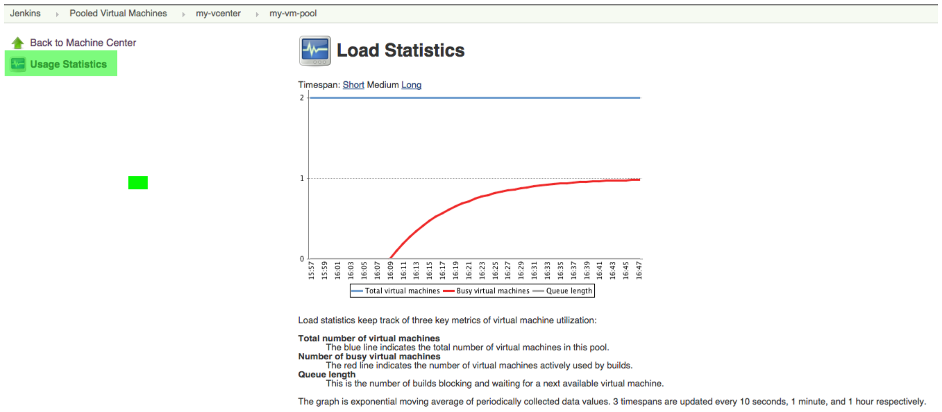


Static Machine Pool: my-vm-pool

Static Machine	Status	UUID
slave-2.vsphere.beesshop.org	Available	Unknown
slave-1.vsphere.beesshop.org	Available	50360e87-3bb4-d725-48c6-cd3cf9c2f652

Figure 7.31. Monitoring vSphere machines, #4

Click on "Usage Statistics" to view monitoring data of the pool.



Load statistics keep track of three key metrics of virtual machine utilization:

Total number of virtual machines
The blue line indicates the total number of virtual machines in this pool.
Number of busy virtual machines
The red line indicates the number of virtual machines actively used by builds.
Queue length
This is the number of builds blocking and waiting for a next available virtual machine.

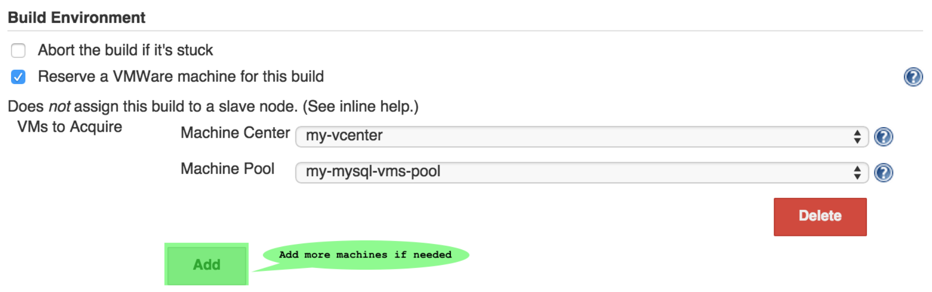
The graph is exponential moving average of periodically collected data values. 3 timespans are updated every 10 seconds, 1 minute, and 1 hour respectively.

Figure 7.32. Monitoring vSphere machines, #5

7.2.8. Using ephemeral servers during job execution with vSphere Virtual Machines

The virtual machines of the VMWare Polled Virtual Machines can also be used as ephemeral server dedicated to builds. You can for example use a temporary database server during a build or test the deployment of an artifact on a server.

In the "Build Environment" of the configuration screen of your job, check "Reserve a VMWare machine for this build". You can then add more virtual machines if needed.


Build Environment

☐ Abort the build if it's stuck

☑ Reserve a VMWare machine for this build

Does *not* assign this build to a slave node. (See inline help.)

VMs to Acquire

Machine Center: my-vcenter

Machine Pool: my-mysql-vms-pool

Figure 7.33. Using ephemeral vSphere machines as slaves

If only one ephemeral virtual machine is allocated, then its IP address is injected in build steps with the environment variable `VMIP`. If more than one ephemeral virtual machines are allocated, then their IP address is injected in build steps with the environment variables `VMIP_1`, `VMIP_2`…

7.3. Sharing slaves with CloudBees Jenkins Operations Center

CloudBees offers a product which allows masters to share build resources called CloudBees Jenkins Operations Center. CJOC provides a Jenkins-based dashboard for managing Jenkins resources, which include two CJOC-specific items: client masters, which are CJOC-managed Jenkins masters, and shared slaves.

Making build resources shareable ensures that they are being used efficiently and can optionally only be brought online when necessary. This allows organizations to free up slave VMs/machines for other uses and it ensures that no slave resource will be wasted. Teams will also be able share scarce specialty resources between masters – e.g. Windows/OSX slave machines.

7.3.1. Creating shared slaves

Unlike typical slaves, shared slaves are treated by CJOC like jobs – dashboard level objects that can be configured, deleted, have role restrictions, and be placed into folders. Despite this “special” classification, a shared slave is like any traditional Jenkins slave machine. This means that to make a machine a shared slave, it is required to:

- Establish at TCP/IP connection to CJOC
- Have Java installed

Additionally:

- Any tools required for builds/tests should be installed to the machine
- Jenkins should have the appropriate permissions over the slaves’ filesystem
- The administrator should know what the remote filesystem root is
- The administrator should also know how many execution slots the slave should have (general 1 or 2 per CPU core)

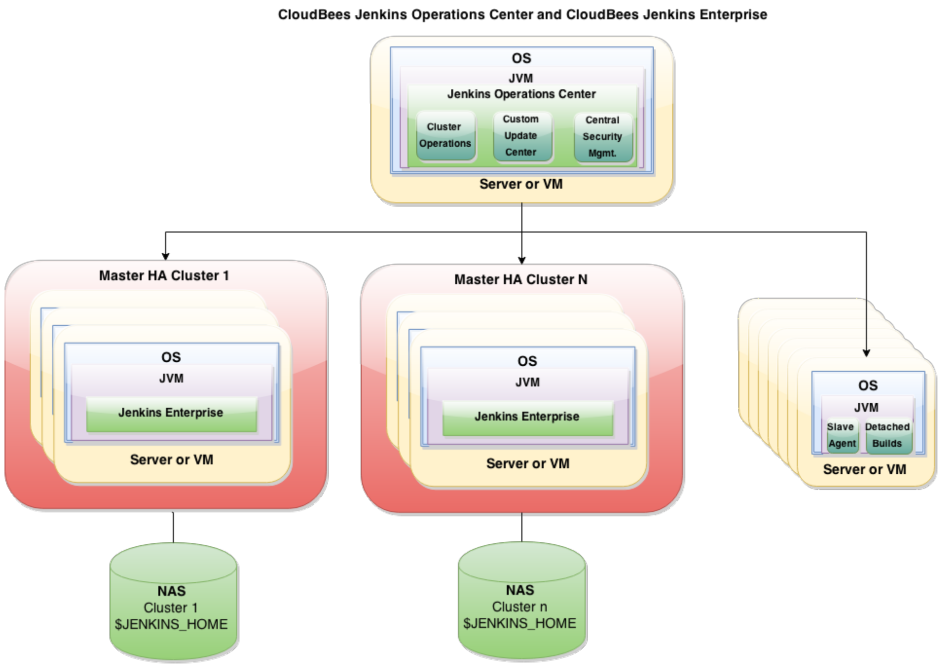
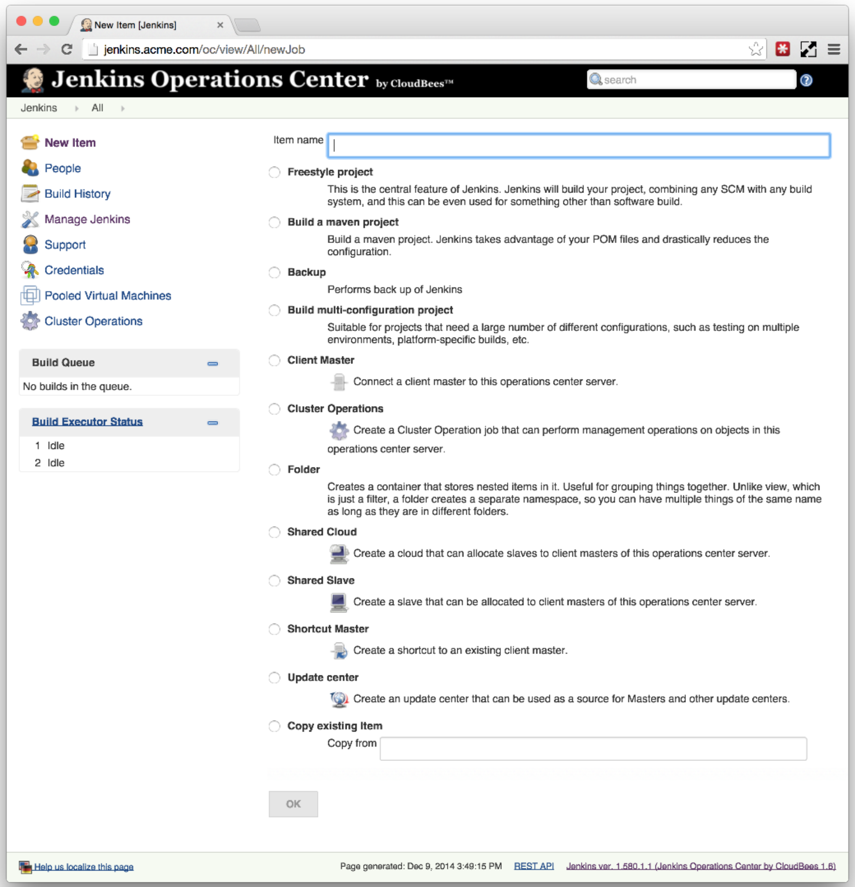


Figure 7.34. Scalable architecture for Jenkins using CJOC and CJE

To create a new shared slave, you will have to create it as a new dashboard-level object of the type “shared slave”.



No builds in the queue.

1 Idle
2 Idle

Item name

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Build a maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Backup
Performs back up of Jenkins

Build multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Client Master
Connect a client master to this operations center server.

Cluster Operations
Create a Cluster Operation job that can perform management operations on objects in this operations center server.

Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

Shared Cloud
Create a cloud that can allocate slaves to client masters of this operations center server.

Shared Slave
Create a slave that can be allocated to client masters of this operations center server.

Shortcut Master
Create a shortcut to an existing client master.

Update center
Create an update center that can be used as a source for Masters and other update centers.

Copy existing Item
Copy from

Page generated: Dec 9, 2014 3:49:15 PM Jenkins ver. 1.580.1.1 (Jenkins Operations Center by CloudBees 1.6)

Figure 7.35. Shared slave creation

You will then be taken to the shared slave configuration screen. This screen is almost identical to a traditional slave's configuration screen, so many of the options will be familiar to an experienced Jenkins administrator.

Shared slaves can be connected to CJOC using the same connection methods as typical Jenkins slaves:

- SSH
- Java Web Start
- Via a command on the master
- As a Windows service

They can also be configured to have availability policies:

- Online as much as possible
- Offline when idle

- Online according to a schedule
- Online when in demand

When the slave configuration is complete save or apply the configuration. If you do not want the slave to be available for immediate use, de-select the “Take on-line after save/apply” checkbox before saving the configuration.

7.3.2. Advanced Usages

Restricted usage. Shared slaves can be restricted to only serving certain jobs by adding labels to the slave on its configuration screen. Labels should be descriptive and one slave can have many labels. For example, a CentOS build slave with an Android SDK installed could be described using the labels “linux && android”.

Once the slave has been assigned a label, you will need to edit the configuration for any jobs which should utilize those labels. Under the initial configurations for any given job is the option to “restrict where this project can be run”. To restrict the configured job to only run on a labeled slave, check the box next to this option and input the name of the labels or boolean expression.

As the size of the cluster grows, it becomes useful not to tie projects to specific slaves, as it hurts resource utilization when slaves may come and go. For such situation, assign labels to slaves to classify their capabilities and characteristics, and specify a boolean expression over those labels to decide where to run.

Here are the valid operators that can be used when working with slave labels:

The following operators are supported, in the order of precedence.

```
(expr)
```

- parenthesis

  ```
  !expr
  ```

- negation

  ```
  expr&&expr
  ```

- and

  ```
  expr||expr
  ```

- or

  ```
  a -> b
  ```

- "implies" operator. Equivalent to !a|b. For example, windows#x64 could be thought of as "if run on a Windows slave, that slave must be 64bit." It still allows Jenkins to run this build on linux.

```
a <-> b
```

- "if and only if" operator. Equivalent to a&&b || !a&&!b. For example, windows<#sfbay could be thought of as "if run on a Windows slave, that slave must be in the SF bay area, but if not on Windows, it must not be in the bay area."

All operators are left-associative (i.e., a#b#c <# (a#b)#c), and an expression can contain whitespace for better readability, as white spaces will be ignored.

Label names or slave names can be quoted if they contain unsafe characters. For example, "jenkins-solaris (Solaris)" || "Windows 2008"

7.4. Backing up with CloudBees Backup Plugin

CloudBees offers a plugin which creates a new job type specifically for creating backups of Jenkins and is an alternative solution to the backup solution suggested in “Setting up a backup policy”[1]. CloudBees offers the Backup Scheduling plugin as a part of its Jenkins Enterprise bundle of proprietary plugins.

This plugin creates a new job type in Jenkins called “Backup”, which has a hard coded build step for taking a back up. This plugin offers the option to configure back ups for all or only some of the following:

- Build records for jobs
- Job configurations
- System configurations - this option also allows you to specify whether you’d like to exclude the master.key file or any other files in your $JENKINS_HOME.

This plugin also offers the option for configuring where the backup will be stored (locally, remote SFTP or WebDAV server), as well as the retention policy and format of the backup (tar.gz or zip).

Like other jobs, back up jobs can be configured to have certain triggers - whether that be other jobs, a remote trigger, periodic builds, or some change to an SCM. This plugin allows users to create backup tasks as jobs. In this way, users can use the familiar interface for scheduling execution of backup and monitor any failure in backup activities.

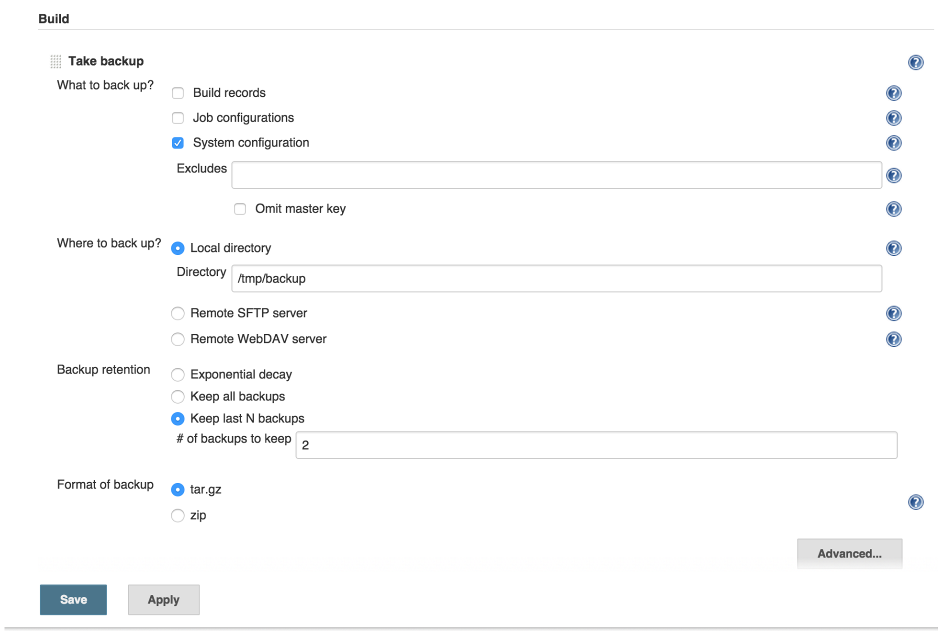
7.4.1. Creating a backup job

To create a backup, click "New Job" from the left and select "Backup Jenkins". This will take you to the page where you can configure the backup project.

[1] http://jenkins-cookbook.cloudbees.com/docs/jenkins-cookbook/_setting_up_a_backup_policy.html

Configuration of a backup job is very similar to that of a freestyle project. In particular, you can use arbitrary build triggers to schedule backup. Once the trigger is configured, click "Add build step" and add "Take backup" builder.

You can add multiple backup build steps to take different subsets of backups at the same time, or you can create multiple backup jobs to schedule different kind of backup execution at different interval.



Build

Take backup

What to back up?
- Build records
- Job configurations
- [✓ System configuration]
- Excludes
- Omit master key

Where to back up?
- [✓ Local directory]
- Directory: /tmp/backup
- Remote SFTP server
- Remote WebDAV server

Backup retention
- Exponential decay
- Keep all backups
- [✓ Keep last N backups]
- # of backups to keep: 2

Format of backup
- [✓ tar.gz]
- zip

Figure 7.36. Backup job configuration options

For example, you might have a daily job that only backs up the system configuration and job configuration, as they are small but more important, then use another job to take the full backup of the system once a week.

7.5. Setting up High Availability with CloudBees High Availability Plugin

The high-availability setup offered by CloudBees Jenkins Enterprise provides the means for multiple JVMs to coordinate and ensure that the Jenkins master is running somewhere. This plugin is a part of the CloudBees Jenkins Enterprise offering, which is a set of plugins combined with support from the CloudBees support team.

A typical HA architecture with this plugin is as follows:

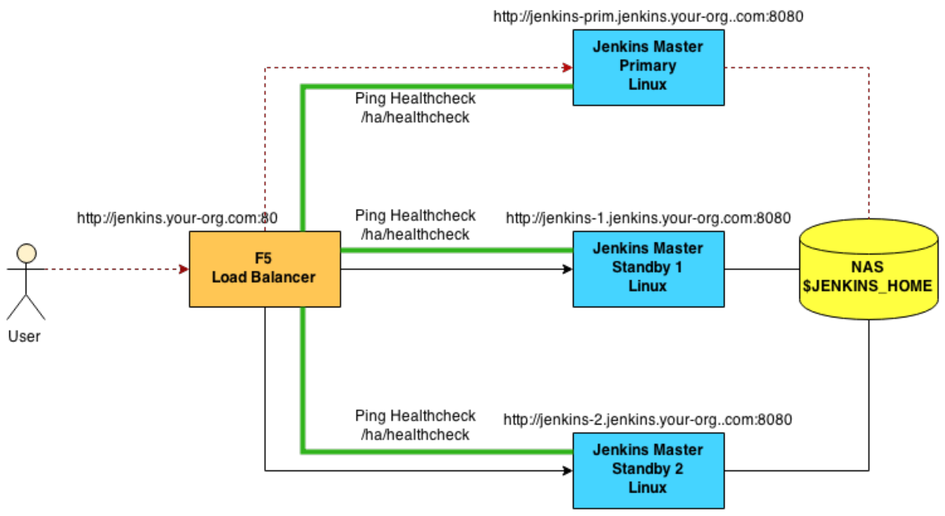

http://jenkins-prim.jenkins.your-org..com:8080
Jenkins Master Primary Linux
Ping Healthcheck /ha/healthcheck
http://jenkins.your-org.com:80
F5 Load Balancer
User
Ping Healthcheck /ha/healthcheck
http://jenkins-1.jenkins.your-org.com:8080
Jenkins Master Standby 1 Linux
NAS $JENKINS_HOME
Ping Healthcheck /ha/healthcheck
http://jenkins-2.jenkins.your-org..com:8080
Jenkins Master Standby 2 Linux

Figure 7.37. High availability architecture using F5 and CJE

Where a load balancer is placed in front of the Jenkins masters and all masters use a NAS for their $JENKINS_HOME. The load balancer then pings all nodes in the cluster via a health check URL and uses their return codes to determine which node is the primary/active master.

The actual details of implementation are highly dependent on what set up is viable within your actual environment.

7.5.1. Setting up an HA architecture

To utilize this feature, Requirements from the environment

Aside from NFS as a storage and IP aliasing as a reverse proxy mechanism, CloudBees Jenkins Enterprise can run on a wide range of environments. In this section, we’ll describe the parameters required from them and discuss more examples of the deployment mode.

Storage. All the member nodes of a Jenkins Enterprise HA cluster need to see a single coherent file system that can be read and written simultaneously. That is to say, for any node A and B in the cluster, if node A creates a file in $JENKINS_HOME, node B needs to be able to see it within a reasonable amount of time. A "reasonable amount of time" here means the time window during which you are willing to lose data in case of a failure.

So long as the same directory is visible on every member node, it need not be mounted on the same path. For example, node A can have *$JENKINS_HOME* at /net/storage/jenkins while node B has *$JENKINS_HOME* at /mnt/jenkins.

$JENKINS_HOME is read intensively during the start-up. If bandwidth to your storage is limited, you will see the most impact in startup performance. Large latency causes a similar issue, but this can be mitigated somewhat by using a higher value in the bootup concurrency by a system property -Djenkins.InitReactorRunner.concurrency=8 (in 1.458 and later).

Builds that run on master use *$JENKINS_HOME* as their work space. Because builds are often I/O intensive, it is recommended to set the number of executors on the master to 0 to avoid doing non-slave builds. If you do allow the master to perform builds, consider limiting the builds to those that are not I/O-bound, or set the "workspace root directory" to be on local storage, not shared storage.

Additional deployment-specific discussions follow.

NFS

When mounting NFS, use the intr mount flag so that you can interrupt the process in case of a failure to access storage. For truly highly-available Jenkins, NFS storage itself needs to be made highly-available. There are many resources on the web describing how to do this. See the Linux HA-NFS Wiki as a starting point.

DRBD

Distributed Replicated Block Device can be used to host $JENKINS_HOME. It provides highly available storage without the need to have a separate storage node. Because of the access requirements, however, DRBD needs to be run in the dual-primary mode, which restricts the choice of file system.

HTTP reverse proxy. When a fail-over happens and the Jenkins master role moves from one node to another, the location of the service moves, and therefore an additional mechanism is needed to shield users from this change. We call this "HTTP reverse proxy".

Jenkins Enterprise instances acting as stand-by nodes respond to all inbound HTTP requests with the 503 "service unavailable" HTTP status code. The front-end can use this as a signal to determine the node to forward the requests to.

The rest of this section discusses various reverse proxy deployment options.

IP aliasing

IP aliasing that we saw in the tutorial is a cheap way to emulate the effect of a highly-available HTTP reverse proxy, but this requires you to have root access to the system, and your machines need to be running in the same subnet. Google "Linux IP aliasing" for more details.

HAproxy

HAproxy is a load balancer that can be used as a reverse proxy. For truly highly-available setup, HAproxy itself also needs to be made highly available. This is a feature built into HAproxy itself, and is separate from the CloudBees High Availability plugin.

Network Normally the nodes in an HA cluster discover one another automatically by means of files in the *$JENKINS_HOME/jgroups/* directory. This works so long as each node can “see” the others’ advertised IP addresses and make TCP connections to them.

Tutorial. In this tutorial, we will describe the simplest HA Jenkins setup, which creates a baseline when for later discussions on other modes of deployment. Here, we deploy Jenkins in the following configuration for high availability:

- An NFS server that hosts $JENKINS_HOME. We assume this is available already, and we will not discuss how one would set this up. (We'll call this machine sierra.)

- Two linux systems that form a Jenkins HA cluster, by running one JVM each (we'll call them *alpha* and *bravo*). In this tutorial, those two machines need to be on the same local network.

- One floating IP address (we'll call this 1.2.3.4). Alpha and bravo take this IP address while each is acting as the primary, thereby ensuring that users can always access Jenkins through the DNS name that's associated with this IP address.

First, install CloudBees Jenkins Enterprise packages to alpha and bravo. You need to install both the Jenkins package for the CloudBees Jenkins Enterprise master itself and the jenkins-ha-monitor package, which is the Jenkins Enterprise by CloudBees HA monitor tool.

Tip

Linux packages for jenkins-ha-monitor are not included in all versions of the Jenkins Enterprise repository. If yours is missing this package, or you are using a non-Linux system, see the section in the CloudBees Jenkins Enterprise plugin documentation[2] called "Jenkins Enterprise HA monitor tool" for information on direct downloads.

Choose the appropriate Debian/Red Hat/openSUSE package format depending on the type of your distribution. Upon installation, both instances of Jenkins will start running. Stop them by issuing /etc/init.d/jenkins stop.

Let us say sierra exports the /jenkins directory that hosts *$JENKINS_HOME*. Log on to alpha and mount this directory.

```
$ mount -t nfs -o rw,hard,intr sierra:/jenkins /var/lib/jenkins
```

/var/lib/jenkins is chosen to match what the Jenkins Enterprise packages use as $JENKINS_HOME. If you change them, update /etc/default/jenkins (on Debian) or /etc/sysconfig/jenkins (on RedHat and SUSE) to have *$JENKINS_HOME* point to the correct directory.

[2] http://jenkins-enterprise.cloudbees.com/docs

To make this mount automatically happen, update your /etc/fstab by adding the following entry:

```
sierra:/jenkins nfs rw,hard,intr 0 2
```

Repeat this mount setup on bravo, and ensure that both alpha and bravo see the same data. (For example, touch a from alpha, and make sure ls from bravo will see it. Make sure the uid and the gid appear the same on alpha and bravo.)

Boot Jenkins on alpha and bravo by issuing /etc/init.d/jenkins start. Now Jenkins Enterprise boots up in a two-node HA cluster. Access http://alpha:8080/ and http://bravo:8080/. One will serve the familiar Jenkins UI, and the other will tell you that it's acting as a stand-by node. In the Jenkins UI, go to "Manage Jenkins" then click "High Availability Status" and make sure two nodes are listed as members. You can kill the primary JVM (for example by kill -9 PID) while watching the log file via tail -f /var/log/jenkins/jenkins.log, and you will see the stand-by node take over the primary role.

Finally, we set up a monitoring service to ensure that the floating IP address gets assigned to the system that's hosting the primary. To do this, log on to alpha and install the jenkins-ha-monitor package.

This monitoring program watches Jenkins as root, and when the role transition occurs, it will execute the promotion script or the demotion script. In this tutorial, we will make these scripts assign/release the floating IP address.

In /etc/jenkins-ha-monitor/promotion.sh, write the following script:

```
#!/bin/sh
# assign the floating IP address 1.2.3.4 as an alias of eth1
ifconfig eth1:100 1.2.3.4
```

Similarly, in /etc/jenkins-ha-monitor/demotion.sh, write the following script:

```
#!/bin/sh
# release the floating IP address
ifconfig eth1:100 down
```

eth1:100 needs to be your network interface name followed by an unique alias ID (see "Linux IP aliasing" for more details.) Now that the configuration file is updated, restart the JA monitoring service by running /etc/init.d/jenkins-ha-monitor restart. Access the "High Availability Status" in the "Manage Jenkins" section from the web UI to verify that the monitoring service is recognized as a part of the cluster. Run ifconfig to verify that the virtual IP address is assigned to the system that's hosting the primary JVM.

Once you have completed these steps successfully, you will have a highly-available Jenkins instance!

Using HAproxy as a reverse proxy

Let's expand on this setup further by introducing an external load balancer / reverse proxy that receives traffic from users, then direct them to the active primary JVM.

Compared to IP aliasing, this is more complex, but it allows two nodes that aren’t in the same subnet to form a cluster, and you can set this up without having a root access.

HAproxy can be installed on most Linux systems via native packages, such as apt-get install haproxy or yum install haproxy. For Jenkins Enterprise HA, the configuration file (normally /etc/haproxy/haproxy.cfg) should look like the following:

```
global
        log 127.0.0.1   local0
        log 127.0.0.1   local1 notice
        maxconn 4096
        user haproxy
        group haproxy
defaults
        log     global
        # The following log settings are useful for debugging
        # Tune these for production use
        option  logasap
        option  http-server-close
        option  redispatch
        option  abortonclose
        option  log-health-checks
        mode    http
        option  dontlognull
        retries 3
        maxconn         2000
        timeout         http-request    10s
        timeout         queue           1m
        timeout         connect         10s
        timeout         client          1m
        timeout         server          1m
        timeout         http-keep-alive 10s
        timeout         check           500
        default-server  inter 5s downinter 500 rise 1 fall 1
listen application 0.0.0.0:80
  balance roundrobin
  reqadd    X-Forwarded-Proto:\ http
  option    forwardfor except 127.0.0.0/8
  option    httplog
  option    httpchk HEAD /ha/health-check
  server    alpha alpha:8080 check
  server    bravo bravo:8080 check
listen jnlp 0.0.0.0:10001
  mode      tcp
  option    tcplog
  timeout   server 15m
  timeout   client 15m
  # Jenkins by default runs a ping every 10 minutes and waits 4
  # minutes for a timeout before killing the connection, thus we
  # need to keep these TCP raw sockets open for at least that
  # long.
  option    httpchk HEAD /ha/health-check
  server    alpha alpha:10001 check port 8080
  server    bravo bravo:10001 check port 8080
listen ssh 0.0.0.0:2022
  mode      tcp
  option    tcplog
  option    httpchk HEAD /ha/health-check
  server    alpha alpha:2022 check port 8080
  server    bravo bravo:2022 check port 8080
listen status 0.0.0.0:8081
  stats enable
  stats uri /
```

The global section is stock settings. “defaults” has been modified to include additional logging for debugging. It is advisable to review and tune the log settings before production use. “defaults” has also been configured with typical timeout settings. Again, these should be tuned before production use.

The part that you will need to add and configure specifically for your environment is the listen blocks for application, JNLP and ssh. These tell HAproxy to forward traffic to two servers alpha and bravo, and periodically check their health by sending a GET request to /ha/health-check. Stand-by nodes do not respond positively to this health check, while the active and primary Jenkins master will. This is how HAproxy will know which of the two machines to send the traffic to.

Note the specific timeout overrides for the JNLP service based on Jenkins internal behavior.

For the JNLP and ssh services HAproxy is configured to forward tcp requests. For these services we have specifically configured HAproxy to use the same health check on the application port (8080). This ensures that all services fail over together when the health check fails.

The second listen status section allows you to monitor HAproxy by accessing port 8081. This is handy when you want to understand how HAproxy is behaving.