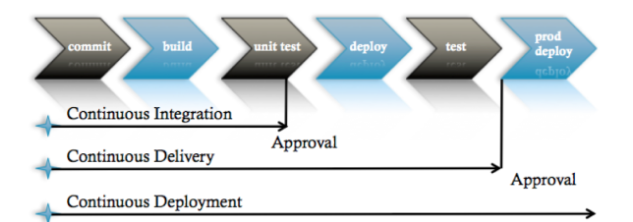
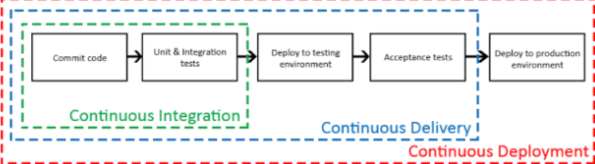
**What is pipeline in Jenkins(sonar,nexus,tomcat)**





* It models simple to complex pipelines as code by using **Groovy DSL** (Domain Specific Language)

<https://wilsonmar.github.io/jenkins2-pipeline/>

<https://wilsonmar.github.io/jenkins2-pipeline/>

<https://www.sourcefield.nl/post/jenkins-pipeline-tutorial/>

<https://dzone.com/articles/how-to-use-the-jenkins-declarative-pipeline>

Creating a Multibranch Pipeline

The **Multibranch Pipeline** project type enables you to implement different Jenkinsfiles for different branches of the same project. In a Multibranch Pipeline project, Jenkins automatically discovers, manages and executes Pipelines for branches which contain a Jenkinsfile in source control.

<https://jenkins.io/doc/tutorials/build-a-multibranch-pipeline-project/#end-to-end-multibranch-pipeline-project-creation>

$@ = stores all the arguments in a list of string  
$\* = stores all the arguments as a single string  
$# = stores the number of arguments

echo $!

Will print blank if you haven't yet run any **process in background** in current shell. If you now run:

date &

echo $!

Then it will print something like (i.e. process id of last executed background process):

47833

$! - shows last process ID which has started in background.

I am trying to get the current workspace of my Jenkins build using a Groovy pipeline script:

node('master') {

// PULL IN ENVIRONMENT VARIABLES

// Jenkins makes these variables available for each job it runs

def buildNumber = env.BUILD\_NUMBER

def workspace = env.WORKSPACE

def buildUrl = env.BUILD\_URL

Using [process expansion](http://fishshell.com/docs/current/index.html#expand-process), you could write

set PID %1 # if you know it's the first background job

set PID %something # if you know it's the only "something" running

Otherwise, we can use the [jobs](http://fishshell.com/docs/current/commands.html#jobs) command

set PID (jobs -l | awk '{print $2}') # just get the pid

jobs -l | read jobid pid cpu state cmd # get all the thing

**How to run the Jenkins pipeline even if the given stage would be failed and run remaining stages**

Use

propagate: false

flag to move to next stage when previous stage fails.

example:

stage('<stage-name>'){

node('<node-name>'){

build job: '<job-name>', propagate: false

}

}

stage('<stage-name>'){

node('<node-name>'){

build job: '<job-name>'

}

}

When you install this plugin, your project configuration page gets additional "Post build task" option as illustrated below:

There is a nice option in the Check Precondition stage to enable/disable "Fail Pipeline". See screenshot:

You are supposed to set it yourself. You can use a try catch block for this.

Example that forces a failure then prints result:

node {

try {

// do something that fails

sh "exit 1"

currentBuild.result = 'SUCCESS'

} catch (Exception err) {

currentBuild.result = 'FAILURE'

}

echo "RESULT: ${currentBuild.result}"

}

https://medium.com/@Lenkovits/jenkins-pipelines-and-their-dirty-secrets-1-9e535cd603f4

**various ways of scheduling jobs in Jenkins**

scheduling a job that gets automatically triggered by changes in an SCM repository such as GitHub, Bitbucket

**We have to provide value in a**[**Cron-compliant format**](https://www.baeldung.com/cron-expressions). There’s extensive information available on the page if we click the question mark next to the box.

Let’s type *\*/2 \* \* \* \** here, which represents an interval of two minutes:

In the Build Triggers section, **instead of selecting Build Periodically, let’s select Poll SCM**. As soon as we do that, we should see a text box with Label Schedule.

https://medium.com/@Lenkovits/jenkins-pipelines-and-their-dirty-secrets-1-9e535cd603f4

H/2 \* \* \* \* is polling every second minute, but H/1 \* \* \* \* only once per hour

H/2 \* \* \* \* is polling every second minute.  
H/1 \* \* \* \* is polling once per hour.

Try \* \* \* \* \* to run every minute. \* \* \* \* \* https://crontab.guru/#\*\_\*\_\*\_\*\_\*

Unfortunately H/1 \* \* \* \* does not work due to open defect.

**What is the difference between host and inventory file in ansible**

Host variable files store host specific variables, like more workers on a host with more memory, while inventory files are like a list of hosts which can be referred in playbooks.

Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systems listed in Ansible’s inventory, which defaults to being saved in the location /etc/ansible/hosts. You can specify a different inventory file using the -i <path> option on the command line.

When ansible-playbook is executed with --check it will not make any changes on remote systems. Instead, any module instrumented to support ‘check mode’ (which contains most of the primary core modules, but it is not required that all modules do this) will report what changes they would have made rather than making them. Other modules that do not support check mode will also take no action, but just will not report what changes they might have made.

Check mode is just a simulation, and if you have steps that use conditionals that depend on the results of prior commands, it may be less useful for you. However it is great for one-node-at-time basic configuration management use cases.

Example:

ansible-playbook foo.yml --check

There are two options:

1. Force a task to **run in check mode**, even when the playbook is called **without** --check. This is called check\_mode: yes.
2. Force a task to **run in normal mode** and make changes to the system, even when the playbook is called **with** --check. This is called check\_mode: no.

**Note**

Prior to version 2.2 only the equivalent of check\_mode: no existed. The notation for that was always\_run: yes

What we do is to put it in a separate directory on the server (you could use something like /config, /opt/config, /root/config, /home/username/config, or anything you want). When our servlets start up, they read the XML file, get a few things out of it (most importantly DB connection information), and that's it.

Most of the time, I store development application config in root directory of the project, like this:

app

|-- config.json

But that doesn't seem to be the best approach, since this config ends up being stored in version control system - possibly resulting in leaked usernames, passwords and other sensitive stuff.

... stores config in environment variables. Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

We are battling the same problem where I work. Right now all of our configurations are file-based and source controlled with the individual applications that use them. This leads to duplication and to the developers having access to production/qa passwords instead of just development.

That said I think we've come up with a good solution going forward. We are moving our config files to a separate git repo (labeled the config repo).

Instead of storing the whole configuration in one file, store it in several files.

* Have a configuration *directory*. All files there are interpreted as configuration files, except maybe README\*.
* All file names are sorted alphabetically, and the files are loaded in that order. This is why files in such cases often start with a digit or two: 01-logging.json. 02-database.json, etc.
* Data from all the files are loaded into the *same* configuration structure available to the application. This is how several files can complement each others' settings, and even override them in a predictable way.
* Only store in the VCS the config files with safe-to-see values, or default values. Add the config files with secrets during deployment, or, better yet, use an authenticated secrets storage service.

On your nearest Linux box, take a look at /etc/sudoers.d or /etc/nginx/conf.d. It shows the same pattern.

Secrets management is a different beast. You can manage them as a manual step while you're small. You can use things like Zookeeper. You can even [check the secrets into a VCS *in encrypted form*](https://www.agwa.name/projects/git-crypt/), and decrypt them as a deployment step. A number of other options exists.

(Also, an opinion piece: JSON is not a good config file format, because it does not allow comments; comments are crucial. TOML, YAML, and even INI formats are better in practical use.)

**Namespaces**. Generally speaking, a **namespace**(sometimes also called a context) is a naming system for making names unique to avoid ambiguity. ... **Namespaces**in **Python** are implemented as **Python** dictionaries, this means it is a mapping from names (keys) to objects (values).

**Local Namespace**

This namespace covers the local names inside a function. Python creates this namespace for every function called in a program. It remains active until the function returns.

**Global Namespace**

This namespace covers the names from various imported modules used in a project. Python creates this namespace for every module included in your program. It’ll last until the program ends.

**Built-in Namespace**

This namespace covers the built-in functions and built-in exception names. Python creates it as the interpreter starts and keeps it until you exit.

What Is Scope In Python?

Namespaces make our programs immune from name conflicts. However, it doesn’t give us a free ride to use a variable name anywhere we want. Python restricts names to be bound by specific rules known as a scope

Python Decoders:

This module defines base classes for standard **Python** codecs (encoders and**decoders**) and provides access to the internal **Python** codec registry which manages the codec and error handling lookup process. Encodes obj using the codec registered for encoding. The default encoding is 'ascii' .

The method **decode()** decodes the string using the codec registered for *encoding*. It defaults to the default string encoding.

## Syntax

Str.decode(encoding='UTF-8',errors='strict')

## Parameters

* **encoding** − This is the encodings to be used. For a list of all encoding schemes please visit: [Standard Encodings.](http://docs.python.org/library/codecs.html#standard-encodings)
* **errors** − This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register\_error().

#### Table 3.1 Built-in Python Types

|  |  |  |
| --- | --- | --- |
| **Type Category** | **Type Name** | **Description** |
| None | types.NoneType | The null object None |
| Numbers | int | Integer |
|  | long | Arbitrary-precision integer |
|  | float | Floating point |
|  | complex | Complex number |
|  | bool | Boolean (True or False) |
| Sequences | str | Character string |
|  | unicode | Unicode character string |
|  | basestring | Abstract base type for all strings |
|  | list | List |
|  | tuple | Tuple |
|  | xrange | Returned by xrange() |
| Mapping | dict | Dictionary |
| Sets | set | Mutable set |
|  | frozenset | Immutable set |
| Callable | types.BuiltinFunctionType | Built-in functions |
|  | types.BuiltinMethodType | Built-in methods |
|  | type | Type of built-in types and classes |
|  | object | Ancestor of all types and classes |
|  | types.FunctionType | User-defined function |
|  | types.InstanceType | Class object instance |
|  | types.MethodType | Bound class method |
|  | types.UnboundMethodType | Unbound class method |
| Modules | types.ModuleType | Module |
| Classes | object | Ancestor of all types and classes |
| Types | type | Type of built-in types and classes |
| Files | file | File |
| Internal | types.CodeType | Byte-compiled code |
|  | types.FrameType | Execution frame |
|  | types.GeneratorType | Generator object |
|  | types.TracebackType | Stacks traceback of an exception |
|  | types.SliceType | Generated by extended slices |
|  | types.EllipsisType | Used in extended slices |
| Classic Classes | types.ClassType | Old-style class definition |
|  | types.InstanceType | Old-style class instance |

*lambda operator or lambda function*is used for creating small, one-time and anonymous function objects in Python.

**Basic syntax**

lambda arguments : expression

how to find highest cpu utilization of different processes in shell

The following command will show the list of top processes ordered by RAM and CPU use in descendant form (remove the **pipeline** and **head** if you want to see the full list):

# ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head

The **-o** (or **–format**) option of **ps** allows you to specify the output format. A favorite of mine is to show the processes’ **PIDs** (**pid**), **PPIDs** (**pid**), the name of the executable file associated with the process (**cmd**), and the RAM and CPU utilization (**%mem** and **%cpu**, respectively).

Additionally, I use **--sort** to sort by either **%mem** or **%cpu**. By default, the output will be sorted in ascendant form, but personally I prefer to reverse that order by adding a minus sign in front of the sort criteria.

To add other fields to the output, or change the sort criteria, refer to the **OUTPUT FORMAT CONTROL** section in the man page of **ps** command.

ps -eo pcpu,pid,user,args | sort -k1 -r -n | head -10

Works for me, show the top 10 cpu using threads, sorted numerically

In addition to ps and top commands, you can also run vmstat to figure out what is happening in terms of CPU, memory usage on the system, i.e.:

vmstat 1 100

**resolv**.**conf**. **resolv**.**conf** is the name of a computer **file used** in various operating systems to configure the system's Domain Name System (DNS) resolver. The **file** is a plain-text **file** usually created by the network administrator or by applications that manage the **configuration** tasks of the system. This **file** is another text **file**, **used by** the resolver a library that determines the IP address for a host name

How to install same software in 100 servers at a time ansible yaml file example

# ansible -a "/bin/hostnamectl --static" webservers

In the example above, we ran **hostnamectl --static** on **node1** and **node2**. It doesn’t take long for one to realize that this method of running tasks on remote computers works fine for short commands but can quickly become burdensome or messy for more complex tasks that require further well-structured configuration parameters or interactions with other services

Simply put, **Playbooks** are plain text files written in the **YAML** format, and contain a list with items with one or more key/value pairs (also known as a “**hash**” or a “**dictionary**”).

Inside each Playbook you will find one or more group of hosts (each one of these groups is also called a **play**) where the desired tasks are to be performed.

An example from the official docs will help us to illustrate:

**1.** **hosts**: this is a list of machines (as per **/etc/ansible/hosts**) where the following tasks will be performed.

**2.** **remote\_user**: remote account that will be used to perform the tasks.

**3.** **vars**: variables used to modify the behavior of the remote system(s).

**4.** tasks are executed in order, one at a time, against all machines that match hosts. Within a play, all hosts are going to get the same task directives.

If you need to execute a different set of associated tasks for a specific host, create another play in the current **Playbook** (in other words, the purpose of a play is to map a specific selection of hosts to well-defined tasks).

In that case, start a new play by adding the hosts directive at the bottom and starting over:

---

- hosts: webservers

remote\_user: root

vars:

variable1: value1

variable2: value2

remote\_user: root

tasks:

- name: description for task1

task1: parameter1=value\_for\_parameter1 parameter2=value\_for\_parameter2

- name: description for task1

task2: parameter1=value\_for\_parameter1 parameter2=value\_for\_parameter2

handlers:

- name: description for handler 1

service: name=name\_of\_service state=service\_status

- hosts: dbservers

remote\_user: root

vars:

variable1: value1

variable2: value2

…

**5.** handlers are actions that are triggered at the end of the tasks section in each play, and are mostly used to restart services or trigger reboots in the remote systems.

# mkdir /etc/ansible/playbooks

And a file named **apache.yml** inside of there with the following contents:

---

- hosts: webservers

vars:

http\_port: 80

max\_clients: 200

remote\_user: root

tasks:

- name: ensure apache is at the latest version

yum: pkg=httpd state=latest

- name: replace default index.html file

copy: src=/static\_files/index.html dest=/var/www/html/ mode=0644

notify:

- restart apache

- name: ensure apache is running (and enable it at boot)

service: name=httpd state=started enabled=yes

handlers:

- name: restart apache

service: name=httpd state=restarted

Second, create a directory /static\_files:

# mkdir /static\_files

where you will store the custom **index.html** file:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8"/>

</script>

</head>

<body>

<h1>Apache was started in this host via Ansible</h1><br>

<h2>Brought to you by Tecmint.com</h2>

</body>

</html>

That said, now it’s time to use this playbook to perform the tasks mentioned earlier. You will note that Ansible will go through each task by host, one at a time, and will report on the status of such tasks:

# ansible-playbook /etc/ansible/playbooks/apache.yml

# ansible-playbook /etc/ansible/playbooks/apache.yml

This time, the task reports that the Apache web server was started and enabled on each host

Ansible Modules[Permalink](https://linode.com/docs/applications/configuration-management/automatically-configure-servers-with-ansible-and-playbooks/#ansible-modules)

Ansible ships with a large collection of modules that you can run as tasks or via commands as needed. To see a listing of all available modules, run:

ansible-doc -l

Common core modules include:

* command - [Executes a command on a remote node](http://docs.ansible.com/ansible/command_module.html)
* script - [Runs a local script on a remote node after transferring it](http://docs.ansible.com/ansible/script_module.html)
* shell - [Execute commands in nodes](http://docs.ansible.com/ansible/shell_module.html)
* mysql\_db - [Add or remove MySQL databases from a remote host](http://docs.ansible.com/ansible/mysql_db_module.html)
* mysql\_user - [Adds or removes a user from a MySQL database](http://docs.ansible.com/ansible/mysql_user_module.html)
* postgresql\_db - [Add or remove PostgreSQL databases from a remote host](http://docs.ansible.com/ansible/postgresql_db_module.html)
* postgresql\_user - [Adds or removes a users (roles) from a PostgreSQL database](http://docs.ansible.com/ansible/postgresql_user_module.html)
* fetch - [Fetches a file from remote nodes](http://docs.ansible.com/ansible/fetch_module.html)
* template - [Templates a file out to a remote server](http://docs.ansible.com/ansible/template_module.html)
* yum - [Manages packages with the yum package manager](http://docs.ansible.com/ansible/yum_module.html)
* apt - [Manages apt-packages](http://docs.ansible.com/ansible/apt_module.html)
* git - [Deploy software (or files) from git checkouts](http://docs.ansible.com/ansible/git_module.html)
* service - [Manage services](http://docs.ansible.com/ansible/service_module.html)

update and upgrade package

#apt-get update  
  
Is the command to update the source list, if you modify the source list or you want to make a sync refresh or added new ppa source then you should execute above command.  
  
Where as  
  
#apt-get upgrade  
  
Command will try to download all the packages which are having updates at apt server and then try to install them if you press “y”. This something like System upgrade to new packages.

"update" updates the package list with any newly available packages from the mirrors and "upgrade" executes the actual upgrading of packages. That's it.

$ ps aux

**Show Running Processes with PID**

Jenkins 2 brings Pipeline as code, a new setup experience and other UI improvements all while maintaining total backwards compatibility with existing Jenkins installations.

Highlights of Jenkins 2:

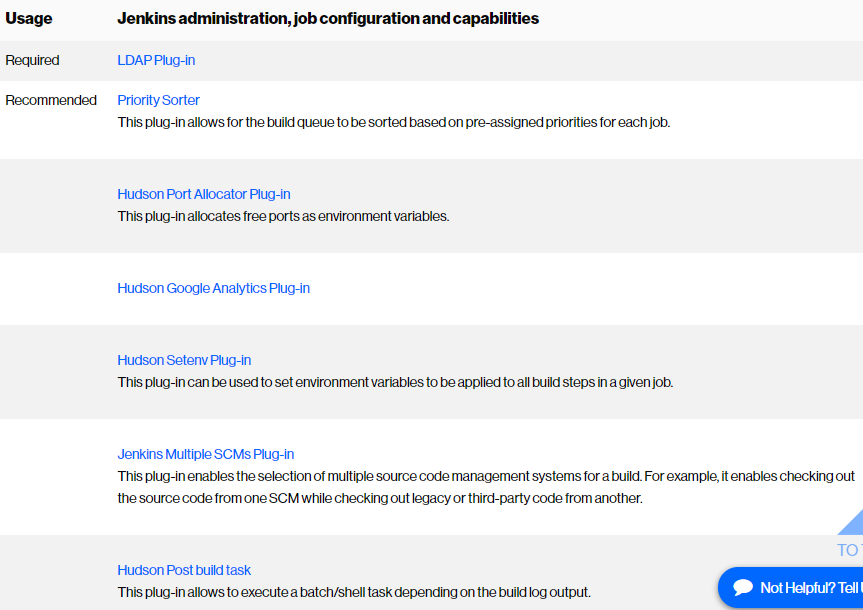
* [**Built-in support for delivery pipelines.**](https://jenkins.io/2.0/#pipelines)
* [**Improved usability.**](https://jenkins.io/2.0/#ux)
* [**Fully backwards compatible.**](https://jenkins.io/2.0/#compat)

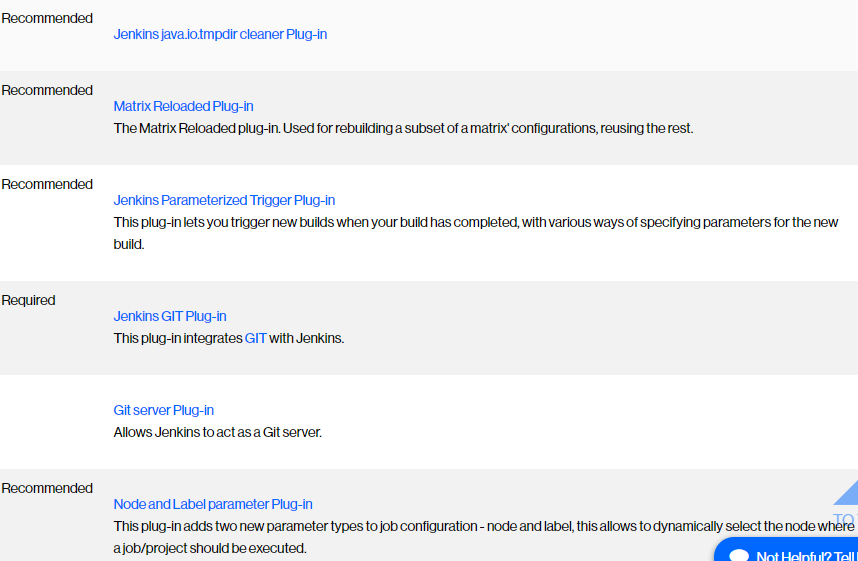
### Key Features and Benefits of Pipelines

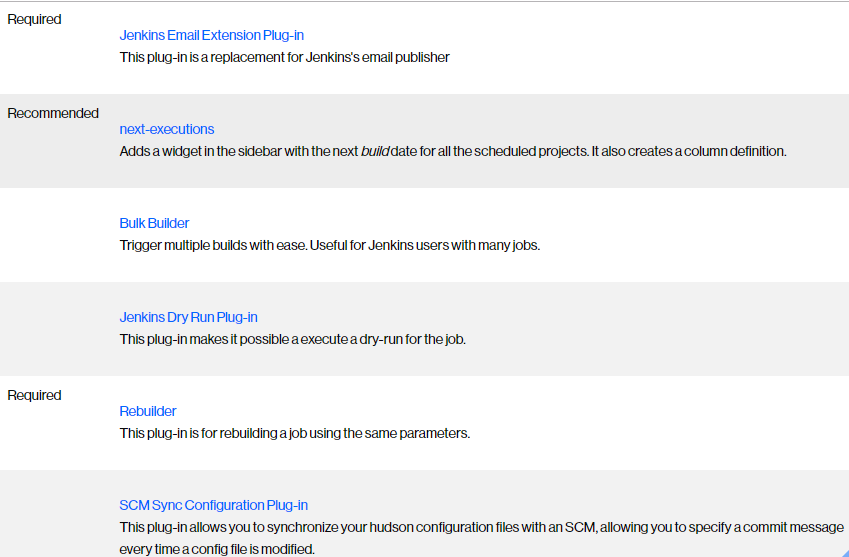
#### Pipeline as Code

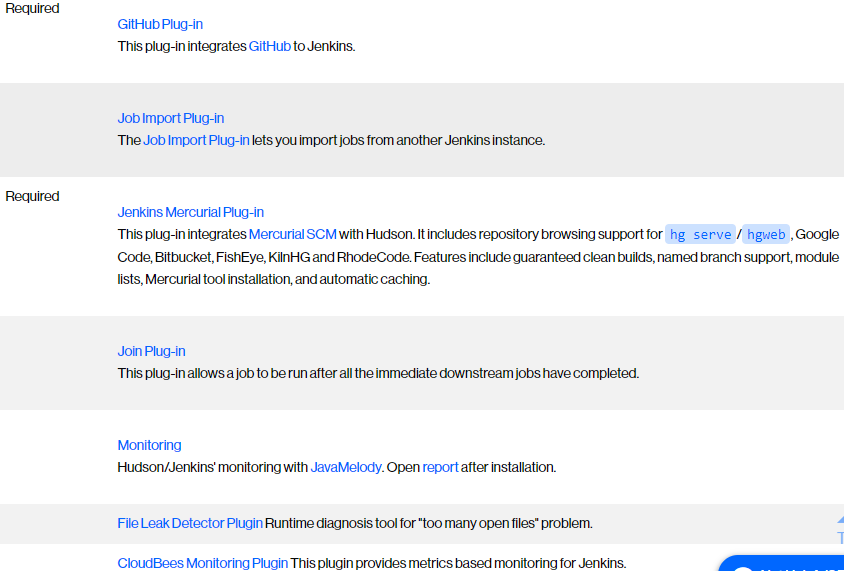
* Easily define simple and complex pipelines through the DSL in a Jenkinsfile.
* Pipeline as code provides a common language to help teams (e.g. Dev and Ops) work together.
* Easily share pipelines between teams by storing common "steps" in shared repositories.

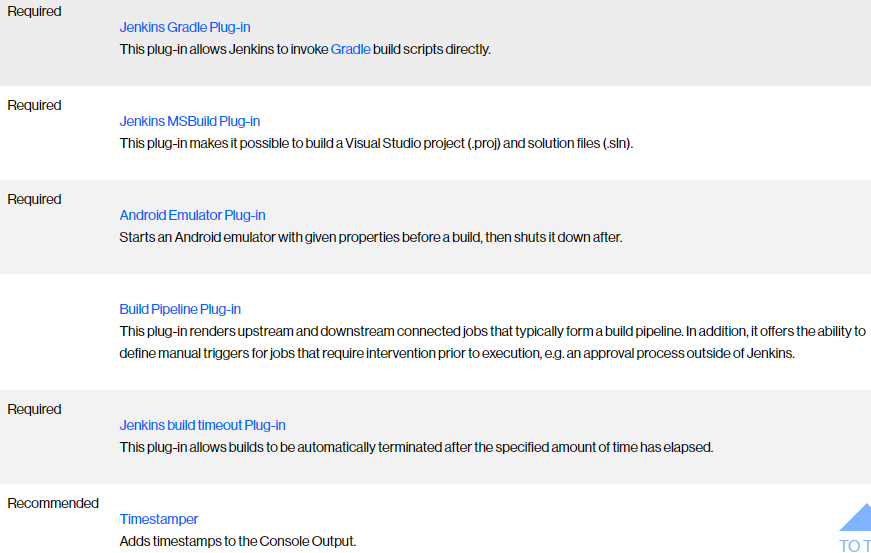
**Types of dffeent plugins used in Jenkins**

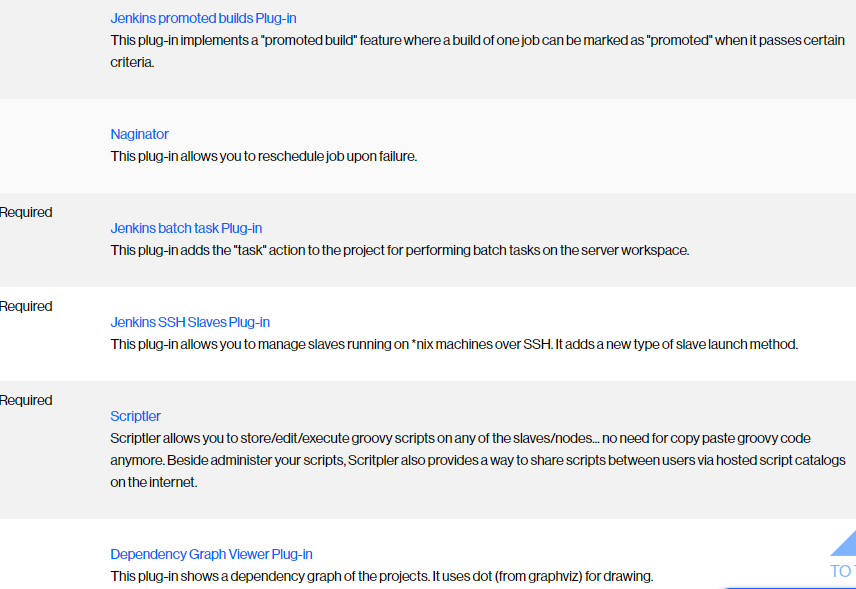


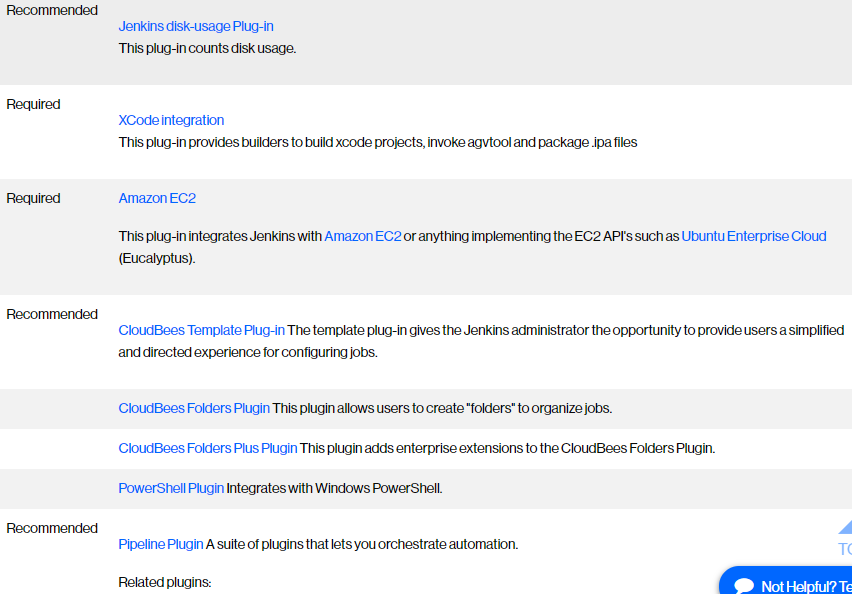


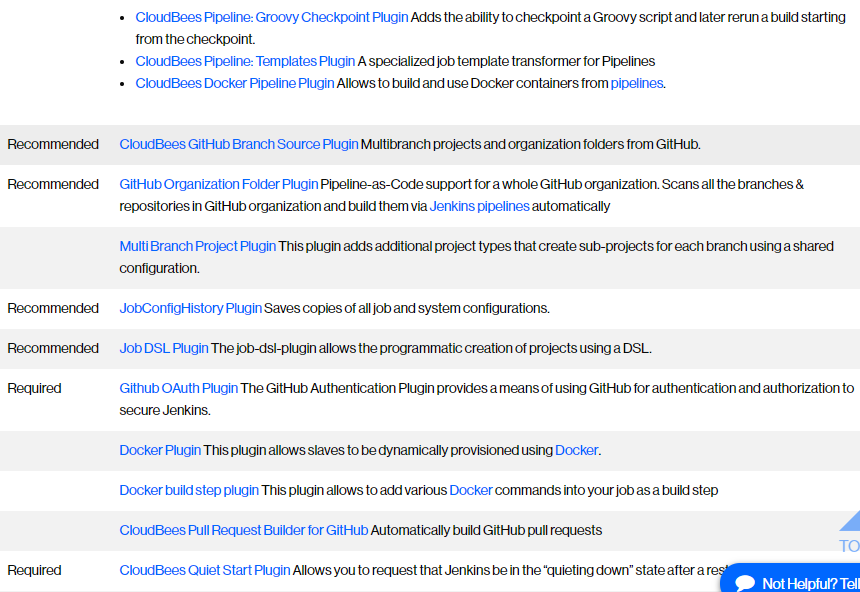


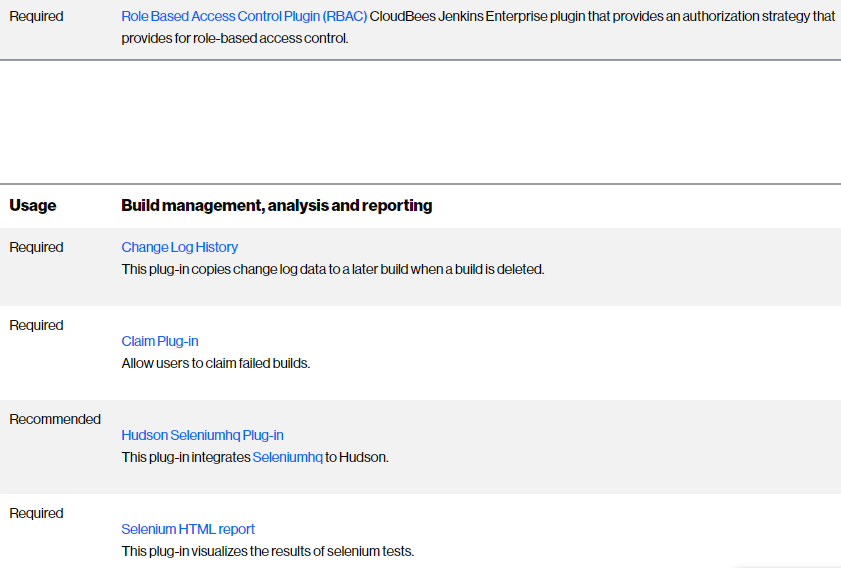


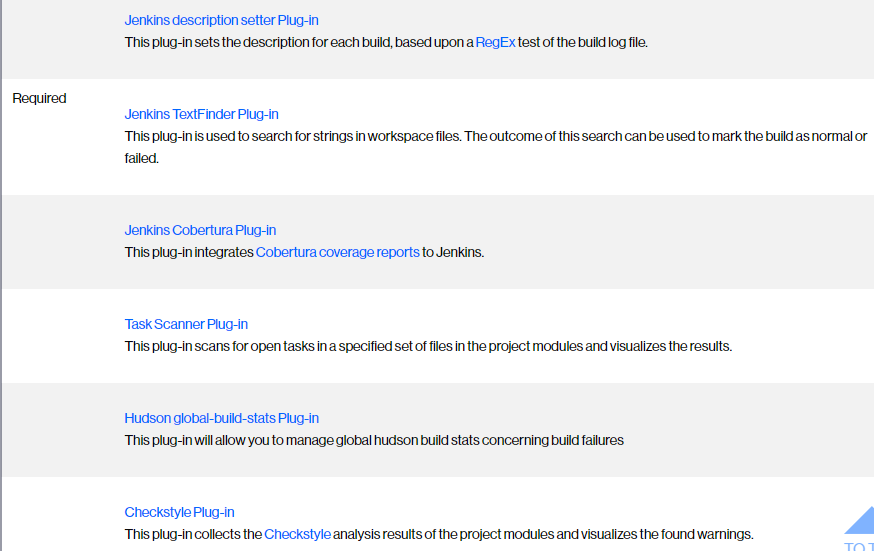


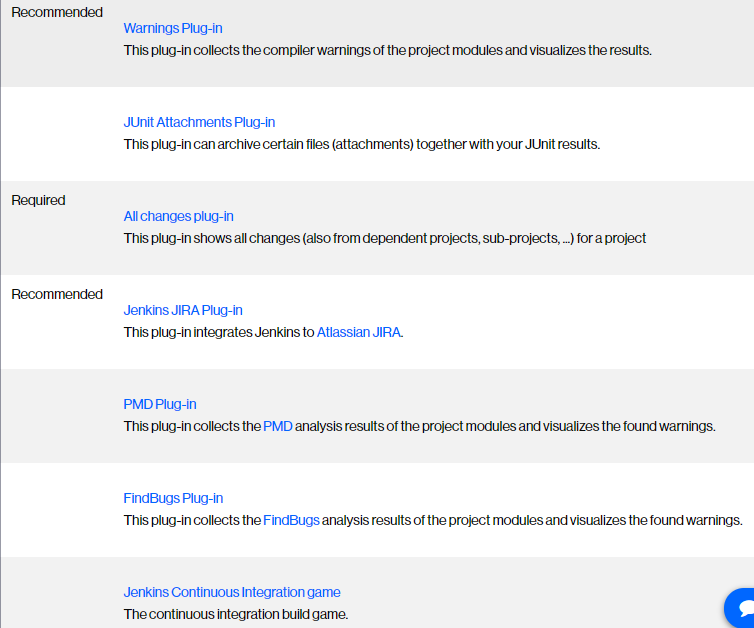


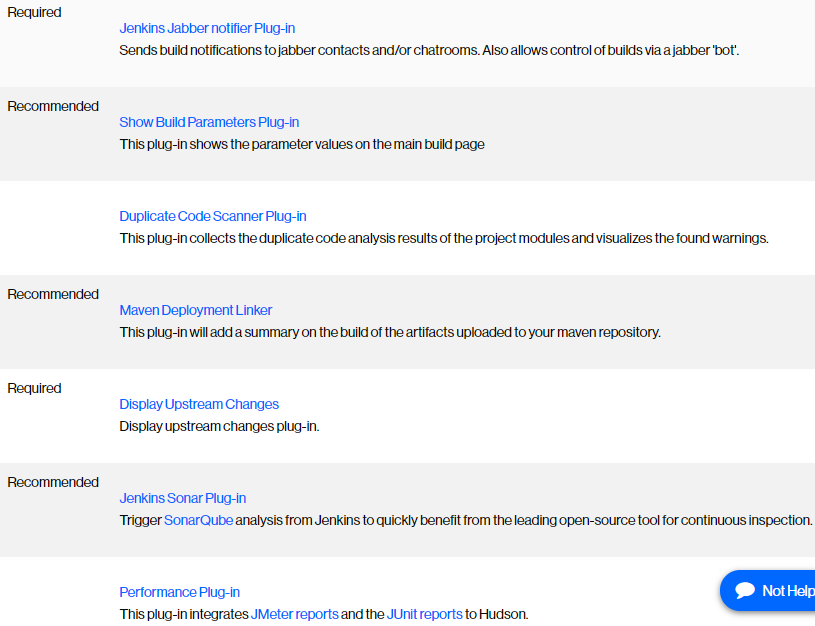


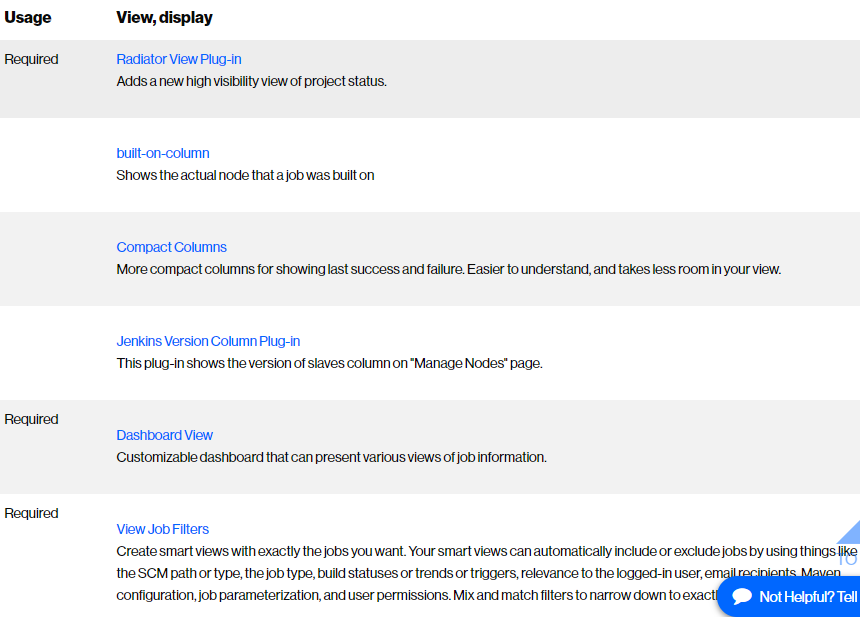




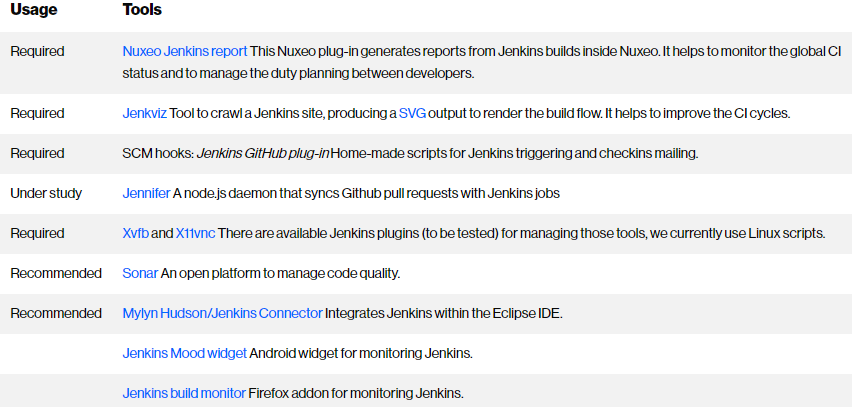












diff/b pipeline and freestyle

# Resolution

Let’s see the same job in three different scenarios:  
\* Freestyle Job.  
\* Scripted Pipeline Job.  
\* Declarative Pipeline Job.

**FreestyleJob**

You need two jobs.

1. You need to set up the label for the agent with "SlaveSCMConnected" and configure your repository and add an "Execute shell" build with:

\* mvn clean install

2. You need to set up the label for the agent with "SlaveWithoutConnectionToSCM" and configure an email post-build step with the corresponding configuration.

**Scripted Pipeline Job**

node('SlaveSCMConnected') {

stage('SCM operation'){

checkout scm

sh 'mvn clean install'

}

}

node('SlaveWithoutConnectionToSCM')

stage('Operation without SCM'){

mail (to: 'devops@acme.com',

subject: "Job '${env.JOB\_NAME}' (${env.BUILD\_NUMBER}) is waiting for input",

body: "Please go to ${env.BUILD\_URL}.");

}

}

Declarative Pipeline JOB

pipeline {

agent none

options {

skipDefaultCheckout()

}

stages{

stage('SCM operation'){

agent {label 'SlaveSCMConnected'}

steps{

checkout scm

sh 'mvn clean install'

}

}

stage('Operation without SCM'){

agent {label 'SlaveWithoutConnectionToSCM'}

steps{

mail (to: 'devops@acme.com',

subject: "Job '${env.JOB\_NAME}' (${env.BUILD\_NUMBER}) is waiting for input",

body: "Please go to ${env.BUILD\_URL}.");

}

}

}

}

# General Job vs Scripted Pipeline Job

**Main difference**

The main difference between any job and a Pipeline Job is that the **Pipeline Scripted job runs on the Jenkins master, using a lightweight executor expected to use very few resources in order to translate the pipeline to atomic commands** that execute or send to the agents.

In our example, in the Freestyle job everything is executed in the agent, but for the Scripted Pipeline Job, the pipeline code is translated in the master to atomic commands that are sent to the agents.

# Scripted Pipeline Job vs Declarative Pipeline

**Main difference**

There are several differences:

* You need to use a Jenkinsfile to define it that have to be located in an SCM.
* By default each step is going to do a “checkout scm” on each “node”.

In our example, if the Master doesn’t have access to the SCM it couldn’t work because the first step is that the Master get the Jenkinsfile from the repository and translate the steps in atomic commands.

One of our nodes doesn’t have access to the SCM so by default it’ll fail because each “node” do a “checkout scm” perhaps it doesn’t need anything from there.

To avoid that you need to add this piece of code:

options {

skipDefaultCheckout()

}

[**https://jenkins.io/pipeline/getting-started-pipelines/**](https://jenkins.io/pipeline/getting-started-pipelines/)

A maven project is a project that will analyze the pom.xml file in greater detail and produce a project that's geared towards the targets that are invoked. The maven project is smart enough to incorporate build targets like the javadoc or test targets and automatically setup the reports for those targets. There is little configuration required for it. See: <https://wiki.jenkins.io/display/JENKINS/Building+a+maven2+project> for more information on Maven project.

A Free-Style project is a project that can incorporate almost any type of build. While a maven project you can only build maven projects, the Free-Style project is the more "generic" form of a project. You can execute shell/dos scripts, invoke ant, and a lot more. Majority of the plugins are written to use the free-style project. The maven module is limited in that it can't invoke a shell script, or anything else just the maven targets

if you've used jenkins in the past without using a Jenkinsfile, you've used something more similar to a freestyle project.

if you hate typing things into CI systems and therefore want to use pipelines as code--where you put all of your CI configuration into a file in source control (Jenkinsfile) and let Jenkins read that file to figure out what to do--use [pipelines](https://jenkins.io/solutions/pipeline/). once you know pipelines, there won't be many cases where you'll prefer freestyle projects.

The difference is that in Pipeline we have the ability to break our jobs out into different stages and we can have whatever stages we'd like to represent the process we use to deploy software and of course, if anything goes wrong, we can see which stage had the problem; for example. We even have the ability to add in verification before we proceed. We have the ability to run stages in parallel so we could have multiple tests executing in separate branches very easily

**difference between Etcd,Kubectl,Kubelet**

Etcd is a [key-value store](https://en.wikipedia.org/wiki/Key-value_database) where Kubernetes (K8s) stores all of the cluster and configuration data.

Components like your pods, or replicasets, or deployments, etc. It holds the state that these should be in (what should be running, what shouldn’t). Etcd has a watch feature that is heavily used by K8s for monitoring when configurations change.

* [**kubelet**](https://kubernetes.io/docs/admin/kubelet/), which communicates with the Kubernetes Master.
* [**kube-proxy**](https://kubernetes.io/docs/admin/kube-proxy/), a network proxy which reflects Kubernetes networking services on each node

Set which Kubernetes cluster **kubectl** communicates with and modifies configuration information

# [How can I schedule a job to run in the future, but only one time ever](https://stackoverflow.com/questions/35029486/how-can-i-schedule-a-job-to-run-in-the-future-but-only-one-time-ever)

I want to schedule a job that will execute at most once at a specified time in the future (usually that time would just be later that night, after hours). I want to use Jenkins to do this. If Jenkins happened to be down during that time, the job wouldn't fire - that's fine.

At the moment, I'm planning to make a new job with "build periodically" enabled, and set the schedule to something like "0 19 29 01 \*".

The intent being to schedule the job to run at 19:00 on the 29th of January.

The downside being that every time I do this, I must remember to delete/disable the job sometime before next year, or it will run again (that would be bad). I'd be checking the results of the job manually anyway, so not too hard to do. I'm just hoping someone might be able to point out a way to schedule a "one off" job run with Jenkins - that way I can't mess things up by forgetting to disable the job

**How many types of deployments will be there in tomcat**

2 ways

<https://jenkins.io/doc/tutorials/build-a-multibranch-pipeline-project/>

<https://jenkins.io/blog/2015/12/03/pipeline-as-code-with-multibranch-workflows-in-jenkins/>

Now, you can add a dependency in pom.xml. For example, if you want to add LoginRadius for Java library/dependency, you can add as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7 Dependency Scope Transitive Dependencies Discovery can be restricted using various Dependency Scope as mentioned below.   |  |  | | --- | --- | | **Sr.No.** | **Scope & Description** | | 1 | **compile**  This scope indicates that dependency is available in classpath of project. It is default scope. | | 2 | **provided**  This scope indicates that dependency is to be provided by JDK or web-Server/Container at runtime. | | 3 | **runtime**  This scope indicates that dependency is not required for compilation, but is required during execution. | | 4 | **test**  This scope indicates that the dependency is only available for the test compilation and execution phases. | | 5 | **system**  This scope indicates that you have to provide the system path. | | 6 | **import**  This scope is only used when dependency is of type pom. This scope indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section. | | <dependencies>  ….  <dependency>              <groupId>com.loginradius.sdk</groupId>              <artifactId>java-sdk</artifactId>              <version>3.2.0</version>  </dependency>  …..  </dependencies> |

How to share docker images between one host to other host

You will need to save the Docker image as a tar file:

docker save -o <path for generated tar file> <image name>

Then copy your image to a new system with regular file transfer tools such as cp or scp. After that you will have to load the image into Docker:

docker load -i <path to image tar file>

Export vs. Save

Docker supports two different types of methods for saving container images to a single tarball:

* docker export - saves a container’s running or paused *instance* to a file
* docker save - saves a non-running container *image* to a file
* docker export command to export the instance’s image:
* $ # output truncated slightly
* $ docker ps
* CONTAINER ID IMAGE COMMAND
* defc757ce803 ubuntu "/bin/bash"
* $ docker export defc | gzip > ubuntu.tar.gz
* $ ls -lah ubuntu.tar.gz
* -rw-r--r-- 1 bob staff 63M Aug 28 13:56 ubuntu.tar.gz
* We can now have Bob upload the ubuntu.tar.gz file on his computer to Alice’s computer using [sneakernet](https://en.wikipedia.org/wiki/Sneakernet). Once Bob finishes, Alice can use a docker import on the image and give it a new tag in the process:
* $ ls -lah ubuntu.tar.gz
* docker save on the Ubuntu image to give Alice a copy of the container’s *image*, which she could use without Bob’s modifications:
* $ docker save ubuntu | gzip > ubuntu-golden.tar.gz
* Alice would then take that copy and load it, instead of doing an import :
* $ gzcat ubuntu-golden.tar.gz | docker load
* App, DB micro services, where do you store the kubernetes credentials file

pods/inject/secret-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: secret-test-pod

spec:

containers:

- name: test-container

image: nginx

volumeMounts:

# name must match the volume name below

- name: secret-volume

mountPath: /etc/secret-volume

# The secret data is exposed to Containers in the Pod through a Volume.

volumes:

- name: secret-volume

secret:

secretName: test-secret

To check the version, enter **kubectl version**.

## Convert your secret data to a base-64 representation

Suppose you want to have two pieces of secret data: a username **my-app** and a password **39528$vdg7Jb**. First, use [Base64 encoding](https://www.base64encode.org/) to convert your username and password to a base-64 representation. Here’s a Linux example:

**echo -n 'my-app' | base64**

**echo -n '39528$vdg7Jb' | base64**

## Create a Secret

Here is a configuration file you can use to create a Secret that holds your username and password:

| [**pods/inject/secret.yaml**](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/pods/inject/secret.yaml) |
| --- |
| **apiVersion: v1**  **kind: Secret**  **metadata:**  **name: test-secret**  **data:**  **username: bXktYXBw**  **password: Mzk1MjgkdmRnN0pi** |

Create the Secret

**kubectl create -f https://k8s.io/docs/tasks/inject-data-application/secret.yaml**

**Note:**

If you want to skip the Base64 encoding step, you can create a Secret by using the **kubectl create secret** command:

**kubectl create secret generic test-secret --from-literal=username='my-app' --from-literal=password='39528$vdg7Jb'**

1. View information about the Secret:

kubectl get secret test-secret

Output:

**NAME TYPE DATA AGE**

**test-secret Opaque 2 1m**

1. View more detailed information about the Secret:

kubectl describe secret test-secret

Output:

**Name: test-secret**

**Namespace: default**

**Labels: <none>**

**Annotations: <none>**

**Type: Opaque**

**Data**

**====**

**password: 13 bytes**

**username: 7 bytes**

**Note:** If you want to skip the Base64 encoding step, you can create a Secret by using the **kubectl create secret** command:

**kubectl create secret generic test-secret --from-literal=username='my-app' --from-literal=password='39528$vdg7Jb'**

Create a Pod that has access to the secret data through a Volume

Here is a configuration file you can use to create a Pod:

| [**pods/inject/secret-pod.yaml**](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/pods/inject/secret-pod.yaml) |
| --- |
| **apiVersion: v1**  **kind: Pod**  **metadata:**  **name: secret-test-pod**  **spec:**  **containers:**  **- name: test-container**  **image: nginx**  **volumeMounts:**  ***# name must match the volume name below***  **- name: secret-volume**  **mountPath: /etc/secret-volume**  ***# The secret data is exposed to Containers in the Pod through a Volume.***  **volumes:**  **- name: secret-volume**  **secret:**  **secretName: test-secret** |

1. Create the Pod:

**kubectl create -f https://k8s.io/docs/tasks/inject-data-application/secret-pod.yaml**

1. Verify that your Pod is running:

**kubectl get pod secret-test-pod**

Output:

**NAME READY STATUS RESTARTS AGE**

**secret-test-pod 1/1 Running 0 42m**

1. Get a shell into the Container that is running in your Pod:

**kubectl exec -it secret-test-pod -- /bin/bash**

1. The secret data is exposed to the Container through a Volume mounted under **/etc/secret-volume**. In your shell, go to the directory where the secret data is exposed:

**root@secret-test-pod:/# cd /etc/secret-volume**

1. In your shell, list the files in the **/etc/secret-volume** directory:

**root@secret-test-pod:/etc/secret-volume# ls**

The output shows two files, one for each piece of secret data:

**password username**

1. In your shell, display the contents of the **username** and **password** files:

**root@secret-test-pod:/etc/secret-volume# cat username; echo; cat password; echo**

The output is your username and password:

**my-app**

**39528$vdg7Jb**

Create a Pod that has access to the secret data through environment variables

Here is a configuration file you can use to create a Pod:

| [**pods/inject/secret-envars-pod.yaml**](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/pods/inject/secret-envars-pod.yaml) |
| --- |
| **apiVersion: v1**  **kind: Pod**  **metadata:**  **name: secret-envars-test-pod**  **spec:**  **containers:**  **- name: envars-test-container**  **image: nginx**  **env:**  **- name: SECRET\_USERNAME**  **valueFrom:**  **secretKeyRef:**  **name: test-secret**  **key: username**  **- name: SECRET\_PASSWORD**  **valueFrom:**  **secretKeyRef:**  **name: test-secret**  **key: password** |

1. Create the Pod:

**kubectl create -f https://k8s.io/docs/tasks/inject-data-application/secret-envars-pod.yaml**

1. Verify that your Pod is running:

**kubectl get pod secret-envars-test-pod**

Output:

**NAME READY STATUS RESTARTS AGE**

**secret-envars-test-pod 1/1 Running 0 4m**

1. Get a shell into the Container that is running in your Pod:

**kubectl exec -it secret-envars-test-pod -- /bin/bash**

1. In your shell, display the environment variables:

**root@secret-envars-test-pod:/# printenv**

The output includes your username and password:

**...**

**SECRET\_USERNAME=my-app**

**...**

**SECRET\_PASSWORD=39528$vdg7Jb**

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. This page provides a series of usage examples demonstrating how to create ConfigMaps and configure Pods using data stored in ConfigMaps.

## Create a ConfigMap

Use the **kubectl create configmap** command to create configmaps from [directories](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#create-configmaps-from-directories), [files](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#create-configmaps-from-files), or [literal values](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#create-configmaps-from-literal-values):

**kubectl create configmap <map-name> <data-source>**

where <map-name> is the name you want to assign to the ConfigMap and <data-source> is the directory, file, or literal value to draw the data from.

The data source corresponds to a key-value pair in the ConfigMap, where

* key = the file name or the key you provided on the command line, and
* value = the file contents or the literal value you provided on the command line.

You can use [**kubectl describe**](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#describe) or [**kubectl get**](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#get) to retrieve information about a ConfigMap.

### Create ConfigMaps from directories

You can use **kubectl create configmap** to create a ConfigMap from multiple files in the same directory.

For example:

**mkdir -p configure-pod-container/configmap/kubectl/**

**wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/game.properties -O configure-pod-container/configmap/kubectl/game.properties**

**wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/ui.properties -O configure-pod-container/configmap/kubectl/ui.properties**

**kubectl create configmap game-config --from-file=configure-pod-container/configmap/kubectl/**

combines the contents of the **configure-pod-container/configmap/kubectl/** directory

**ls configure-pod-container/configmap/kubectl/**

**game.properties**

**ui.properties**

into the following ConfigMap:

**kubectl describe configmaps game-config**

**Name: game-config**

**Namespace: default**

**Labels: <none>**

**Annotations: <none>**

**Data**

**====**

**game.properties: 158 bytes**

**ui.properties: 83 bytes**

The **game.properties** and **ui.properties** files in the **configure-pod-container/configmap/kubectl/** directory are represented in the **data** section of the ConfigMap.

**kubectl get configmaps game-config -o yaml**

**apiVersion: v1**

**data:**

**game.properties: *|***

***enemies=aliens***

***lives=3***

***enemies.cheat=true***

***enemies.cheat.level=noGoodRotten***

***secret.code.passphrase=UUDDLRLRBABAS***

***secret.code.allowed=true***

***secret.code.lives=30***

**ui.properties: *|***

***color.good=purple***

***color.bad=yellow***

***allow.textmode=true***

***how.nice.to.look=fairlyNice***

**kind: ConfigMap**

**metadata:**

**creationTimestamp: 2016-02-18T18:52:05Z**

**name: game-config**

**namespace: default**

**resourceVersion: "516"**

**selfLink: /api/v1/namespaces/default/configmaps/game-config**

**uid: b4952dc3-d670-11e5-8cd0-68f728db1985**

### Create ConfigMaps from files

You can use **kubectl create configmap** to create a ConfigMap from an individual file, or from multiple files.

For example,

**kubectl create configmap game-config-2 --from-file=configure-pod-container/configmap/kubectl/game.properties**

would produce the following ConfigMap:

**kubectl describe configmaps game-config-2**

**Name: game-config-2**

**Namespace: default**

**Labels: <none>**

**Annotations: <none>**

**Data**

**====**

**game.properties: 158 bytes**

You can pass in the **--from-file** argument multiple times to create a ConfigMap from multiple data sources.

**kubectl create configmap game-config-2 --from-file=configure-pod-container/configmap/kubectl/game.properties --from-file=configure-pod-container/configmap/kubectl/ui.properties**

**kubectl describe configmaps game-config-2**

**Name: game-config-2**

**Namespace: default**

**Labels: <none>**

**Annotations: <none>**

**Data**

**====**

**game.properties: 158 bytes**

**ui.properties: 83 bytes**

Use the option **--from-env-file** to create a ConfigMap from an env-file, for example:

***# Env-files contain a list of environment variables.***

***# These syntax rules apply:***

***# Each line in an env file has to be in VAR=VAL format.***

***# Lines beginning with # (i.e. comments) are ignored.***

***# Blank lines are ignored.***

***# There is no special handling of quotation marks (i.e. they will be part of the ConfigMap value)).***

**wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/game-env-file.properties -O configure-pod-container/configmap/kubectl/game-env-file.properties**

**cat configure-pod-container/configmap/kubectl/game-env-file.properties**

**enemies=aliens**

**lives=3**

**allowed="true"**

**# This comment and the empty line above it are ignored**

**kubectl create configmap game-config-env-file \**

**--from-env-file=configure-pod-container/configmap/kubectl/game-env-file.properties**

would produce the following ConfigMap:

**kubectl get configmap game-config-env-file -o yaml**

**apiVersion: v1**

**data:**

**allowed: '"true"'**

**enemies: aliens**

**lives: "3"**

**kind: ConfigMap**

**metadata:**

**creationTimestamp: 2017-12-27T18:36:28Z**

**name: game-config-env-file**

**namespace: default**

**resourceVersion: "809965"**

**selfLink: /api/v1/namespaces/default/configmaps/game-config-env-file**

**uid: d9d1ca5b-eb34-11e7-887b-42010a8002b8**

When passing **--from-env-file** multiple times to create a ConfigMap from multiple data sources, only the last env-file is used:

**wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/ui-env-file.properties -O configure-pod-container/configmap/kubectl/ui-env-file.properties**

**kubectl create configmap config-multi-env-files \**

**--from-env-file=configure-pod-container/configmap/kubectl/game-env-file.properties \**

**--from-env-file=configure-pod-container/configmap/kubectl/ui-env-file.properties**

would produce the following ConfigMap:

**kubectl get configmap config-multi-env-files -o yaml**

**apiVersion: v1**

**data:**

**color: purple**

**how: fairlyNice**

**textmode: "true"**

**kind: ConfigMap**

**metadata:**

**creationTimestamp: 2017-12-27T18:38:34Z**

**name: config-multi-env-files**

**namespace: default**

**resourceVersion: "810136"**

**selfLink: /api/v1/namespaces/default/configmaps/config-multi-env-files**

**uid: 252c4572-eb35-11e7-887b-42010a8002b8**

#### Define the key to use when creating a ConfigMap from a file

You can define a key other than the file name to use in the **data** section of your ConfigMap when using the **--from-file** argument:

**kubectl create configmap game-config-3 --from-file=<my-key-name>=<path-to-file>**

where **<my-key-name>** is the key you want to use in the ConfigMap and **<path-to-file>** is the location of the data source file you want the key to represent.

For example:

**kubectl create configmap game-config-3 --from-file=game-special-key=configure-pod-container/configmap/kubectl/game.properties**

**kubectl get configmaps game-config-3 -o yaml**

**apiVersion: v1**

**data:**

**game-special-key: *|***

***enemies=aliens***

***lives=3***

***enemies.cheat=true***

***enemies.cheat.level=noGoodRotten***

***secret.code.passphrase=UUDDLRLRBABAS***

***secret.code.allowed=true***

***secret.code.lives=30***

**kind: ConfigMap**

**metadata:**

**creationTimestamp: 2016-02-18T18:54:22Z**

**name: game-config-3**

**namespace: default**

**resourceVersion: "530"**

**selfLink: /api/v1/namespaces/default/configmaps/game-config-3**

**uid: 05f8da22-d671-11e5-8cd0-68f728db1985**

### Create ConfigMaps from literal values

You can use **kubectl create configmap** with the **--from-literal** argument to define a literal value from the command line:

**kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm**

You can pass in multiple key-value pairs. Each pair provided on the command line is represented as a separate entry in the **data** section of the ConfigMap.

**kubectl get configmaps special-config -o yaml**

**apiVersion: v1**

**data:**

**special.how: very**

**special.type: charm**

**kind: ConfigMap**

**metadata:**

**creationTimestamp: 2016-02-18T19:14:38Z**

**name: special-config**

**namespace: default**

**resourceVersion: "651"**

**selfLink: /api/v1/namespaces/default/configmaps/special-config**

**uid: dadce046-d673-11e5-8cd0-68f728db1985**

## Define container environment variables using ConfigMap data

### Define a container environment variable with data from a single ConfigMap

1. Define an environment variable as a key-value pair in a ConfigMap:

**kubectl create configmap special-config --from-literal=special.how=very**

1. Assign the **special.how** value defined in the ConfigMap to the **SPECIAL\_LEVEL\_KEY** environment variable in the Pod specification.

**kubectl edit pod dapi-test-pod**

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: dapi-test-pod**

**spec:**

**containers:**

**- name: test-container**

**image: k8s.gcr.io/busybox**

**command: [ "/bin/sh", "-c", "env" ]**

**env:**

***# Define the environment variable***

**- name: SPECIAL\_LEVEL\_KEY**

**valueFrom:**

**configMapKeyRef:**

***# The ConfigMap containing the value you want to assign to SPECIAL\_LEVEL\_KEY***

**name: special-config**

***# Specify the key associated with the value***

**key: special.how**

**restartPolicy: Never**

1. Save the changes to the Pod specification. Now, the Pod’s output includes **SPECIAL\_LEVEL\_KEY=very**.

### Define container environment variables with data from multiple ConfigMaps

1. As with the previous example, create the ConfigMaps first.
2. **apiVersion: v1**
3. **kind: ConfigMap**
4. **metadata:**
5. **name: special-config**
6. **namespace: default**
7. **data:**

**special.how: very**

**apiVersion: v1**

**kind: ConfigMap**

**metadata:**

**name: env-config**

**namespace: default**

**data:**

**log\_level: INFO**

1. Define the environment variables in the Pod specification.
2. **apiVersion: v1**
3. **kind: Pod**
4. **metadata:**
5. **name: dapi-test-pod**
6. **spec:**
7. **containers:**
8. **- name: test-container**
9. **image: k8s.gcr.io/busybox**
10. **command: [ "/bin/sh", "-c", "env" ]**
11. **env:**
12. **- name: SPECIAL\_LEVEL\_KEY**
13. **valueFrom:**
14. **configMapKeyRef:**
15. **name: special-config**
16. **key: special.how**
17. **- name: LOG\_LEVEL**
18. **valueFrom:**
19. **configMapKeyRef:**
20. **name: env-config**
21. **key: log\_level**

**restartPolicy: Never**

1. Save the changes to the Pod specification. Now, the Pod’s output includes **SPECIAL\_LEVEL\_KEY=very** and **LOG\_LEVEL=INFO**.

## Configure all key-value pairs in a ConfigMap as container environment variables

**Note:** This functionality is available in Kubernetes v1.6 and later.

1. Create a ConfigMap containing multiple key-value pairs.
2. **apiVersion: v1**
3. **kind: ConfigMap**
4. **metadata:**
5. **name: special-config**
6. **namespace: default**
7. **data:**
8. **SPECIAL\_LEVEL: very**

**SPECIAL\_TYPE: charm**

1. Use **envFrom** to define all of the ConfigMap’s data as container environment variables. The key from the ConfigMap becomes the environment variable name in the Pod.
2. **apiVersion: v1**
3. **kind: Pod**
4. **metadata:**
5. **name: dapi-test-pod**
6. **spec:**
7. **containers:**
8. **- name: test-container**
9. **image: k8s.gcr.io/busybox**
10. **command: [ "/bin/sh", "-c", "env" ]**
11. **envFrom:**
12. **- configMapRef:**
13. **name: special-config**

**restartPolicy: Never**

1. Save the changes to the Pod specification. Now, the Pod’s output includes **SPECIAL\_LEVEL=very** and **SPECIAL\_TYPE=charm**.

## Use ConfigMap-defined environment variables in Pod commands

You can use ConfigMap-defined environment variables in the **command** section of the Pod specification using the **$(VAR\_NAME)** Kubernetes substitution syntax.

For example:

The following Pod specification

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: dapi-test-pod**

**spec:**

**containers:**

**- name: test-container**

**image: k8s.gcr.io/busybox**

**command: [ "/bin/sh", "-c", "echo $(SPECIAL\_LEVEL\_KEY) $(SPECIAL\_TYPE\_KEY)" ]**

**env:**

**- name: SPECIAL\_LEVEL\_KEY**

**valueFrom:**

**configMapKeyRef:**

**name: special-config**

**key: SPECIAL\_LEVEL**

**- name: SPECIAL\_TYPE\_KEY**

**valueFrom:**

**configMapKeyRef:**

**name: special-config**

**key: SPECIAL\_TYPE**

**restartPolicy: Never**

produces the following output in the **test-container** container:

**very charm**

## Add ConfigMap data to a Volume

As explained in [Create ConfigMaps from files](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#create-configmaps-from-files), when you create a ConfigMap using **--from-file**, the filename becomes a key stored in the **data** section of the ConfigMap. The file contents become the key’s value.

The examples in this section refer to a ConfigMap named special-config, shown below.

**apiVersion: v1**

**kind: ConfigMap**

**metadata:**

**name: special-config**

**namespace: default**

**data:**

**special.level: very**

**special.type: charm**

### Populate a Volume with data stored in a ConfigMap

Add the ConfigMap name under the **volumes** section of the Pod specification. This adds the ConfigMap data to the directory specified as **volumeMounts.mountPath** (in this case, **/etc/config**). The **command**section references the **special.level** item stored in the ConfigMap.

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: dapi-test-pod**

**spec:**

**containers:**

**- name: test-container**

**image: k8s.gcr.io/busybox**

**command: [ "/bin/sh", "-c", "ls /etc/config/" ]**

**volumeMounts:**

**- name: config-volume**

**mountPath: /etc/config**

**volumes:**

**- name: config-volume**

**configMap:**

***# Provide the name of the ConfigMap containing the files you want***

***# to add to the container***

**name: special-config**

**restartPolicy: Never**

When the pod runs, the command (**"ls /etc/config/"**) produces the output below:

**special.level**

**special.type**

**Caution:** If there are some files in the **/etc/config/** directory, they will be deleted.

### Add ConfigMap data to a specific path in the Volume

Use the **path** field to specify the desired file path for specific ConfigMap items. In this case, the **special.level** item will be mounted in the **config-volume** volume at **/etc/config/keys**.

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: dapi-test-pod**

**spec:**

**containers:**

**- name: test-container**

**image: k8s.gcr.io/busybox**

**command: [ "/bin/sh","-c","cat /etc/config/keys" ]**

**volumeMounts:**

**- name: config-volume**

**mountPath: /etc/config**

**volumes:**

**- name: config-volume**

**configMap:**

**name: special-config**

**items:**

**- key: special.level**

**path: keys**

**restartPolicy: Never**

When the pod runs, the command (**"cat /etc/config/keys"**) produces the output below:

**very**

### Project keys to specific paths and file permissions

You can project keys to specific paths and specific permissions on a per-file basis. The [Secrets](https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets-as-files-from-a-pod) user guide explains the syntax.

### Mounted ConfigMaps are updated automatically

When a ConfigMap already being consumed in a volume is updated, projected keys are eventually updated as well. Kubelet is checking whether the mounted ConfigMap is fresh on every periodic sync. However, it is using its local ttl-based cache for getting the current value of the ConfigMap. As a result, the total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the pod can be as long as kubelet sync period + ttl of ConfigMaps cache in kubelet.

**Note:** A container using a ConfigMap as a [subPath](https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath) volume will not receive ConfigMap updates.

## Understanding ConfigMaps and Pods

The ConfigMap API resource stores configuration data as key-value pairs. The data can be consumed in pods or provide the configurations for system components such as controllers. ConfigMap is similar to [Secrets](https://kubernetes.io/docs/concepts/configuration/secret/), but provides a means of working with strings that don’t contain sensitive information. Users and system components alike can store configuration data in ConfigMap.

**Note:** ConfigMaps should reference properties files, not replace them. Think of the ConfigMap as representing something similar to the Linux **/etc** directory and its contents. For example, if you create a [Kubernetes Volume](https://kubernetes.io/docs/concepts/storage/volumes/)from a ConfigMap, each data item in the ConfigMap is represented by an individual file in the volume.

The ConfigMap’s **data** field contains the configuration data. As shown in the example below, this can be simple – like individual properties defined using **--from-literal** – or complex – like configuration files or JSON blobs defined using **--from-file**.

**kind: ConfigMap**

**apiVersion: v1**

**metadata:**

**creationTimestamp: 2016-02-18T19:14:38Z**

**name: example-config**

**namespace: default**

**data:**

***# example of a simple property defined using --from-literal***

**example.property.1: hello**

**example.property.2: world**

***# example of a complex property defined using --from-file***

**example.property.file: |-**

**property.1=value-1**

**property.2=value-2**

**property.3=value-3**

### Restrictions

* You must create a ConfigMap before referencing it in a Pod specification (unless you mark the ConfigMap as “optional”). If you reference a ConfigMap that doesn’t exist, the Pod won’t start. Likewise, references to keys that don’t exist in the ConfigMap will prevent the pod from starting.
* If you use **envFrom** to define environment variables from ConfigMaps, keys that are considered invalid will be skipped. The pod will be allowed to start, but the invalid names will be recorded in the event log (**InvalidVariableNames**). The log message lists each skipped key. For example:

**kubectl get events**

**LASTSEEN FIRSTSEEN COUNT NAME KIND SUBOBJECT TYPE REASON SOURCE MESSAGE**

**0s 0s 1 dapi-test-pod Pod Warning InvalidEnvironmentVariableNames {kubelet, 127.0.0.1} Keys [1badkey, 2alsobad] from the EnvFrom configMap default/myconfig were skipped since they are considered invalid environment variable names.**

* ConfigMaps reside in a specific [namespace](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/). A ConfigMap can only be referenced by pods residing in the same namespace.
* Kubelet doesn’t support the use of ConfigMaps for pods not found on the API server. This includes pods created via the Kubelet’s –manifest-url flag, –config flag, or the Kubelet REST API.

### How to Remove Unused or Dangling Images, Containers, Volumes, and Networks

You can delete all dangling and unreferenced data such as containers stopped, images without containers, with this single command. By default, volumes are not removed, to prevent vital data from being deleted if there is currently no container using the volume.

Docker containers are temporary instances that are usually isolated from the rest of the computing environment. While this makes containers very portable, it also leads to the [common question of "How do I save my data?"](https://github.com/rocker-org/rocker/issues/123). Here we present several strategies so you can choose what works best for you.

**Using docker commit**

docker commit allows you to save a snapshot of your container as a docker image so you can return to it later. Like any docker image, these can be moved around to a different machines. Docker's git-like features really shine here -- you can roll back to previous commits, and pushing images with docker push is really fast since it pushes only differences.

Quick overview:

* On the host machine that is running docker, look up the name or container id of the running container using docker ps. (You can also assign your own choice of name to the container when calling docker run and then use that).
* Save the running container as a docker image, e.g. docker commit <container-id> username/imagename. Optionally you can include commit messages with -m. Once the container is committed, you can now stop or remove the container without losing data.
* Push the container to the Docker Hub: docker push username/imagename. Be sure to use a private image (either on the Hub or by running a private registry) if necessary: just create the private image name on the Hub before pushing. (Alternatively you can save the container as a tarball with docker save and download that for future use. This approach does not benefit from transferring only the changed layers, so should be avoided in favor of docker push/pull if possible).

NOTE: If you start an instance with a linked volume, docker commit will not capture changes to that volume.

**Using linked volumes**

An alternative to using docker commit is to just use linked volumes, as described in more detail on the wiki page [Sharing files with the host machine](https://github.com/rocker-org/rocker/wiki/Sharing-files-with-host-machine).

By sharing volumes with the host machine, you never have to remember steps like docker commitas the files will always persist locally.

**Other ways**

* Use docker cp to copy data from the container onto the host machine.
* Use 'download' option from the file pane of the RStudio-server
* Use git to commit your changed files and push to a server
* Use and commit a "volume-container". This allows you to have separate docker containers for your files and for the computational environment.

$ docker system prune

Docker provides 2 ways to backup and sync container data on the local machine i.e. *volume* and *mount*. Both behave in the same way except for a few things I noticed:

1. A volume always keeps data in /var/lib/docker/volumes, while mount points can be created wherever we want.
2. If a container which is assigned a mount point is also assigned a volume then all data from the mount point is copied to the volume automatically, while the opposite is not true.
3. We cannot describe a mount point in a Dockerfile but can give volumes in a Dockerfile.

Ok, so we can say there are some advantages and disadvantages of methodology but are there still some classification or differences in term of optimization.

Please provide an explained answer.

There are actually three types of volumes:

* Host Volume: what you refer to as a mount in a container, the more common term is a bind mount.
* Named Volume: any volume managed by docker which you give a name.
* Anonymous Volume: any volume without a source, docker will create this as a local volume with a long unique id, and it behaves as a named volume.

Volumes have a source and a target. The source identifies the type of volume, so a path (including the leading slash) to a file/directory results in a host volume. If you do not provide a source, you get the anonymous volumes. If you define a volume inside a Dockerfile, you cannot specify a source there, so by default docker will create anonymous volumes unless you direct it otherwise at runtime.

For each type, here are the pros/cons:

* Host:
  + Pro: easy to access the underlying files from the host
  + Con: uid/gid permission issues occur when container user's uid does not match the host gid
  + Con: data is not initialized
* Named:
  + Pro: easy to create an reuse between different containers/images. If you only give it a name with no other settings, the local driver will default to storing your data in /var/lib/docker/volumes which should only be accessible by root from outside of docker.
  + Pro: initializes content to the image contents when it is empty/new and the container is created. This initialization includes file owners and permissions from the image, which can resolve most uid/gid issues.
  + Pro: Can connect to anything that a mount command can, including a bind mount or NFS mount, with a local driver. Other drivers let you reference data in even more locations (e.g. cloud providers).
  + Con: managing content should be done via a container.
* Anonymous:
  + Pro: requires no planning to use
  + Con: data typically goes here to be lost since there is no mapping from the volume back to the container/image that created it. This is the worst way to store volumes in my opinion, and the reason that no one should ever define a volume inside their Dockerfile.

When possible, I use a named volume. The initialization of data and better handling of uid/gid issues trump the convenience of a host volume. If I really need access outside of docker directly to the data, then I try to use a named volume that points to a bind mount instead of the default local driver settings. A simple example of this is:

$ docker volume create --driver local \

--opt type=none \

--opt device=/home/user/test \

--opt o=bind \

test\_vol

For defining my volumes, since you do not want to do this in a Dockerfile, I use a docker-compose.yml and define my volumes in there. If it's deployed with swarm mode, I'll point to a NFS server with a named volume to allow the data to be reached as the containers migrate to different hosts. Otherwise it's a local named volume that can be easily used with docker-compose.

How to display the populate the values corresponding the values(state--cities likewise)

<https://wiki.jenkins.io/display/JENKINS/Extended+Choice+Parameter+plugin>

[Extended Choice Parameter plugin](https://wiki.jenkins.io/display/JENKINS/Extended+Choice+Parameter+plugin)

**Version 0.20 (Jan 07, 2012)**

* New field to configure number of items visible in selectbox without scrolling
* New 'checkbox' and 'radio button' parameter types added
* changed all validation error checks to warnings
* added ability to specify an url instead of absolute directory path for property files
* ability to specify property references for values, for example
  + prop1=a,b,c,d,e
  + prop2=${prop1},f,g,h  --(prop2 will now evaluate to a,b,c,d,e,f,g,h)

##### Version 0.5 (Jan 10, 2012)

* Use a dropdown when using single select mode
* Trim properties
* When using a properties file, do not load the file content when editing the job
* Load the 'latest' property at each build
* Added some validation for properties file names

##### Version 0.1 (Jul 14, 2010)

* Initial release

This plugin allows single select and multi select build parameters to be configured.

Here is a screenshot of the configuration page.

The 'value' field is a comma separated list of values for the single select or multi-select box.  
This field can be left blank if the comma separated values need to be picked up from a properties file.  
In that case the fields 'Property File' and 'Property Key' need to be filled in.

The 'default value' field is used to set the initial selection of the single-select or mult-select box.  
in case of the multi-select box default value can be a comma separated string.  
This field can be left blank if the default value needs to be picked up from a properties file.  
In that case the fields 'Default Property File' and 'Default Property Key' need to be filled in.

Note: Neither "Property File" or "Default Property File" support referencing environment variables in their values. Thus, their values should be absolute paths specified without using environment variables.

The property file should be evaluated during the build step too, not only if configuring.

I'm seeing the same thing, with regards to setting the default for the list.  The list itself does appear to be updating with new values.

If I change the default in the properties file, it only changes the default parameter if I go to the configure page, and save it.

Simply going to "Build Now" does not update the default choice in the list.  (I've only tried this with the single select, not the multi select settings)

Also, I have tried this using the same properties file for both the default and the list, as well as having them in separate files.

How to know how many volumes has been mounted in docker

When using docker images from registries, I often need to see the volumes created by the image's containers.

With docker 1.10, you now have new commands for data-volume containers.  
(for regular containers, see the next section, for docker 1.8+):

* [docker volume ls](https://docs.docker.com/engine/reference/commandline/volume_ls/)
* [docker volume inspect](https://docs.docker.com/engine/reference/commandline/volume_inspect/)

## Options

| Name, shorthand | Default | | Description |
| --- | --- | --- | --- |
| --filter , -f |  | | Provide filter values (e.g. ‘dangling=true’) |
| --format |  | | Pretty-print volumes using a Go template |
| --quiet , -q |  | | Only display volume names |
| Command | | Description | |
| [docker volume create](https://docs.docker.com/engine/reference/commandline/volume_create/) | | Create a volume | |
| [docker volume inspect](https://docs.docker.com/engine/reference/commandline/volume_inspect/) | | Display detailed information on one or more volumes | |
| [docker volume ls](https://docs.docker.com/engine/reference/commandline/volume_ls/) | | List volumes | |
| [docker volume prune](https://docs.docker.com/engine/reference/commandline/volume_prune/) | | Remove all unused local volumes | |
| [docker volume rm](https://docs.docker.com/engine/reference/commandline/volume_rm/) | | Remove one or more volumes | |

**What are the series of steps that happen when a URL is requested from the address field of a browser?**

**1. You type maps.google.com into the address bar of your browser.**

**2. The browser checks the cache for a DNS record to find the corresponding IP address of maps.google.com.**

DNS(Domain Name System) is a database that maintains the name of the website (URL) and the particular IP address it links to. Every single URL on the internet has a unique IP address assigned to it. The IP address belongs to the computer which hosts the server of the website we are requesting to access. For an example, [www.google.com](http://www.google.com/) has an IP address of 209.85.227.104. So if you’d like you can reach [www.google.com](http://www.google.com/) by typing [http://209.85.227.104](http://209.85.227.104/) on your browser. DNS is a list of URLs and their IP addresses just like how a phone book is a list of names and their corresponding phone numbers.

The main purpose of DNS is human-friendly navigation. You can easily access a website by typing the correct IP address for it on your browser but imagine having to remember different sets of numbers for all the websites we regularly access? Therefore, it is easier to remember the name of the website using an URL and let DNS do the work for us with mapping it to the correct IP.

In order to find the DNS record, the browser checks four caches.

● First, it checks the browser cache. The browser maintains a repository of DNS records for a fixed duration for websites you have previously visited. So, it is the first place to run a DNS query.

● Second, the browser checks the OS cache. If it is not found in the browser cache, the browser would make a system call (i.e. *gethostname* on Windows) to your underlying computer OS to fetch the record since the OS also maintains a cache of DNS records.

● Third, it checks the router cache. If it’s not found on your computer, the browser would communicate with the router that maintains its’ own cache of DNS records.

● Fourth, it checks the ISP cache. If all steps fail, the browser would move on to the ISP. Your ISP maintains its’ own DNS server which includes a cache of DNS records which the browser would check with the last hope of finding your requested URL.

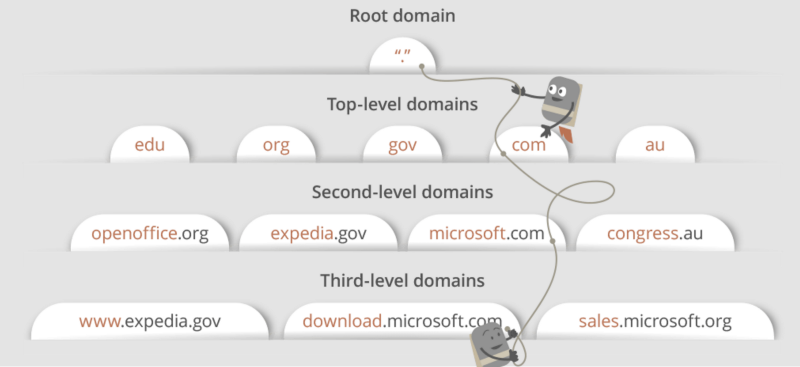
You may wonder why there are so many caches maintained at so many levels. Although our information being cached somewhere doesn’t make us feel very comfortable when it comes to privacy, caches are important for regulating network traffic and improving data transfer times.

**3. If the requested URL is not in the cache, ISP’s DNS server initiates a DNS query to find the IP address of the server that hosts maps.google.com.**

As mentioned earlier, in order for my computer to connect with the server that hosts maps.google.com, I need the IP address of maps.google.com. The purpose of a DNS query is to search multiple DNS servers on the internet until it finds the correct IP address for the website. This type of search is called a recursive search since the search will continue repeatedly from DNS server to DNS server until it either finds the IP address we need or returns an error response saying it was unable to find it.

In this situation, we would call the ISP’s DNS server a DNS recursor whose responsibility is to find the proper IP address of the intended domain name by asking other DNS servers on the internet for an answer. The other DNS servers are called name servers since they perform a DNS search based on the domain architecture of the website domain name.

Without further confusing you, I’d like to use the following diagram to explain the domain architecture.



<https://webhostinggeeks.com/guides/dns/>

Many website URLs we encounter today contain a third-level domain, a second-level domain, and a top-level domain. Each of these levels contains their own name server which is queried during the DNS lookup process.

For maps.google.com, first, the DNS recursor will contact the root name server. The root name server will redirect it to **.com** domain name server. **.com** name server will redirect it to **google.com** name server. **google.com** name server will find the matching IP address for maps.google.com in its’ DNS records and return it to your DNS recursor which will send it back to your browser.

These requests are sent using small data packets which contain information such as the content of the request and the IP address it is destined for (IP address of the DNS recursor). These packets travel through multiple networking equipment between the client and the server before it reaches the correct DNS server. This equipment use routing tables to figure out which way is the fastest possible way for the packet to reach its’ destination. If these packets get lost you’ll get a request failed error. Otherwise, they will reach the correct DNS server, grab the correct IP address, and come back to your browser.

**4. Browser initiates a TCP connection with the server.**

Once the browser receives the correct IP address it will build a connection with the server that matches IP address to transfer information. Browsers use internet protocols to build such connections. There are a number of different internet protocols which can be used but TCP is the most common protocol used for any type of HTTP request.

In order to transfer data packets between your computer(client) and the server, it is important to have a TCP connection established. This connection is established using a process called the TCP/IP three-way handshake. This is a three step process where the client and the server exchange SYN(synchronize) and ACK(acknowledge) messages to establish a connection.

1. Client machine sends a SYN packet to the server over the internet asking if it is open for new connections.

2. If the server has open ports that can accept and initiate new connections, it’ll respond with an ACKnowledgment of the SYN packet using a SYN/ACK packet.

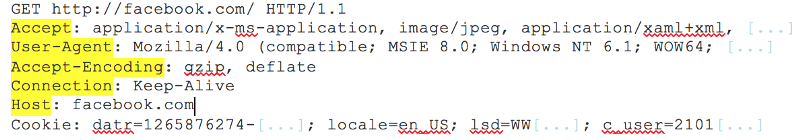
3. The client will receive the SYN/ACK packet from the server and will acknowledge it by sending an ACK packet.

Then a TCP connection is established for data transmission!

**5. The browser sends an HTTP request to the web server.**

Once the TCP connection is established, it is time to start transferring data! The browser will send a GET request asking for maps.google.com web page. If you’re entering credentials or submitting a form this could be a POST request. This request will also contain additional information such as browser identification (*User-Agent* header), types of requests that it will accept (*Accept*header), and connection headers asking it to keep the TCP connection alive for additional requests. It will also pass information taken from cookies the browser has in store for this domain.

Sample GET request (Headers are highlighted):



(If you’re curious about what’s going on behind the scenes you can use tools such as Firebug to take a look at HTTP requests. It is always fun to see the information passed between clients and servers without us knowing).

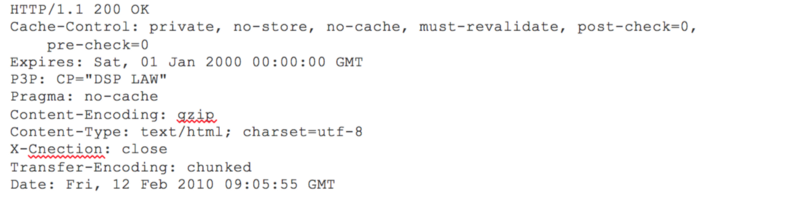
**6. The server handles the request and sends back a response.**

The server contains a web server (i.e Apache, IIS) which receives the request from the browser and passes it to a request handler to read and generate a response. The request handler is a program (written in ASP.NET, PHP, Ruby, etc.) that reads the request, its’ headers, and cookies to check what is being requested and also update the information on the server if needed. Then it will assemble a response in a particular format (JSON, XML, HTML).

**7. The server sends out an HTTP response.**

The server response contains the web page you requested as well as the status code, compression type (*Content-Encoding)*, how to cache the page (*Cache-Control*), any cookies to set, privacy information, etc.

Example HTTP server response:



If you look at the above response the first line shows a status code. This is quite important as it tells us the status of the response. There are five types of statuses detailed using a numerical code.

● 1xx indicates an informational message only

● 2xx indicates success of some kind

● 3xx redirects the client to another URL

● 4xx indicates an error on the client’s part

● 5xx indicates an error on the server’s part

So, if you encountered an error you can take a look at the HTTP response to check what type of status code you have received.

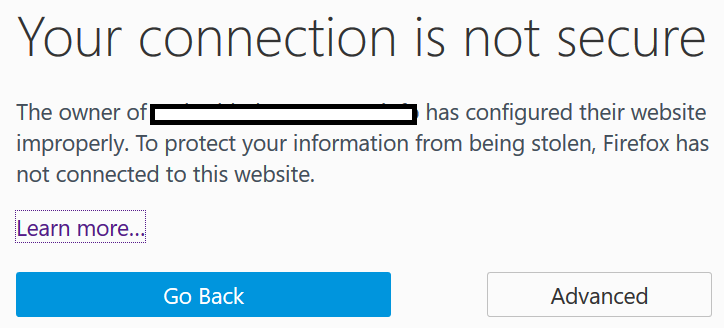
**8. The browser displays the HTML content (for HTML responses which is the most common).**

The browser displays the HTML content in phases. First, it will render the bare bone HTML skeleton. Then it will check the HTML tags and sends out GET requests for additional elements on the web page, such as images, CSS stylesheets, JavaScript files etc. These static files are cached by the browser so it doesn’t have to fetch them again the next time you visit the page. At the end, you’ll see maps.google.com appearing on your browser.

That’s it!

Although this seems like a very tedious prolonged process we know that it takes less than seconds for a web page to render after we hit enter on our keyboard. All of these steps happens within milliseconds before we could even notice. I sincerely hope this article helps you answer the question “What happens when you type an URL in the browser and press enter?”.

There are two kinds of SSL Certificates you can create for your own server: self-signed certificates and certificates that are signed by a Certificate Authority (CA). With a self-signed certificate, you always get a warning in the browser since the certificate is not trusted by the browser. However, a certificate signed by a CA that is trusted by the browser gives you the green bar. A browser trusts the CA if the CA's public root certificate is installed in the browser and/or computer you are using.



**Self-signed or non-trusted CA certificate error in the browser**

Being your own CA has the minor inconvenience that you must install the CA root certificate in all clients (browsers) that visit any of the sites with a certificate signed by your root CA. For this reason, being your own CA is mainly suitable for sites used by a small group of users. On the bright side, there are no costs associated with being your own CA.

If you have a small group of users using your secure site, you may provide a web page with information on how to install your public CA root certificate. As an example, we provide a CA test certificate and [a web page that explains how to install this CA test certificate](https://realtimelogic.com/downloads/root-certificate/).

You can find several tutorials on the internet that explain how to use the OpenSSL command line tool for setting up your own CA infrastructure. If you are new to X.509 certificate chain of trust, these tutorials will make your head spin. In addition, the OpenSSL command line tool is a bit cumbersome to use and gives difficult to understand error messages if you make mistakes.

To make it much easier to be your own CA, we created a web application that wraps around the OpenSSL command line tool. The web application includes a wizard making it very easy to create a root CA and to create any number of certificates signed by the root CA. The web application is self-contained, and you simply need to download the tool and run it on your computer.

Create a Certificate Signed by a Certificate Authority

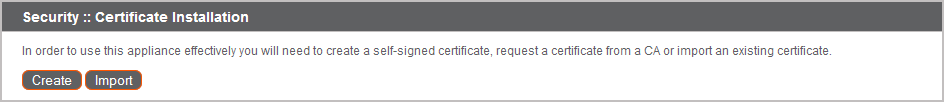
To have full functionality of the Bomgar software and to avoid security risks, it is very important that as soon as possible, you obtain a valid SSL certificate signed by a certificate authority (CA). While a CA-signed certificate is the best way to secure your site, you may need a self-signed certificate or an internally-signed certificate (see [**Create a Self-Signed Certificate**](https://www.bomgar.com/docs/remote-support/how-to/sslcertificates/create-self-signed.htm)).

To obtain a certificate signed by a certificate authority, you must first create a certificate signing request (CSR) from the /appliance interface of your Bomgar Appliance. You will then submit the request data to a certificate authority. Once the signed certificate is obtained, the Bomgar software should be updated.

Create the Certificate Signing Request

The first step is to create the CSR. The request data associated with the CSR contains the details about your organization and Bomgar site. This request data is submitted to your certificate authority for them to publicly certify your organization and Bomgar Appliance.

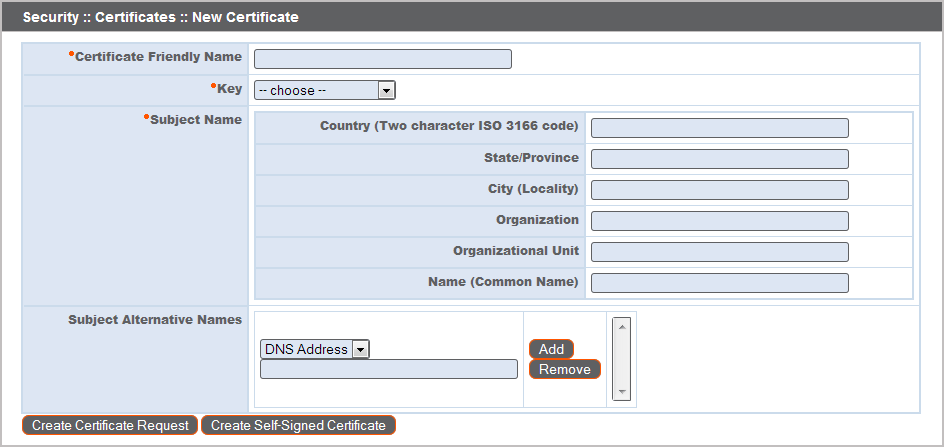
Certificates consist of a **friendly name**, **key**, **subject name**, and one or more **subject alternative names**. You must enter this information in the Bomgar /appliance web interface to create a certificate signing request.

[Security > Certificates](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-header_4-0.png)  
[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-installation_3-3.png)

1. Log into the /appliance web interface of your Bomgar Appliance and go to **Security > Certificates**.

**Note:** You will see a "Bomgar Appliance" certificate listed. This is a standard certificate which ships with all Bomgar appliances. Both the certificate and its warning should be ignored.

1. In the **Security :: Certificate Installation** section, click **Create**.

[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-new_3-3.png)

1. Create a descriptive title for **Certificate Friendly Name**. Examples could include your primary DNS name or the current month and year. This name helps you identify your certificate request on your Bomgar Appliance **Security > Certificates** page.
2. Choose a key size from the **Key** dropdown. Verify with your certificate authority which key strengths they support. Larger key sizes normally require more processing overhead and may not be supported by older systems. However, smaller key sizes are likely to become obsolete or insecure sooner than larger ones.
3. The **Subject Name** consists of the contact information for the organization and department creating the certificate along with the name of the certificate.
   1. Enter your organization's two-character **Country** code. If you are unsure of your country code, please visit [www.iso.org/iso/home/standards/country\_codes.htm](http://www.iso.org/iso/home/standards/country_codes.htm).
   2. Enter your **State/Province** name if applicable. Enter the full state name, as some certificate authorities will not accept a state abbreviation.
   3. Enter your **City (Locality)**.
   4. In **Organization**, provide the name of your company.
   5. **Organizational Unit** is normally the group or department within the organization managing the certificate and/or the Bomgar deployment for the organization.
   6. For **Name (Common Name)**, enter a title for your certificate. In many cases, this should be simply a human-readable label. It is not recommended that you use your DNS name as the common name. However, some certificate authorities may require that you do use your fully qualified DNS name for backward compatibility. Contact your certificate authority for details. This name must be unique to differentiate the certificate from others on the network. Be aware that this network could include the public internet.
4. In **Subject Alternative Names**, list the fully qualified domain name for each DNS A-record which resolves to your Bomgar Appliance (e.g., support.example.com). After entering each subject alternative name (SAN), click the **Add**button.

**Note:** If you entered the fully qualified domain name as your subject's common name, you must re-enter this as the first SAN entry. If you wish to use IP addresses instead of DNS names, contact Bomgar Technical Supportfirst.

A SAN lets you protect multiple hostnames with a single SSL certificate. A DNS address could be a fully qualified domain name, such as support.example.com, or it could be a wildcard domain name, such as \*.example.com. A wildcard domain name covers multiple subdomains, such as support.example.com, remote.example.com, and so forth. If you are going to use multiple hostnames for your site that are not covered by a wildcard certificate, be sure to define those as additional SANs.

**Note:** If you plan to use multiple Bomgar Appliances in an Atlas setup, it is recommended that you use a wildcard certificate that covers both your Bomgar site hostname and each traffic node hostname. If you do not use a wildcard certificate, adding traffic nodes that use different certificates will require a rebuild of the Bomgar software.

1. Click **Create Certificate Request** and wait for the page to refresh.
2. The certificate request should now appear in the **Security :: Certificate Requests** section.

Submit the Certificate Signing Request

Once the certificate signing request has been created, you must submit it to a certificate authority for certification. You can obtain an SSL certificate from a commercial or public certificate authority or from an internal CA server if your organization uses one. Bomgar does not require or recommend any specific certificate authority, but these are some of the most well known.

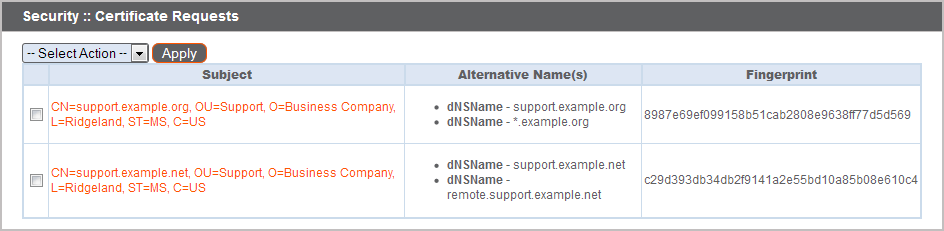
* Comodo ([www.comodo.com](https://www.comodo.com/)) - As of 24 February 2015, Comodo is the largest issuer of SSL certificates.
* Digicert ([www.digicert.com](https://www.digicert.com/)) - Digicert is a US-based certificate authority that has been in business for over a decade.
* GeoTrust, Inc. ([www.geotrust.com](https://www.geotrust.com/)) - GeoTrust is the world's second largest digital certificate provider.
* GoDaddy SSL ([www.godaddy.com/ssl/ssl-certificates.aspx](https://www.godaddy.com/ssl/ssl-certificates.aspx)) - GoDaddy is the world's largest domain name registrar, and their SSL certificates are widely used.
* Symantec SSL ([www.symantec.com/ssl-certificates](https://www.symantec.com/ssl-certificates)) - 97 of the world's 100 largest financial institutions and 75 percent of the 500 biggest e-commerce sites in North America use SSL certificates from Symantec.

Once you have selected a certificate authority, you must purchase a certificate from them. Bomgar does not require any special type of certificate. Bomgar accepts wildcard certificates, subject alternative name (SAN) certificates, unified communications (UC) certificates, extended validation (EV) certificates, and so forth, as well as standard certificates.

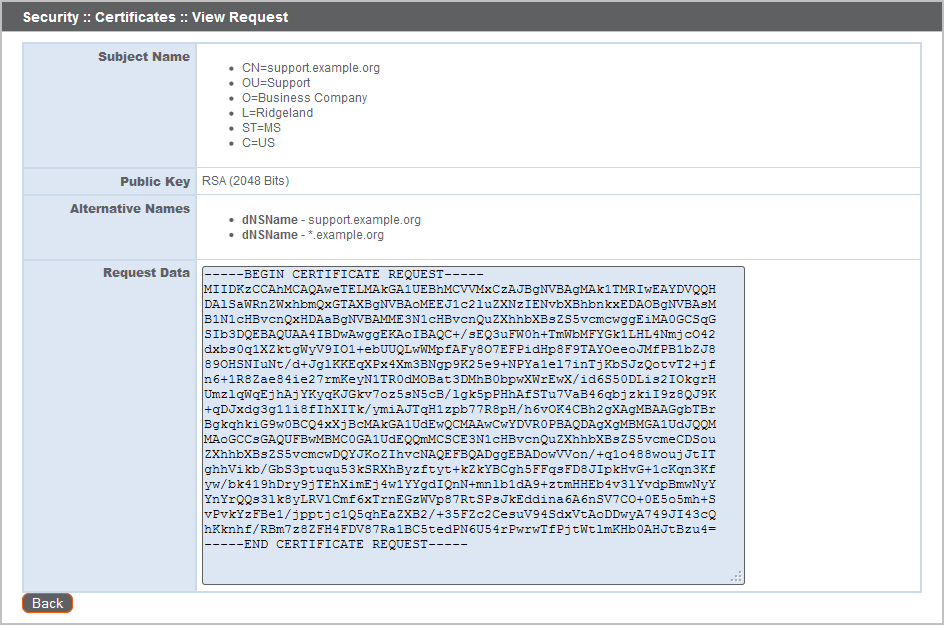
During or after the purchase, you will be prompted to upload or copy/paste your request data. The certificate authority should give you instructions for doing so. To retrieve your request data from Bomgar, take these steps:

[Security > Certificates](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-header_4-0.png)

1. When prompted to submit the request information, log into the /appliance interface of your Bomgar Appliance. Go to **Security > Certificates**.

[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-requests_3-3.png)

1. In the **Security :: Certificate Requests** section, click the subject of your certificate request.

[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-viewrequest_3-3.png)

1. Select and copy the **Request Data**, and then submit this information to your certificate authority. Some certificate authorities require you to specify the type of server the certificate is for. If this is a required field, submit that the server is Apache-compatible. If given more than one Apache type as options, select Apache/ModSSL.

Import the Certificate

Once the certificate authority has the request data, they will review it and sign it. After the certificate authority has signed the certificate, they will send it back to you, often with the root and/or intermediate certificate files. All these together constitute your certificate chain. The CA or Issuing Authority issues multiple certificates in a certificate chain, proving that your site's certificate was issued by the CA. This proof is validated using a public and private key pair. The public key, available to all of your site visitors, must validate the private key in order to verify the authenticity of the certificate chain. The certificate chain typically consists of three types of certificate:

* Root Certificate – The certificate that identifies the certificate authority.
* Intermediate Root Certificates – Certificates digitally signed and issued by an Intermediate CA, also called a Signing CA or Subordinate CA.
* Identity Certificate – A certificate that links a public key value to a real-world entity such as a person, a computer, or a web server.

All of these certificate files must be imported to your Bomgar Appliance before it will be completely operational. The certificate chain will be sent in one of multiple certificate file formats. The following certificate formats are acceptable:

* DER-encoded X.509 certificate (.cer, .der, .crt)
* PEM-wrapped DER-encoded X.509 certificate (.pem, .crt, .b64)
* DER-encoded PKCS #7 certificates (.p7, .p7b, .p7c)

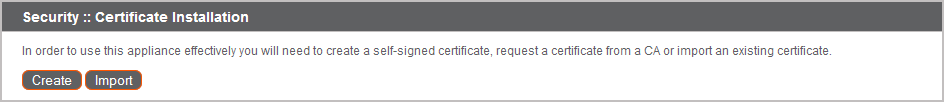
You must download all of the certificate files in your certificate chain to a secure location. This location should be accessible from the same computer used to access the /appliance interface. Sometimes the CA's certificate download interface prompts for a server type. If prompted to select a server type, select Apache. If given more than one Apache type as options, select Apache/ModSSL.

Many certificate authorities do not send the root certificate of your certificate chain. Bomgar requires this root certificate to function properly. If no links were provided to obtain the root certificate, then it is suggested that the CA be contacted for assistance. If this is impracticle for any reason, it should be possible to find the correct root certificate in your CA's online root certificate repository. Some of the major repositories are these:

* Comodo > Repository > Root Certificates ([www.comodo.com/about/comodo-agreements.php](https://www.comodo.com/about/comodo-agreements.php))
* DigiCert Trusted Root Authority Certificates ([www.digicert.com/digicert-root-certificates.htm](https://www.digicert.com/digicert-root-certificates.htm))
* GeoTrust Root Certificates ([www.geotrust.com/resources/root-certificates](https://www.geotrust.com/resources/root-certificates))
* GoDaddy > Repository ([certs.godaddy.com/repository](https://certs.godaddy.com/repository))
* Symantec > Licensing and Use of Root Certificates ([www.symantec.com/page.jsp?id=roots](https://www.symantec.com/page.jsp?id=roots))

To identify which root is appropriate for your certificate chain, you should contact your certificate authority. However, it is also possible on most systems to open your certificate file on the local system and check the certificate chain from there. For instance, in Windows 7, the certificate chain is shown under the **Certification Path** tab of the certificate file, and the root certificate is listed at the top. Opening the root certificate here normally allows you to identify the approprate root on the CA's online repository.

Once you have downloaded all the certificate files for your certificate chain, you must import these files to your Bomgar Appliance.

[Security > Certificates](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-header_4-0.png)  
[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-installation_3-3.png)

1. Log into the /appliance interface of your Bomgar Appliance. Go to **Security > Certificates**.
2. In the **Security :: Certificate Installation** section, click the **Import** button.

[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-import_3-3.png)

1. Browse to your certificate file and click **Upload**. Then upload the intermediate certificate files and root certificate file used by the CA.

Your signed certificate should now appear in the **Security :: Certificates** section. If the new certificate shows a warning beneath its name, this typically means the intermediate and/or root certificates from the CA have not been imported. The components of the certificate chain can be identified as follows:

* The Bomgar server certificate has an **Issued To** field and/or an **Alternative Name(s)** field matching the Bomgar Appliance's URL (e.g., support.example.com).
* Intermediate certificates have different **Issued To** and **Issued By** fields, neither of which is a URL.
* The root certificate has identical values for the **Issued To** and **Issued By** fields, neither of which is a URL.

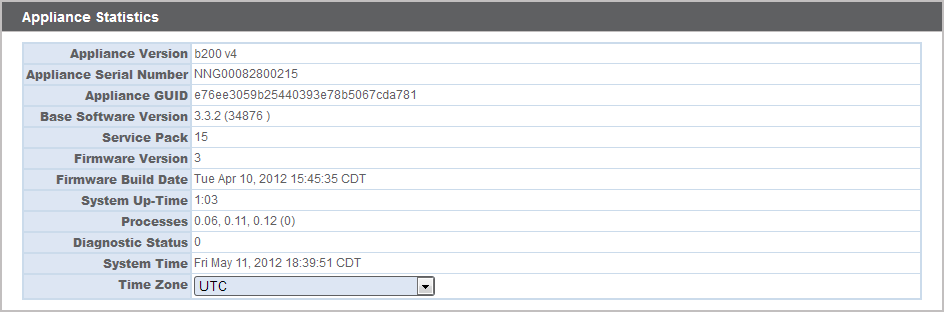
If any of these are missing, contact your certificate authority and/or follow the instructions given above in this guide to locate, download, and import the missing certificates.

Update the Bomgar Appliance

To insure the reliability of your client software, Bomgar Technical Support builds your root certificate into your software. Therefore, any time you import a new root certificate to your Bomgar Appliance, you must send to Bomgar Technical Support a copy of the new SSL certificate and also a screenshot of your **Status > Basics** page to identify the appliance being updated.

**IMPORTANT!**

Do NOT send your private key file (which ends in **.p12**) to Bomgar Technical Support. This key is private because it allows the owner to authenticate your Bomgar Appliance's identity. Ensure that the private key and its passphrase are kept in a secure, well-documented location on your private network. If this key is ever exposed to the public (via email, for instance), the security of your appliance is compromised.

[Status > Basics](https://www.bomgar.com/docs/remote-support/resources/images/appliance/status-basics-header_4-0.png)  
[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/status-basics-stats_3-3.png)

1. Go to **/appliance > Status > Basics** and take a screenshot of the page.
2. Add the saved screenshot and the all of the SSL certificates files for your certificate chain to a .zip archive. Do NOT include any private key files (e.g., **.p12**, **.pfx**, or **.key** files).
3. Compose an email to Bomgar Technical Supportrequesting a software update. Attach the .zip archive containing the certificate files and screenshot. If you have an open incident with Support, include your incident number in the email. Send the email.
4. Once Bomgar Technical Support has built your new software package, they will email you instructions for how to install it. Update your software following the emailed instructions.

After these steps are complete, it is advisable to wait 24-48 hours before proceeding further. This allows time for your Bomgar client software (especially Jump Clients) to update themselves with the new certificate which Bomgar Technical Support included in your recent software update.

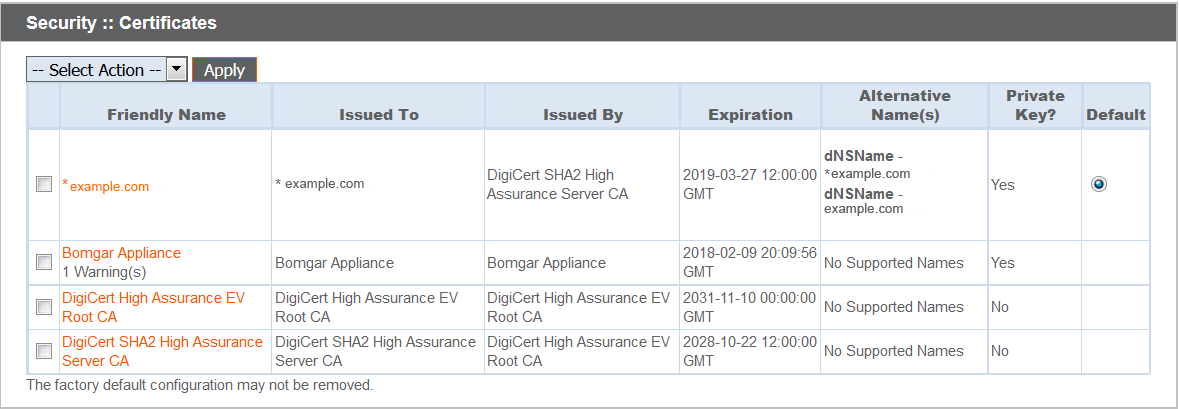
SSL Certificate Auto-Selection

Through the utilization of Server Name Indication (SNI), an extension to the TLS networking protocol, any SSL certificate stored on the appliance is a candidate to be served to any client. Because most TLS clients send Server Name Indication (SNI) information at the start of the handshaking process, this enables the appliance to determine which SSL certificate to send back to a client that requests a connection.

You may choose a default certificate to serve to clients who do not send SNI information with their request, or to clients who do send SNI information, but which does not match anything in the appliance database.

[Security > Certificates](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-header_4-0.png)

1. Go to **/appliance > Security > Certificates**.

[](https://www.bomgar.com/docs/remote-support/resources/images/appliance/security-certificates-17-1.png)

1. In the Default column, select the radio button for the certificate you wish to make default.

# Generating a Certificate Authority (CA) Certificate for Self-Signing

This section explains how to establish yourself as a root certificate authority for self-signing your certificates. Self-signed certificates are useful during development when you do not need to purchase a commercial certificate. For instructions on generating a commercially-signed certificate, see [Generating an SSL Certificate with Verisign](https://www.simba.com/products/SEN/doc/Client-Server_user_guide/content/clientserver/configuringssl/generatingcommercialcertificate.htm).

**To install OpenSSL:**

1. Install OpenSSL from [http://openssl.org](http://openssl.org/).
2. Add the path to the openssl.exe executable to your PATH variable. Refer to [http://openssl.org](http://openssl.org/) for other configuration properties.

**To create the root CA certificate:**

1. Create the C:\newcerts directory:

> md C:\newcerts

1. Change to the newcerts directory:

> cd C:\newcerts

1. Generate a CA private key:

> openssl genrsa -des3 -out CA-key.pem 2048

1. Generate the root CA certificate.

> openssl req -new -key CA-key.pem -x509 -days 1000 -out CA-cert.pem

You will be prompted for information which will be incorporated into the certificate, such as Country, City, Company Name, etc. Remember what information you entered as you may get prompted for this information again at a later stage. When asked for an email address, provide the email address of the CA contact.

The root CA certificate is created.

You will need CA-key.pem and CA-cert.pem in the following steps.

**To create a Signing a Server Certificate:**

You will need CA-key.pem and CA-cert.pem from the previous step.

1. Generate a new key:

openssl genrsa -des3 -out server-key.pem 2048

1. Generate a certificate signing request:
2. Locate the openssl.cnf file is in your OpenSSL installation directory.
3. Copy the openssl.cnf file to the newcerts directory. You may need to modify some of the configuration settings in this file.
4. Enter the following command (all in one line):

openssl req –new –config openssl.cnf –key server-key.pem –out signingReq.csr

1. Self-sign the certificate using your CA-cert.pem certificate. Enter the following command (all in one line):

openssl x509 -req -days 365 -in signingReq.csr -CA CA-cert.pem -CAkey CA-key.pem -CAcreateserial -out server-cert.pem

A server certificate is created and signed.

I'm unable to determine, how many users my apache instance can handle.

I installed Apache 2.4.18 from Ubuntu repository, as far as I remember, all settings are default ones.

I did a server stress test and discovered, that when exceeding **300** ongoing requests, server start to drop new ones, serving response code 502 or 504.

I tried to find an explanation for that in server configuration file and documentation, but I was unable to discover where that "300" came from.

I'm using mpm\_prefork, with config file like this:

<IfModule mpm\_prefork\_module>

StartServers 5

MinSpareServers 5

MaxSpareServers 10

MaxRequestWorkers 150

MaxConnectionsPerChild 0

</IfModule>

Some of the properites from apache2.conf:

Timeout 300

KeepAlive On

MaxKeepAliveRequests 500

KeepAliveTimeout 5

So the question is **what is Apache DEFAULT max concurrent clients limit** in my (default) configuration? I'm not interested in tuning Apache to handle as many connections as possible - only to understand, how Apache calculates max concurrent clients limit.

For non-threaded servers (i.e., prefork), MaxRequestWorkers translates into the maximum number of child processes that will be launched to serve requests. The default value is 256.

Sample configuration:

# minimum request rate (bytes/sec at request reading):

QS\_SrvRequestRate 120

# limits the connections for this virtual host:

QS\_SrvMaxConn 800

# allows keep-alive support till the server reaches 600 connections:

QS\_SrvMaxConnClose 600

# allows max 50 connections from a single ip address:

QS\_SrvMaxConnPerIP 50

# disables connection restrictions for certain clients:

QS\_SrvMaxConnExcludeIP 172.18.3.32

QS\_SrvMaxConnExcludeIP 192.168.10.

For example, if you are permitting file upload to a particular location, say **/var/www/example.com/wp-uploads** and wish to restrict the size of the uploaded file to 5M = 5242880Bytes, add the following directive into your .htaccess or httpd.conf file.

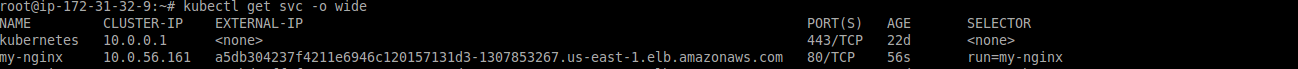
<Directory "/var/www/example.com/wp-uploads">

LimitRequestBody 5242880

</Directory>

Get the status of the service with the below command:

|  |  |
| --- | --- |
| 1 | kubectl get service -o wide |



Curl the ELB endpoint and you should see the **Welcome Nginx Page** response back.

Deployments, pods, and services can also be created using yaml/json file. Please find below a sample yaml file nginx.yaml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | apiVersion: extensions/v1beta1  kind: Deployment  metadata:  name: my-nginx  spec:  replicas: 1  template:  metadata:  labels:  run: my-nginx  spec:  containers:  - name: my-nginx  image: nginx  ports:  - containerPort: 80  ---  apiVersion: v1  kind: Service  metadata:  name: my-nginx  spec:  ports:  - name: "www"  port: 80  targetPort: 80  selector:  run: my-nginx  type: LoadBalancer |

Run the below command which uses the above yaml and creates deployment controller, a pod and service.

|  |  |
| --- | --- |
| 1 | kubectl apply -f nginx.yaml |

**Assign pod to a specific node:** Suppose there has been an application which requires to run on the same machine i.e on the same IP. Steps to achieve the same:

* Get the nodes

|  |  |
| --- | --- |
| 1 | kubectl get nodes |

* Assign label to the node. Here I am assigning node=app label to the node:

|  |  |
| --- | --- |
| 1 | kubectl label nodes <node name> node=app |

* Update the yaml to add nodeSelector in pod spec:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | ---------   spec:        containers:        - name: my-nginx          image: nginx          ports:          - containerPort: 80          nodeSelector:           node: app  ---- |

* Run the kubectl apply command and get the pods status to verify it is running on the node which has the lable node=app:

|  |  |
| --- | --- |
| 1  2 | kuebctl apply -f nginx.yaml  kubectl get pods -o wide |

Hope, now we know how to create deployment pods, services and how to assign pod to a specific node. In the next blog we will covering deployment strategy, multi-container pods, communication between pods.

To scale an application and provide a reliable service, you need to understand how the application behaves when it is deployed. You can examine application performance in a Kubernetes cluster by examining the containers, [pods](https://kubernetes.io/docs/user-guide/pods), [services](https://kubernetes.io/docs/user-guide/services), and the characteristics of the overall cluster. Kubernetes provides detailed information about an application’s resource usage at each of these levels. This information allows you to evaluate your application’s performance and where bottlenecks can be removed to improve overall performance.

* [**Resource metrics pipeline**](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/#resource-metrics-pipeline)
* [**Full metrics pipelines**](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/#full-metrics-pipelines)
* [**CronJob monitoring**](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/#cronjob-monitoring)

In Kubernetes, application monitoring does not depend on a single monitoring solution. On new clusters, you can use two separate pipelines to collect monitoring statistics by default:

* The [**resource metrics pipeline**](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/#resource-metrics-pipeline) provides a limited set of metrics related to cluster components such as the HorizontalPodAutoscaler controller, as well as the **kubectl top** utility. These metrics are collected by [metrics-server](https://github.com/kubernetes-incubator/metrics-server) and are exposed via the **metrics.k8s.io** API. **metrics-server**discovers all nodes on the cluster and queries each node’s [Kubelet](https://kubernetes.io/docs/admin/kubelet) for CPU and memory usage. The Kubelet fetches the data from [cAdvisor](https://github.com/google/cadvisor). **metrics-server** is a lightweight short-term in-memory store.
* A [**full metrics pipeline**](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/#full-metrics-pipelines), such as Prometheus, gives you access to richer metrics. In addition, Kubernetes can respond to these metrics by automatically scaling or adapting the cluster based on its current state, using mechanisms such as the Horizontal Pod Autoscaler. The monitoring pipeline fetches metrics from the Kubelet, and then exposes them to Kubernetes via an adapter by implementing either the **custom.metrics.k8s.io** or **external.metrics.k8s.io** API.

## Resource metrics pipeline

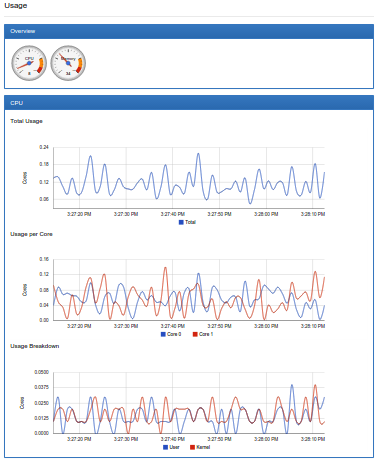
### Kubelet

The Kubelet acts as a bridge between the Kubernetes master and the nodes. It manages the pods and containers running on a machine. Kubelet translates each pod into its constituent containers and fetches individual container usage statistics from cAdvisor. It then exposes the aggregated pod resource usage statistics via a REST API.

### cAdvisor

cAdvisor is an open source container resource usage and performance analysis agent. It is purpose-built for containers and supports Docker containers natively. In Kubernetes, cAdvisor is integrated into the Kubelet binary. cAdvisor auto-discovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics. cAdvisor also provides the overall machine usage by analyzing the ‘root’ container on the machine.

On most Kubernetes clusters, cAdvisor exposes a simple UI for on-machine containers on port 4194. Here is a snapshot of part of cAdvisor’s UI that shows the overall machine usage:



## Full metrics pipelines

Many full metrics solutions exist for Kubernetes.

### Prometheus

[Prometheus](https://prometheus.io/) can natively monitor kubernetes, nodes, and prometheus itself. The [Prometheus Operator](https://coreos.com/operators/prometheus/docs/latest/)simplifies Prometheus setup on Kubernetes, and allows you to serve the custom metrics API using the[Prometheus adapter](https://github.com/directxman12/k8s-prometheus-adapter). Prometheus provides a robust query language and a built-in dashboard for querying and visualizing your data. Prometheus is also a supported data source for [Grafana](https://prometheus.io/docs/visualization/grafana/).

### Google Cloud Monitoring

Google Cloud Monitoring is a hosted monitoring service you can use to visualize and alert on important metrics in your application. can collect metrics from Kubernetes, and you can access them using the [Cloud Monitoring Console](https://app.google.stackdriver.com/). You can create and customize dashboards to visualize the data gathered from your Kubernetes cluster.

This video shows how to configure and run a Google Cloud Monitoring backed Heapster:

[](https://www.youtube.com/watch?v=xSMNR2fcoLs)



#### Google Cloud Monitoring dashboard example

This dashboard shows cluster-wide resource usage.

## CronJob monitoring

### Kubernetes Job Monitor

With the [Kubernetes Job Monitor](https://github.com/pietervogelaar/kubernetes-job-monitor) dashboard a Cluster Administrator can see which jobs are running and view the status of completed jobs.

### New Relic Kubernetes monitoring integration

[New Relic Kubernetes](https://docs.newrelic.com/docs/integrations/host-integrations/host-integrations-list/kubernetes-monitoring-integration) integration provides increased visibility into the performance of your Kubernetes environment. New Relic’s Kubernetes integration instruments the container orchestration layer by reporting metrics from Kubernetes objects. The integration gives you insight into your Kubernetes nodes, namespaces, deployments, replica sets, pods, and containers.

Marquee capabilities: View your data in pre-built dashboards for immediate insight into your Kubernetes environment. Create your own custom queries and charts in Insights from automatically reported data. Create alert conditions on Kubernetes data. Learn more on this [page](https://docs.newrelic.com/docs/integrations/host-integrations/host-integrations-list/kubernetes-monitoring-integration).

I've been writing a lot about building continuous delivery pipelines with various tools and the whole pattern in the recent past. It's becoming more and more of a mainstream pattern that many development teams now follow. And the benefits are really clear, teams can deploy software faster in smaller change batches and thus more safely. Now when you deploy your services many times a day, one thing becomes a necessity in most of the cases: zero downtime deployments.

## Zero Downtime?

Zero downtime really means that even during a deployment process, your service is responsive the whole time. Typical services in today's world are some HTTP services, so in practice this means that no requests are dropped at any point during the deployment process. During a typical deployment process we usually change our software to a newer version on the servers.

## Containers to the Rescue?

This is another good example of where using containers can really make a difference. In my opinion zero downtime deployments are much more easily achieved by using containers and especially a container management platform. By all means, this is 100% fully doable without containers too. And there are lots of organisations who have pulled it off in the pre-container era.

One of the reasons why containers make this a lot easier is the fact that we can change a container running our software to a new container with a newer version of our software so easily. In many cases it's really two commands if done manually on the command line:

docker rm -f my-service docker run -d --name my-service my\_service:v2

Of course things are much more complex than that in real life. :)

## Requirements for the Deployment Process

Let's look at the deployment process from the requirements point of view, and what it needs to do in order for us to see our software up-and-running 100% during the deployment.

### Multiple Instances of the Software Running

It's pretty obvious that in 2018 there needs to be many instances of the software running, not only for zero downtime deployment purposes but for general availability and scalability. Instance in many cases nowadays does mean a container, or a set of containers, but it could mean something else too. It could mean a virtual machine if you'd want to use those as a deployment vessel of sorts.

### Rolling Deployment

Running the software in many instances means that we need to do the update so that we will not bring everything down and then up with the newer version. Instead we need to run a gradual upgrade, updating the software one instance at a time. This ensures there's some instances serving requests while others are being updated.

Running the software in many instances also means there needs to be some sort of load balancer in front of the service. The load balancer will distribute the incoming requests among online instances of the software we're running. During the deployment process, something or someone needs to make sure load balancing is kept in sync with the deployment process. In practice it means that when shutting down an instance, it's taken out of load balancing rotation and only put back when it's running again with the newer version of the software we're deploying. One thing is of the uttermost importance: Load balancing updates must be done in a way that the load balancer itself will be online 100% during all changes.

If you are using [HAProxy](https://www.haproxy.org/), for example, as the load balancer, it would mean that we need to leave the "old" processes running with the old config until there's no connections on those anymore.

### Graceful Shutdown of Service Instances

Perhaps the most important requirement is that your software needs to work in sync with the deployment process. In practice we need to be able to tell our software it's time to go down gracefully, not to accept any more requests. This needs to also be in sync with the load balancer, when our software is going down, load balancers should not give more traffic to it. Also we need to be able to tell our automation (more on that next) how long to wait for ongoing requests to finish up, before terminating the old instance and spinning up the new version of the software.

## 100% Automation

There are many things that have to work as a well conducted-orchestra during the deployment, so it's pretty obvious that this whole process must be 100% automated. When we're doing multiple deployments per day, we have tens of servers and tens of instances of our software components running and there's no way a human can manage all of this.

On a high level the automated process looks something like this in pseudo-code:

trigger deployment

for\_each instance

send shutdown signal

remove from loadbalancer

wait for shutdown

remove old version

spin up new version

wait for startup

add to loadbalancer

done

One big reason to use containers is the fact that pretty much all container orchestrators have a built-in process of rolling deployment.

## Example Application

Let's look how all this ties in together with an example application. As usual, the whole app is available in a GitHub repo [here](https://github.com/jnummelin/graceful-stop-test)

### The App

The application is really the Simple Go app that just serves two static HTTP routes, /hello and /ping. The most interesting bit is how we handle the shutdown process in order to be able to go down gracefully and in a way that load balancers also can understand.

When the app is going down in a container, container runtimes send a SIGTERM signal to the processes. So in the app, we must catch this signal and somehow make the application go down gracefully.

// Put in a signal handler for SIGTERM

log.Println("Setting signal handler")

signals := make(chan os.Signal, 1)

signal.Notify(signals, os.Interrupt, syscall.SIGTERM)

<-signals

log.Println("Got shutdown signal, waiting 10s to finish ongoing reqs")

// This will make the healthcheck fail --> LBs will not give any more traffic

healthy = false

// Wait for the ongoing requests to finish

// Note: s.Shutdown actually does gracefull shutdown, here we just make things bit more explicit for

// demonstration purposes

time.Sleep(10 \* time.Second)

// Stop the server

s.Shutdown(nil)

On the other hand, when we're going down, load balancers should not give us any more traffic and thus allow us to drain the request queue. For this reason I made the /ping endpoint which will be used by load balancers to see if the app is healthy or not. So upon receiving SIGTERM the app will start to send an un-healthy status on the /ping endpoint and thus load balancers will not give any more traffic to it.

func ping(w http.ResponseWriter, r \*http.Request) {

if !healthy {

w.WriteHeader(http.StatusServiceUnavailable)

}

fmt.Fprintf(w, "pong")

}

The rolling deployment part is handled by the [Kontena](https://kontena.io/) container platform with a configuration that ties in the application to both the deployment process and to load balancers automatically:

services:

stop:

image: jnummelin/graceful-stop:latest

instances: 3

links:

- ingress-lb/lb

deploy:

wait\_for\_port: 8080

min\_health: 0.8

health\_check:

protocol: http

port: 8080

uri: /ping

initial\_delay: 10

timeout: 2

interval: 10

stop\_grace\_period: 15s

environment:

KONTENA\_LB\_INTERNAL\_PORT: 8080

KONTENA\_LB\_VIRTUAL\_HOSTS: stop.kontena.works

I'll walk you through the most interesting bits.

**links: ingress-lb/lb**

This tells Kontena to always configure load balancing for the application. And not only to configure it, it also keeps it in sync during the rolling deployment process.

**deploy.min\_health**

How big a portion of the application instances, in practice containers, to keep up-and-running during the deployment. In this case it means that 80% of my instances shall be kept running during the deployment. As I've specified I want to have 3 instances, it means that the rolling deployment swaps one ( round(3 \* (1 - 0.8)) = 1) container at a time for a new version.

**deploy.wait\_for\_port**

During deployment, Kontena will wait for the application instance to start listening on the given port before continuing the process to the next instance. Allows sufficient time for the app to actually boot up and to be able to serve traffic.

**health\_check**

The defined health check also makes the load balancers to use this endpoint for checking the application status. So now when the app instance is going down, the /ping endpoint will respond with 503 - Service Unavailable status to the load balancers and thus the stopping instance will be able to drain its requests.

**stop\_grace\_period**

How long to wait between sending the app SIGTERM and SIGKILL, i.e. how much time to allow for draining all the requests.

## Testing

So now with our test app handling signals properly and with proper deployment configurations for Kontena we should see 0 failed request during a deployment. Let's test this:

#### Forcing a Deployment

We must be able to force a deployment and change of containers during the deployment so we can observe the correct behavior. Luckily we have the needed commands available out of the box:

$ kontena service deploy --force graceful/stop

[done] Deploying service graceful/stop

⊛ Deployed instance demo-grid/graceful/stop-1 to node little-frost-55

⊛ Deployed instance demo-grid/graceful/stop-2 to node late-sound-19

⊛ Deployed instance demo-grid/graceful/stop-3 to node twilight-lake-92

The --force flag instructs Kontena to go and change all the containers to new ones, regardless if the image or service configuration is changed or not.

To see what happened during the deployment, we can see the service related events log:

2018-03-20T08:38:58.697Z deploy service demo-grid/graceful/stop deploy started

2018-03-20T08:38:59.107Z create\_instance pulling image jnummelin/graceful-stop:latest for graceful/stop-1 (little-frost-55)

2018-03-20T08:39:00.243Z create\_instance pulled image jnummelin/graceful-stop:latest for graceful/stop-1 (little-frost-55)

2018-03-20T08:39:10.734Z create\_instance removed previous version of service graceful/stop-1 instance (little-frost-55)

2018-03-20T08:39:10.894Z create\_instance service graceful/stop-1 instance created (little-frost-55)

2018-03-20T08:39:11.361Z create\_instance service graceful/stop-1 instance started (little-frost-55)

2018-03-20T08:39:14.421Z create\_instance pulling image jnummelin/graceful-stop:latest for graceful/stop-2 (late-sound-19)

2018-03-20T08:39:15.651Z create\_instance pulled image jnummelin/graceful-stop:latest for graceful/stop-2 (late-sound-19)

2018-03-20T08:39:26.035Z create\_instance removed previous version of service graceful/stop-2 instance (late-sound-19)

2018-03-20T08:39:26.182Z create\_instance service graceful/stop-2 instance created (late-sound-19)

2018-03-20T08:39:26.642Z create\_instance service graceful/stop-2 instance started (late-sound-19)

2018-03-20T08:39:29.784Z create\_instance pulling image jnummelin/graceful-stop:latest for graceful/stop-3 (twilight-lake-92)

2018-03-20T08:39:31.029Z create\_instance pulled image jnummelin/graceful-stop:latest for graceful/stop-3 (twilight-lake-92)

2018-03-20T08:39:41.531Z create\_instance removed previous version of service graceful/stop-3 instance (twilight-lake-92)

2018-03-20T08:39:41.705Z create\_instance service graceful/stop-3 instance created (twilight-lake-92)

2018-03-20T08:39:42.052Z create\_instance service graceful/stop-3 instance started (twilight-lake-92)

2018-03-20T08:39:44.578Z deploy service demo-grid/graceful/stop deployed

What we see is the changing of the containers as expected.

At the same time, while running the deployment, we want to make sure we do not drop any requests. For that, I use a tool called [ApacheBench](https://httpd.apache.org/docs/2.4/programs/ab.html).

$ docker run --rm -ti --net host jordi/ab ab -n 10000 -c 100 http://stop.kontena.works/hello

This is ApacheBench, Version 2.3 <$Revision: 1807734 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking stop.kontena.works (be patient)

Completed 1000 requests

Completed 2000 requests

Completed 3000 requests

Completed 4000 requests

Completed 5000 requests

Completed 6000 requests

Completed 7000 requests

Completed 8000 requests

Completed 9000 requests

Completed 10000 requests

Finished 10000 requests

Server Software:

Server Hostname: stop.kontena.works

Server Port: 80

Document Path: /hello

Document Length: 12 bytes

Concurrency Level: 100

Time taken for tests: 60.757 seconds

Complete requests: 10000

Failed requests: 0

Total transferred: 1290000 bytes

HTML transferred: 120000 bytes

Requests per second: 164.59 [#/sec] (mean)

Time per request: 607.570 [ms] (mean)

Time per request: 6.076 [ms] (mean, across all concurrent requests)

Transfer rate: 20.73 [Kbytes/sec] received

The tool is running 100 requests at a time, 10,000 requests in total.

As seen in the results, 0 requests have failed.

**Mission accomplished.**

## Summary

Getting your application to zero downtime deployments is a goal worth pursuing. It really enables you to do deployments multiple times a day without sacrificing your end-users' experience. To get you there, containers and container management platforms can make things a lot easier for you. You still need to make the app work in sync with the deployment process, but it's actually easier than many think.