Cloud-Init

Release 0.7.7

Oct 05, 2017

Contents

Everything about cloud-init, a set of python scripts and utilities to make your cloud images be all they can be!

CHAPTER 1

Summary

Cloud-init is the *defacto* multi-distribution package that handles early initialization of a cloud instance.

Capabilities

- Setting a default locale
- Setting a instance hostname
- Generating instance ssh private keys
- Adding ssh keys to a users `.ssh/authorized_keys` so they can log in
- Setting up ephemeral mount points

User configurability

Cloud-init 's behavior can be configured via user-data.

User-data can be given by the user at instance launch time.

This is done via the `--user-data` or `--user-data-file` argument to ec2-run-instances for example.

- Check your local clients documentation for how to provide a *user-data* string or *user-data* file for usage by cloud-init on instance creation.

Availability

It is currently installed in the Ubuntu Cloud Images and also in the official Ubuntu images available on EC2.

Versions for other systems can be (or have been) created for the following distributions:

- Ubuntu
- Fedora
- Debian
- RHEL
- CentOS
- *and more...*

So ask your distribution provider where you can obtain an image with it built-in if one is not already available

Formats

User data that will be acted upon by cloud-init must be in one of the following types.

Gzip Compressed Content

Content found to be gzip compressed will be uncompressed. The uncompressed data will then be used as if it were not compressed. This is typically is useful because user-data is limited to ~16384[1] bytes.

Mime Multi Part Archive

This list of rules is applied to each part of this multi-part file. Using a mime-multi part file, the user can specify more than one type of data.

For example, both a user data script and a cloud-config type could be specified.

Supported content-types:

- text/x-include-once-url
- text/x-include-url
- text/cloud-config-archive
- text/upstart-job
- text/cloud-config
- text/part-handler
- text/x-shellscript
- text/cloud-boothook

Helper script to generate mime messages

```
#!/usr/bin/python

import sys

from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
```

[1] See your cloud provider for applicable user-data size limitations...

```
if len(sys.argv) == 1:
    print("%s input-file:type ..." % (sys.argv[0]))
    sys.exit(1)

combined_message = MIMEMultipart()
for i in sys.argv[1:]:
    (filename, format_type) = i.split(":", 1)
    with open(filename) as fh:
        contents = fh.read()
    sub_message = MIMEText(contents, format_type, sys.getdefaultencoding())
    sub_message.add_header('Content-Disposition', 'attachment; filename="%s"' %␣
↪(filename))
    combined_message.attach(sub_message)

print(combined_message)
```

User-Data Script

Typically used by those who just want to execute a shell script.

Begins with: `#!` or `Content-Type: text/x-shellscript` when using a MIME archive.

Example

```
$ cat myscript.sh

#!/bin/sh
echo "Hello World.  The time is now $(date -R)!" | tee /root/output.txt

$ euca-run-instances --key mykey --user-data-file myscript.sh ami-a07d95c9
```

Include File

This content is a `include` file.

The file contains a list of urls, one per line. Each of the URLs will be read, and their content will be passed through this same set of rules. Ie, the content read from the URL can be gzipped, mime-multi-part, or plain text.

Begins with: `#include` or `Content-Type: text/x-include-url` when using a MIME archive.

Cloud Config Data

Cloud-config is the simplest way to accomplish some things via user-data. Using cloud-config syntax, the user can specify certain things in a human friendly format.

These things include:

- apt upgrade should be run on first boot
- a different apt mirror should be used
- additional apt sources should be added
- certain ssh keys should be imported

- *and many more...*

Note: The file must be valid yaml syntax.

See the *Cloud config examples* section for a commented set of examples of supported cloud config formats.

Begins with: `#cloud-config` or `Content-Type: text/cloud-config` when using a MIME archive.

Upstart Job

Content is placed into a file in `/etc/init`, and will be consumed by upstart as any other upstart job.

Begins with: `#upstart-job` or `Content-Type: text/upstart-job` when using a MIME archive.

Cloud Boothook

This content is `boothook` data. It is stored in a file under `/var/lib/cloud` and then executed immediately. This is the earliest `hook` available. Note, that there is no mechanism provided for running only once. The boothook must take care of this itself. It is provided with the instance id in the environment variable `INSTANCE_I`. This could be made use of to provide a 'once-per-instance' type of functionality.

Begins with: `#cloud-boothook` or `Content-Type: text/cloud-boothook` when using a MIME archive.

Part Handler

This is a `part-handler`. It will be written to a file in `/var/lib/cloud/data` based on its filename (which is generated). This must be python code that contains a `list_types` method and a `handle_type` method. Once the section is read the `list_types` method will be called. It must return a list of mime-types that this part-handler handles.

The `handle_type` method must be like:

```
def handle_part(data, ctype, filename, payload):
  # data = the cloudinit object
  # ctype = "__begin__", "__end__", or the mime-type of the part that is being␣
↪handled.
  # filename = the filename of the part (or a generated filename if none is present␣
↪in mime data)
  # payload = the parts' content
```

Cloud-init will then call the `handle_type` method once at begin, once per part received, and once at end. The `begin` and `end` calls are to allow the part handler to do initialization or teardown.

Begins with: `#part-handler` or `Content-Type: text/part-handler` when using a MIME archive.

Example

```
#part-handler
# vi: syntax=python ts=4

def list_types():
    # return a list of mime-types that are handled by this module
    return(["text/plain", "text/go-cubs-go"])

```

```
def handle_part(data,ctype,filename,payload):
    # data: the cloudinit object
    # ctype: '__begin__', '__end__', or the specific mime-type of the part
    # filename: the filename for the part, or dynamically generated part if
    #           no filename is given attribute is present
    # payload: the content of the part (empty for begin or end)
    if ctype == "__begin__":
       print "my handler is beginning"
       return
    if ctype == "__end__":
       print "my handler is ending"
       return

    print "==== received ctype=%s filename=%s ====" % (ctype,filename)
    print payload
    print "==== end ctype=%s filename=%s" % (ctype, filename)
```

Also this blog post offers another example for more advanced usage.

Directory layout

Cloudinits's directory structure is somewhat different from a regular application:

```
/var/lib/cloud/
    - data/
       - instance-id
       - previous-instance-id
       - datasource
       - previous-datasource
       - previous-hostname
    - handlers/
    - instance
    - instances/
        i-00000XYZ/
          - boot-finished
          - cloud-config.txt
          - datasource
          - handlers/
          - obj.pkl
          - scripts/
          - sem/
          - user-data.txt
          - user-data.txt.i
    - scripts/
       - per-boot/
       - per-instance/
       - per-once/
    - seed/
    - sem/
```

`/var/lib/cloud`

The main directory containing the cloud-init specific subdirectories. It is typically located at `/var/lib` but there are certain configuration scenarios where this can be altered.

TBD, describe this overriding more.

`data/`

Contains information releated to instance ids, datasources and hostnames of the previous and current instance if they are different. These can be examined as needed to determine any information releated to a previous boot (if applicable).

`handlers/`

Custom `part-handlers` code is written out here. Files that end up here are written out with in the scheme of `part-handler-XYZ` where `XYZ` is the handler number (the first handler found starts at 0).

`instance`

A symlink to the current `instances/` subdirectory that points to the currently active instance (which is active is dependent on the datasource loaded).

`instances/`

All instances that were created using this image end up with instance identifer subdirectories (and corresponding data for each instance). The currently active instance will be symlinked the the `instance` symlink file defined previously.

`scripts/`

Scripts that are downloaded/created by the corresponding `part-handler` will end up in one of these subdirectories.

`seed/`

TBD

`sem/`

Cloud-init has a concept of a module sempahore, which basically consists of the module name and its frequency. These files are used to ensure a module is only ran *per-once*, *per-instance*, *per-always*. This folder contains sempaphore *files* which are only supposed to run *per-once* (not tied to the instance id).

Cloud config examples

Including users and groups

```
# Add groups to the system
# The following example adds the ubuntu group with members foo and bar and
# the group cloud-users.
groups:
  - ubuntu: [foo,bar]
  - cloud-users

# Add users to the system. Users are added after groups are added.
users:
  - default
  - name: foobar
    gecos: Foo B. Bar
    primary-group: foobar
    groups: users
    selinux-user: staff_u
    expiredate: 2012-09-01
    ssh-import-id: foobar
    lock_passwd: false
    passwd: $6$j212wezy$7H/1LT4f9/
↪N3wpgNunhsIqtMj62OKiS3nyNwuizouQc3u7MbYCarYeAHWYPYb2FT.lbioDm2RrkJPb9BZMN1O/
  - name: barfoo
    gecos: Bar B. Foo
    sudo: ALL=(ALL) NOPASSWD:ALL
    groups: users, admin
    ssh-import-id: None
    lock_passwd: true
    ssh-authorized-keys:
      - <ssh pub key 1>
      - <ssh pub key 2>
  - name: cloudy
    gecos: Magic Cloud App Daemon User
    inactive: true
    system: true

# Valid Values:
#   name: The user's login name
#   gecos: The user name's real name, i.e. "Bob B. Smith"
#   homedir: Optional. Set to the local path you want to use. Defaults to
#           /home/<username>
#   primary-group: define the primary group. Defaults to a new group created
#           named after the user.
#   groups:  Optional. Additional groups to add the user to. Defaults to none
#   selinux-user:  Optional. The SELinux user for the user's login, such as
#           "staff_u". When this is omitted the system will select the default
#           SELinux user.
#   lock_passwd: Defaults to true. Lock the password to disable password login
#   inactive: Create the user as inactive
#   passwd: The hash -- not the password itself -- of the password you want
#           to use for this user. You can generate a safe hash via:
#               mkpasswd --method=SHA-512 --rounds=4096
#           (the above command would create from stdin an SHA-512 password hash
#           with 4096 salt rounds)
#
#           Please note: while the use of a hashed password is better than
#               plain text, the use of this feature is not ideal. Also,
#               using a high number of salting rounds will help, but it should
#               not be relied upon.
#
#               To highlight this risk, running John the Ripper against the
#               example hash above, with a readily available wordlist, revealed
#               the true password in 12 seconds on a i7-2620QM.
#
#               In other words, this feature is a potential security risk and is
#               provided for your convenience only. If you do not fully trust the
#               medium over which your cloud-config will be transmitted, then you
#               should use SSH authentication only.
#
#               You have thus been warned.
#   no-create-home: When set to true, do not create home directory.
#   no-user-group: When set to true, do not create a group named after the user.
#   no-log-init: When set to true, do not initialize lastlog and faillog database.
#   ssh-import-id: Optional. Import SSH ids
#   ssh-authorized-keys: Optional. [list] Add keys to user's authorized keys file
#   sudo: Defaults to none. Set to the sudo string you want to use, i.e.
#           ALL=(ALL) NOPASSWD:ALL. To add multiple rules, use the following
#           format.
#               sudo:
#                   - ALL=(ALL) NOPASSWD:/bin/mysql
#                   - ALL=(ALL) ALL
#           Note: Please double check your syntax and make sure it is valid.
#               cloud-init does not parse/check the syntax of the sudo
#               directive.
#   system: Create the user as a system user. This means no home directory.
#

# Default user creation:
#
# Unless you define users, you will get a 'ubuntu' user on ubuntu systems with the
# legacy permission (no password sudo, locked user, etc). If however, you want
# to have the 'ubuntu' user in addition to other users, you need to instruct
# cloud-init that you also want the default user. To do this use the following
# syntax:
#    users:
#      - default
#      - bob
#      - ....
#  foobar: ...
#
# users[0] (the first user in users) overrides the user directive.
#
# The 'default' user above references the distro's config:
# system_info:
#   default_user:
#    name: Ubuntu
#    plain_text_passwd: 'ubuntu'
#    home: /home/ubuntu
#    shell: /bin/bash
#    lock_passwd: True
#    gecos: Ubuntu
#    groups: [adm, audio, cdrom, dialout, floppy, video, plugdev, dip, netdev]
```

Writing out arbitrary files

```
#cloud-config
# vim: syntax=yaml
#
# This is the configuration syntax that the write_files module
# will know how to understand. encoding can be given b64 or gzip or (gz+b64).
# The content will be decoded accordingly and then written to the path that is
# provided.
#
# Note: Content strings here are truncated for example purposes.
write_files:
-   encoding: b64
    content: CiMgVGhpcyBmaWxlIGNvbnRyb2xzIHRoZSBzdGF0ZSBvZiBTRUxpbnV4...
    owner: root:root
    path: /etc/sysconfig/selinux
    permissions: '0644'
-   content: |
        # My new /etc/sysconfig/samba file

        SMBDOPTIONS="-D"
    path: /etc/sysconfig/samba
-   content: !!binary |
        f0VMRgIBAQAAAAAAAAAAAAIAPgABAAAAwARAAAAAAABAAAAAAAAAAJAVAAAAAAAAAAAAAAEAAOAAI
        AEAAHgAdAAYAAAAFAAAAQAAAAAAAAABAAEAAAAAAAEAAQAAAAAAAwAEAAAAAAADAAQAAAAAAAAgA
        AAAAAAAAAwAAAAQAAAAAAgAAAAAAAAACQAAAAAAAAJAAAAAAAAcAAAAAAAAABwAAAAAAAAAAQAA
        ....
    path: /bin/arch
    permissions: '0555'
-   encoding: gzip
    content: !!binary |
        H4sIAIDb/U8C/1NW1E/KzNMvzuBKTc7IV8hIzcnJVyjPL8pJ4QIA6N+MVxsAAAA=
    path: /usr/bin/hello
    permissions: '0755'
```

Adding a yum repository

```
#cloud-config
# vim: syntax=yaml
#
# Add yum repository configuration to the system
#
# The following example adds the file /etc/yum.repos.d/epel_testing.repo
# which can then subsequently be used by yum for later operations.
yum_repos:
    # The name of the repository
    epel-testing:
        # Any repository configuration options
        # See: man yum.conf
        #
        # This one is required!
        baseurl: http://download.fedoraproject.org/pub/epel/testing/5/$basearch
        enabled: false
        failovermethod: priority
        gpgcheck: true
        gpgkey: file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL
        name: Extra Packages for Enterprise Linux 5 - Testing
```

Configure an instances trusted CA certificates

```
#cloud-config
#
# This is an example file to configure an instance's trusted CA certificates
# system-wide for SSL/TLS trust establishment when the instance boots for the
# first time.
#
# Make sure that this file is valid yaml before starting instances.
# It should be passed as user-data when starting the instance.

ca-certs:
  # If present and set to True, the 'remove-defaults' parameter will remove
  # all the default trusted CA certificates that are normally shipped with
  # Ubuntu.
  # This is mainly for paranoid admins - most users will not need this
  # functionality.
  remove-defaults: true

  # If present, the 'trusted' parameter should contain a certificate (or list
  # of certificates) to add to the system as trusted CA certificates.
  # Pay close attention to the YAML multiline list syntax.  The example shown
  # here is for a list of multiline certificates.
  trusted:
  - |
   -----BEGIN CERTIFICATE-----
   YOUR-ORGS-TRUSTED-CA-CERT-HERE
   -----END CERTIFICATE-----
  - |
   -----BEGIN CERTIFICATE-----
   YOUR-ORGS-TRUSTED-CA-CERT-HERE
   -----END CERTIFICATE-----
```

Configure an instances resolv.conf

Note: when using a config drive and a RHEL like system resolv.conf will also be managed 'automatically' due to the available information provided for dns servers in the config drive network format. For those that wish to have different settings use this module.

```
#cloud-config
#
# This is an example file to automatically configure resolv.conf when the
# instance boots for the first time.
#
# Ensure that your yaml is valid and pass this as user-data when starting
# the instance. Also be sure that your cloud.cfg file includes this
# configuration module in the appropriate section.
#
manage-resolv-conf: true

resolv_conf:
  nameservers: ['8.8.4.4', '8.8.8.8']
  searchdomains:
    - foo.example.com
    - bar.example.com
  domain: example.com
  options:
    rotate: true
    timeout: 1
```

Install and run chef recipes

```
#cloud-config
#
# This is an example file to automatically install chef-client and run a
# list of recipes when the instance boots for the first time.
# Make sure that this file is valid yaml before starting instances.
# It should be passed as user-data when starting the instance.
#
# This example assumes the instance is 12.04 (precise)


# The default is to install from packages.

# Key from http://apt.opscode.com/packages@opscode.com.gpg.key
apt_sources:
 - source: "deb http://apt.opscode.com/ $RELEASE-0.10 main"
   key: |
     -----BEGIN PGP PUBLIC KEY BLOCK-----
     Version: GnuPG v1.4.9 (GNU/Linux)

     mQGiBEppC7QRBADfsOkZU6KZK+YmKw4wev5mjKJEkVGlus+NxW8wItX5sGa6kdUu
     twAyj7Yr92rF+ICFEP3gGU6+lGo0Nve7KxkN/1W7/m3G4zuk+ccIKmjp8KS3qn99
     dxy64vcji9jIllVa+XXOGIp0G8GEaj7mbkixL/bMeGfdMlv8Gf2XPpp9vwCgn/GC
     JKacfnw7MpLKUHOYSlb//JsEAJqao3ViNfav83jJKEkD8cf59Y8xKia5OpZqTK5W
     ShVnNWS3U5IVQk10ZDH97Qn/YrK387H4CyhLE9mxPXs/ul18ioiaars/q2MEKU2I
     XKfV21eMLO9LYd6Ny/Kqj8o5WQK2J6+NAhSwvthZcIEphcFignIuobP+B5wNFQpe
     DbKfA/0WvN2OwFeWRcmmd3Hz7nHTpcnSF+4QX6yHRF/5BgxkG6IqBIACQbzPn6Hm
     sMtm/SVf11izmDqSsQptCrOZILfLX/mE+YOl+CwWSHhl+YsFts1WOuh1EhQD26aO
     Z84HuHV5HFRWjDLw9LriltBVQcXbpfSrRP5bdr7Wh8vhqJTPjrQnT3BzY29kZSBQ
     YWNrYWdlcyA8cGFja2FnZXNAb3BzY29kZS5jb20+iGAEExECACAFAkppC7QCGwMG
     CwkIBwMCBBUCCAMEFgIDAQIeAQIXgAAKCRApQKupg++Caj8sAKCOXmdG36gWji/K
     +o+XtBfvdMnFYQCfTCEWxRy2BnzLoBBFCjDSK6sJqCu5Ag0ESmkLtBAIAIO2SwlR
     lU5i6gTOp42RHWW7/pmW78CwUqJnYqnXROrt3h9F9xrsGkH0Fh1FRtsnncgzIhvh
     DLQnRHnkXm0ws0jV0PF74ttoUT6BLAUsFi2SPP1zYNJ9H9fhhK/pjijtAcQwdgxu
     wwNJ5xCEscBZCjhSRXm0d30bK1o49Cow8ZIbHtnXVP41c9QWOzX/LaGZsKQZnaMx
     EzDk8dyyctR2f03vRSVyTFGgdpUcpbr9eTFVgikCa6ODEBv+0BnCH6yGTXwBid9g
     w0o1e/2DviKUWCC+AlAUOubLmOIGFBuI4UR+rux9affbHcLIOTiKQXv79lW3P7W8
     AAfniSQKfPWXrrcAAwUH/2XBqD4Uxhbs25HDUUiM/m6Gnlj6EsStg8n0nMggLhuN
     QmPfoNByMPUqvA7sULyfr6xCYzbzRNxABHSpf85FzGQ29RF4xsA4vOOU8RDIYQ9X
     Q8NqqR6pydprRFqWe47hsAN7BoYuhWqTtOLSBmnAnzTR5pURoqcquWYiiEavZixJ
     3ZRAq/HMGioJEtMFrvsZjGXuzef7f0ytfR1zYeLVWnL9Bd32CueBlI7dhYwkFe+V
     Ep5jWOCj02C1wHcwt+uIRDJV6TdtbIiBYAdOMPk15+VBdweBXwMuYXr76+A7VeDL
     zIhi7tKFo6WiwjKZq0dzctsJJjtIfr4K4vbiD9Ojg1iISQQYEQIACQUCSmkLtAIb
     DAAKCRApQKupg++CauISAJ9CxYPOKhOxalBnVTLeNUkAHGg2gACeIsbobtaD4ZHG
     0GLl8EkfA8uhluM=
     =zKAm
     -----END PGP PUBLIC KEY BLOCK-----

chef:

 # Valid values are 'gems' and 'packages' and 'omnibus'
 install_type: "packages"

 # Boolean: run 'install_type' code even if chef-client
 #          appears already installed.
 force_install: false

 # Chef settings
 server_url: "https://chef.yourorg.com:4000"

 # Node Name
 # Defaults to the instance-id if not present
 node_name: "your-node-name"

 # Environment
 # Defaults to '_default' if not present
 environment: "production"

 # Default validation name is chef-validator
 validation_name: "yourorg-validator"
 # if validation_cert's value is "system" then it is expected
 # that the file already exists on the system.
 validation_cert: |
     -----BEGIN RSA PRIVATE KEY-----
     YOUR-ORGS-VALIDATION-KEY-HERE
     -----END RSA PRIVATE KEY-----

 # A run list for a first boot json
 run_list:
  - "recipe[apache2]"
  - "role[db]"

 # Specify a list of initial attributes used by the cookbooks
 initial_attributes:
    apache:
      prefork:
        maxclients: 100
      keepalive: "off"

 # if install_type is 'omnibus', change the url to download
 omnibus_url: "https://www.opscode.com/chef/install.sh"


# Capture all subprocess output into a logfile
# Useful for troubleshooting cloud-init issues
output: {all: '| tee -a /var/log/cloud-init-output.log'}
```

Setup and run puppet

```
#cloud-config
#
# This is an example file to automatically setup and run puppetd
# when the instance boots for the first time.
# Make sure that this file is valid yaml before starting instances.
# It should be passed as user-data when starting the instance.
puppet:
 # Every key present in the conf object will be added to puppet.conf:
 # [name]
 # subkey=value
 #
 # For example the configuration below will have the following section
 # added to puppet.conf:
 # [puppetd]
 # server=puppetmaster.example.org
 # certname=i-0123456.ip-X-Y-Z.cloud.internal
 #
 # The puppmaster ca certificate will be available in
 # /var/lib/puppet/ssl/certs/ca.pem
 conf:
   agent:
     server: "puppetmaster.example.org"
     # certname supports substitutions at runtime:
     #   %i: instanceid
     #       Example: i-0123456
```

```
      #   %f: fqdn of the machine
      #       Example: ip-X-Y-Z.cloud.internal
      #
      # NB: the certname will automatically be lowercased as required by puppet
      certname: "%i.%f"
    # ca_cert is a special case. It won't be added to puppet.conf.
    # It holds the puppetmaster certificate in pem format.
    # It should be a multi-line string (using the | yaml notation for
    # multi-line strings).
    # The puppetmaster certificate is located in
    # /var/lib/puppet/ssl/ca/ca_crt.pem on the puppetmaster host.
    #
    ca_cert: |
      -----BEGIN CERTIFICATE-----
      MIICCTCCAXKgAwIBAgIBATANBgkqhkiG9w0BAQUFADANMQswCQYDVQQDDAJjYTAe
      Fw0xMDAyMTUxNzI5MjFaFw0xNTAyMTQxNzI5MjFaMA0xCzAJBgNVBAMMAmNhMIGf
      MA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCu7Q40sm47/E1Pf+r8AYb/V/FWGPgc
      b014OmNoX7dgCxTDvps/h8Vw555PdAFsW5+QhsGr31IJNI3kSYprFQcYf7A8tNWu
      1MASW2CfaEiOEi9F1R3R4Qlz4ix+iNoHiUDTjazw/tZwEdxaQXQVLwgTGRwVa+aA
      qbutJKi93MILLwIDAQABo3kwdzA4BglghkgBhvhCAQ0EKxYpUHVwcGV0IFJ1Ynkv
      T3BlblNTTCBHZW5lcmF0ZWQgQ2VydGlmaWNhdGUwDwYDVR0TAQH/BAUwAwEB/zAd
      BgNVHQ4EFgQUu4+jHB+GYE5Vxo+ol1OAhevspjAwCwYDVR0PBAQDAgEGMA0GCSqG
      SIb3DQEBBQUAA4GBAH/rxlUIjwNb3n7TXJcDJ6MMHUlwjr03BDJXKb34Ulndkpaf
      +GAlzPXWa7bO908M9I8RnPfvtKnteLbvgTK+h+zX1XCty+S2EQWk29i2AdoqOTxb
      hppiGMp0tT5Havu4aceCXiy2crVcudj3NFciy8X66SoECemW9UYDCb9T5D0d
      -----END CERTIFICATE-----
```

Add apt repositories

```
#cloud-config

# Add apt repositories
#
# Default: auto select based on cloud metadata
#  in ec2, the default is <region>.archive.ubuntu.com
# apt_mirror:
#   use the provided mirror
# apt_mirror_search:
#   search the list for the first mirror.
#   this is currently very limited, only verifying that
#   the mirror is dns resolvable or an IP address
#
# if neither apt_mirror nor apt_mirror search is set (the default)
# then use the mirror provided by the DataSource found.
# In EC2, that means using <region>.ec2.archive.ubuntu.com
#
# if no mirror is provided by the DataSource, and 'apt_mirror_search_dns' is
# true, then search for dns names '<distro>-mirror' in each of
# - fqdn of this host per cloud metadata
# - localdomain
# - no domain (which would search domains listed in /etc/resolv.conf)
# If there is a dns entry for <distro>-mirror, then it is assumed that there
# is a distro mirror at http://<distro>-mirror.<domain>/<distro>
#
# That gives the cloud provider the opportunity to set mirrors of a distro
# up and expose them only by creating dns entries.
```

```
#
# if none of that is found, then the default distro mirror is used
apt_mirror: http://us.archive.ubuntu.com/ubuntu/
apt_mirror_search:
 - http://local-mirror.mydomain
 - http://archive.ubuntu.com
apt_mirror_search_dns: False
```

Run commands on first boot

```
#cloud-config

# boot commands
# default: none
# this is very similar to runcmd, but commands run very early
# in the boot process, only slightly after a 'boothook' would run.
# bootcmd should really only be used for things that could not be
# done later in the boot process.  bootcmd is very much like
# boothook, but possibly with more friendly.
# - bootcmd will run on every boot
# - the INSTANCE_ID variable will be set to the current instance id.
# - you can use 'cloud-init-boot-per' command to help only run once
bootcmd:
 - echo 192.168.1.130 us.archive.ubuntu.com > /etc/hosts
 - [ cloud-init-per, once, mymkfs, mkfs, /dev/vdb ]
```

```
#cloud-config

# run commands
# default: none
# runcmd contains a list of either lists or a string
# each item will be executed in order at rc.local like level with
# output to the console
# - runcmd only runs during the first boot
# - if the item is a list, the items will be properly executed as if
#   passed to execve(3) (with the first arg as the command).
# - if the item is a string, it will be simply written to the file and
#   will be interpreted by 'sh'
#
# Note, that the list has to be proper yaml, so you have to quote
# any characters yaml would eat (':' can be problematic)
runcmd:
 - [ ls, -l, / ]
 - [ sh, -xc, "echo $(date) ': hello world!'" ]
 - [ sh, -c, echo "=========hello world'=========" ]
 - ls -l /root
 - [ wget, "http://slashdot.org", -O, /tmp/index.html ]
```

Alter the completion message

```
#cloud-config

# final_message
# default: cloud-init boot finished at $TIMESTAMP. Up $UPTIME seconds
# this message is written by cloud-final when the system is finished
# its first boot
final_message: "The system is finally up, after $UPTIME seconds"
```

Install arbitrary packages

```
#cloud-config

# Install additional packages on first boot
#
# Default: none
#
# if packages are specified, this apt_update will be set to true
#
# packages may be supplied as a single package name or as a list
# with the format [<package>, <version>] wherein the specifc
# package version will be installed.
packages:
 - pwgen
 - pastebinit
 - [libpython2.7, 2.7.3-0ubuntu3.1]
```

Run apt or yum upgrade

```
#cloud-config

# Upgrade the instance on first boot
# (ie run apt-get upgrade)
#
# Default: false
# Aliases: apt_upgrade
package_upgrade: true
```

Adjust mount points mounted

```
#cloud-config

# set up mount points
# 'mounts' contains a list of lists
#  the inner list are entries for an /etc/fstab line
#  ie : [ fs_spec, fs_file, fs_vfstype, fs_mntops, fs-freq, fs_passno ]
#
# default:
# mounts:
#  - [ ephemeral0, /mnt ]
#  - [ swap, none, swap, sw, 0, 0 ]
#
# in order to remove a previously listed mount (ie, one from defaults)
# list only the fs_spec.  For example, to override the default, of
# mounting swap:
# - [ swap ]
# or
# - [ swap, null ]
#
# - if a device does not exist at the time, an entry will still be
#   written to /etc/fstab.
# - '/dev' can be ommitted for device names that begin with: xvd, sd, hd, vd
# - if an entry does not have all 6 fields, they will be filled in
#   with values from 'mount_default_fields' below.
#
# Note, that you should set 'nobootwait' (see man fstab) for volumes that may
# not be attached at instance boot (or reboot)
#
mounts:
 - [ ephemeral0, /mnt, auto, "defaults,noexec" ]
 - [ sdc, /opt/data ]
 - [ xvdh, /opt/data, "auto", "defaults,nobootwait", "0", "0" ]
 - [ dd, /dev/zero ]

# mount_default_fields
# These values are used to fill in any entries in 'mounts' that are not
# complete.  This must be an array, and must have 7 fields.
mount_default_fields: [ None, None, "auto", "defaults,nobootwait", "0", "2" ]


# swap can also be set up by the 'mounts' module
# default is to not create any swap files, because 'size' is set to 0
swap:
   filename: /swap.img
   size: "auto" # or size in bytes
   maxsize: size in bytes
```

Call a url when finished

```
#cloud-config

# phone_home: if this dictionary is present, then the phone_home
# cloud-config module will post specified data back to the given
# url
# default: none
# phone_home:
#  url: http://my.foo.bar/$INSTANCE/
#  post: all
#  tries: 10
#
phone_home:
 url: http://my.example.com/$INSTANCE_ID/
 post: [ pub_key_dsa, pub_key_rsa, pub_key_ecdsa, instance_id ]
```

Reboot/poweroff when finished

```
#cloud-config

## poweroff or reboot system after finished
# default: none
```

```
#
# power_state can be used to make the system shutdown, reboot or
# halt after boot is finished.  This same thing can be acheived by
# user-data scripts or by runcmd by simply invoking 'shutdown'.
#
# Doing it this way ensures that cloud-init is entirely finished with
# modules that would be executed, and avoids any error/log messages
# that may go to the console as a result of system services like
# syslog being taken down while cloud-init is running.
#
# If you delay '+5' (5 minutes) and have a timeout of
# 120 (2 minutes), then the max time until shutdown will be 7 minutes.
# cloud-init will invoke 'shutdown +5' after the process finishes, or
# when 'timeout' seconds have elapsed.
#
# delay: form accepted by shutdown.  default is 'now'. other format
#        accepted is +m (m in minutes)
# mode: required. must be one of 'poweroff', 'halt', 'reboot'
# message: provided as the message argument to 'shutdown'. default is none.
# timeout: the amount of time to give the cloud-init process to finish
#          before executing shutdown.
# condition: apply state change only if condition is met.
#            May be boolean True (always met), or False (never met),
#            or a command string or list to be executed.
#            command's exit code indicates:
#               0: condition met
#               1: condition not met
#            other exit codes will result in 'not met', but are reserved
#            for future use.
#
power_state:
 delay: "+30"
 mode: poweroff
 message: Bye Bye
 timeout: 30
 condition: True
```

Configure instances ssh-keys

```
#cloud-config

# add each entry to ~/.ssh/authorized_keys for the configured user or the
# first user defined in the user definition directive.
ssh_authorized_keys:
  - ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEA3FSyQwBI6Z+nCSjUUk8EEAnnkhXlukKoUPND/
↪RRClWz2s5TCzIkd3Ou5+Cyz71X0XmazM3l5WgeErvtIwQMyT1KjNoMhoJMrJnWqQPOt5Q8zWd9qG7PBl9+eiH5qV7NZ␣
↪mykey@host
  - ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEA3I7VUf2l5gSn5uavROsc5HRDpZdQueUq5ozemNSj8T7enqKHOEaFoU2VoPgGEWC9RyzSQVe
↪+i1D+ey3ONkZLN+LQ714cgj8fRS4Hj29SCmXp5Kt5/82cD/VN3NtHw== smoser@brickies

# Send pre-generated ssh private keys to the server
# If these are present, they will be written to /etc/ssh and
# new random keys will not be generated
#  in addition to 'rsa' and 'dsa' as shown below, 'ecdsa' is also supported
ssh_keys:
  rsa_private: |
    -----BEGIN RSA PRIVATE KEY-----
    MIIBxwIBAAJhAKD0YSHy73nUgysO13XsJmd4fHiFyQ+00R7VVu2iV9Qcon2LZS/x
    1cydPZ4pQpfjEha6WxZ6o8ci/Ea/w0n+0HGPwaxlEG2Z9inNtj3pgFrYcRztfECb
    1j6HCibZbAzYtwIBIwJgO8h72WjcmvcpZ8OvHSvTwAguO2TkR6mPgHsgSaKy6GJo
    PUJnaZRWuba/HX0KGyhz19nPzLpzG5f0fYahlMJAyc13FV7K6kMBPXTRR6FxgHEg
    L0MPC7cdqAwOVNcPY6A7AjEA1bNaIjOzFN2sfZX0j7OMhQuc4zP7r80zaGc5oy6W
    p58hRAncFKEvnEq2CeL3vtuZAjEAwNBHpbNsBYTRPCHM7rZuG/iBtwp8Rxhc9I5w
    ixvzMgi+HpGLWzUIBS+P/XhekIjPAjA285rVmEP+DR255Ls65QbgYhJmTzIXQ2T9
    luLvcmFBC6l35Uc4gTgg4ALsmXLn71MCMGMpSWspEvuGInayTCL+vEjmNBT+FAdO
    W7D4zCpI43jRS9U06JVOeSc9CDk2lwiA3wIwCTB/6uc8Cq85D9YqpM10FuHjKpnP
    REPPOyrAspdeOAV+6VKRavstea7+2DZmSUgE
    -----END RSA PRIVATE KEY-----

  rsa_public: ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAGEAoPRhIfLvedSDKw7XdewmZ3h8eIXJD7TRHtVW7aJX1ByifYtlL/
↪HVzJ09nilCl+MSFrpbFnqjxyL8Rr/DSf7QcY/BrGUQbZn2Kc22PemAWthxHO18QJvWPocKJtlsDNi3␣
↪smoser@localhost

  dsa_private: |
    -----BEGIN DSA PRIVATE KEY-----
    MIIBuwIBAAKBgQDP2HLu7pTExL89USyM0264RCyWX/CMLmukxX0Jdbm29ax8FBJT
    pLrO8TIXVY5rPAJm1dTHnpuyJhOvU9G7M8tPUABtzSJh4GVSHlwaCfycwcpLv9TX
    DgWIpSj+6EiHCyaRlB1/CBp9RiaB+10QcFbm+lapuET+/Au6vSDp9IRtlQIVAIMR
    8KucvUYbOEI+yv+5LW9u3z/BAoGBAI0q6JP+JvJmwZFaeCMMVxXUbqiSko/P1lsa
    LNNBHZ5/8MOUIm8rB2FC6ziidfueJpqTMqeQmSAlEBCwnwreUnGfRrKoJpyPNENY
    d15MG6N5J+z81sEcHFeprryZ+D3Ge9VjPq3Tf3NhKKwCDQ0240aPezbnjPeFm4mH
    bYxxcZ9GAoGAXmLIFSQgiAPu459rCKxT46tHJtM0QfnNiEnQLbFluefZ/yiI4DI3
    8UzTCOXLhUA7ybmZha+D/csj15Y9/BNFuO7unzVhikCQV9DTeXX46pG4s1o23JKC
    /QaYWNMZ7kTRv+wWow9MhGiVdML4ZN4XnifuO5krqAybngIy66PMEoQCFEIsKKWv
    99iziAH0KBMVbxy03Trz
    -----END DSA PRIVATE KEY-----

  dsa_public: ssh-dss AAAAB3NzaC1kc3MAAACBAM/
↪Ycu7ulMTEvz1RLIzTbrhELJZf8Iwua6TFfQl1ubb1rHwUElOkus7xMhdVjms8AmbV1Meem7ImE69T0bszy09QAG3NImHgZVIeX
↪JzByku/
↪1NcOBYilKP7oSIcLJpGUHX8IGn1GJoH7XRBwVub6Vqm4RP78C7q9IOn0hG2VAAAAFQCDEfCrnL1GGzhCPsr/
↪uS1vbt8/wQAAAIEAjSrok/4m8mbBkVp4IwxXFdRuqJKSj8/WWxos00Ednn/
↪ww5QibysHYULrOKJ1+54mmpMyp5CZICUQELCfCt5ScZ9GsqgmnI80Q1h3Xkwbo3kn7PzWwRwcV6muvJn4PcZ71WM+rdN/
↪c2EorAINDTbjRo97NueM94WbiYdtjHFxn0YAAACAXmLIFSQgiAPu459rCKxT46tHJtM0QfnNiEnQLbFluefZ/
↪yiI4DI38UzTCOXLhUA7ybmZha+D/csj15Y9/BNFuO7unzVhikCQV9DTeXX46pG4s1o23JKC/
↪QaYWNMZ7kTRv+wWow9MhGiVdML4ZN4XnifuO5krqAybngIy66PMEoQ= smoser@localhost
```

Datasources

What is a datasource?

Datasources are sources of configuration data for cloud-init that typically come from the user (aka userdata) or come from the stack that created the configuration drive (aka metadata). Typical userdata would include files, yaml, and shell scripts while typical metadata would include server name, instance id, display name and other cloud specific details. Since there are multiple ways to provide this data (each cloud solution seems to prefer its own way) internally a datasource abstract class was created to allow for a single way to access the different cloud systems methods to provide this data through the typical usage of subclasses.

The current interface that a datasource object must provide is the following:

```
# returns a mime multipart message that contains
# all the various fully-expanded components that
# were found from processing the raw userdata string
# - when filtering only the mime messages targeting
#   this instance id will be returned (or messages with
#   no instance id)
def get_userdata(self, apply_filter=False)

# returns the raw userdata string (or none)
def get_userdata_raw(self)

# returns a integer (or none) which can be used to identify
# this instance in a group of instances which are typically
# created from a single command, thus allowing programatic
# filtering on this launch index (or other selective actions)
@property
def launch_index(self)

# the data sources' config_obj is a cloud-config formated
# object that came to it from ways other than cloud-config
# because cloud-config content would be handled elsewhere
def get_config_obj(self)

#returns a list of public ssh keys
def get_public_ssh_keys(self)

# translates a device 'short' name into the actual physical device
# fully qualified name (or none if said physical device is not attached
# or does not exist)
def device_name_to_device(self, name)

# gets the locale string this instance should be applying
# which typically used to adjust the instances locale settings files
def get_locale(self)

@property
def availability_zone(self)

# gets the instance id that was assigned to this instance by the
# cloud provider or when said instance id does not exist in the backing
# metadata this will return 'iid-datasource'
def get_instance_id(self)

# gets the fully qualified domain name that this host should  be using
# when configuring network or hostname releated settings, typically
# assigned either by the cloud provider or the user creating the vm
def get_hostname(self, fqdn=False)

def get_package_mirror_info(self)
```

EC2

The EC2 datasource is the oldest and most widely used datasource that cloud-init supports. This datasource interacts with a *magic* ip that is provided to the instance by the cloud provider. Typically this ip is `169.254.169.254` of which at this ip a http server is provided to the instance so that the instance can make calls to get instance userdata and instance metadata.

Metadata is accessible via the following URL:

```
GET http://169.254.169.254/2009-04-04/meta-data/
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
hostname
instance-id
instance-type
local-hostname
local-ipv4
placement/
public-hostname
public-ipv4
public-keys/
reservation-id
security-groups
```

Userdata is accessible via the following URL:

```
GET http://169.254.169.254/2009-04-04/user-data
1234,fred,reboot,true | 4512,jimbo, | 173,,,
```

Note that there are multiple versions of this data provided, cloud-init by default uses **2009-04-04** but newer versions can be supported with relative ease (newer versions have more data exposed, while maintaining backward compatibility with the previous versions).

To see which versions are supported from your cloud provider use the following URL:

```
GET http://169.254.169.254/
1.0
2007-01-19
2007-03-01
2007-08-29
2007-10-10
2007-12-15
2008-02-01
2008-09-01
2009-04-04
...
latest
```

Config Drive

The configuration drive datasource supports the OpenStack configuration drive disk.

> See the config drive extension and introduction in the public documentation for more information.

By default, cloud-init does *always* consider this source to be a full-fledged datasource. Instead, the typical behavior is to assume it is really only present to provide networking information. Cloud-init will copy off the network information, apply it to the system, and then continue on. The “full” datasource could then be found in the EC2 metadata service. If this is not the case then the files contained on the located drive must provide equivalents to what the EC2 metadata service would provide (which is typical of the version 2 support listed below)

Version 1

The following criteria are required to as a config drive:

1. Must be formatted with vfat filesystem
2. Must be a un-partitioned block device (/dev/vdb, not /dev/vdb1)
3. Must contain *one* of the following files

```
/etc/network/interfaces
/root/.ssh/authorized_keys
/meta.js
```

`/etc/network/interfaces`

This file is laid down by nova in order to pass static networking information to the guest. Cloud-init will copy it off of the config-drive and into /etc/network/interfaces (or convert it to RH format) as soon as it can, and then attempt to bring up all network interfaces.

`/root/.ssh/authorized_keys`

This file is laid down by nova, and contains the ssk keys that were provided to nova on instance creation (nova-boot –key ....)

`/meta.js`

meta.js is populated on the config-drive in response to the user passing "meta flags" (nova boot –meta key=value ...). It is expected to be json formatted.

Version 2

The following criteria are required to as a config drive:

1. Must be formatted with vfat or iso9660 filesystem or have a *filesystem* label of **config-2**
2. Must be a un-partitioned block device (/dev/vdb, not /dev/vdb1)
3. The files that will typically be present in the config drive are:

```
openstack/
  - 2012-08-10/ or latest/
    - meta_data.json
    - user_data (not mandatory)
  - content/
    - 0000 (referenced content files)
    - 0001
    - ....
ec2
  - latest/
    - meta-data.json (not mandatory)
```

Keys and values

Cloud-init's behavior can be modified by keys found in the meta.js (version 1 only) file in the following ways.

```
dsmode:
  values: local, net, pass
  default: pass
```

This is what indicates if configdrive is a final data source or not. By default it is 'pass', meaning this datasource should not be read. Set it to 'local' or 'net' to stop cloud-init from continuing on to search for other data sources after network config.

The difference between 'local' and 'net' is that local will not require networking to be up before user-data actions (or boothooks) are run.

```
instance-id:
  default: iid-dsconfigdrive
```

This is utilized as the metadata's instance-id. It should generally be unique, as it is what is used to determine "is this a new instance".

```
public-keys:
  default: None
```

If present, these keys will be used as the public keys for the instance. This value overrides the content in authorized_keys.

Note: it is likely preferable to provide keys via user-data

```
user-data:
  default: None
```

This provides cloud-init user-data. See *examples* for what all can be present here.

OpenNebula

The OpenNebula (ON) datasource supports the contextualization disk.

> See contextualization overview, contextualizing VMs and network configuration in the public documentation for more information.

OpenNebula's virtual machines are contextualized (parametrized) by CD-ROM image, which contains a shell script *context.sh* with custom variables defined on virtual machine start. There are no fixed contextualization variables, but the datasource accepts many used and recommended across the documentation.

Datasource configuration

Datasource accepts following configuration options.

```
dsmode:
  values: local, net, disabled
  default: net
```

Tells if this datasource will be processed in 'local' (pre-networking) or 'net' (post-networking) stage or even completely 'disabled'.

```
parseuser:
  default: nobody
```

Unprivileged system user used for contextualization script processing.

Contextualization disk

The following criteria are required:

1. Must be formatted with iso9660 filesystem or have a *filesystem* label of **CONTEXT** or **CDROM**
2. Must contain file *context.sh* with contextualization variables. File is generated by OpenNebula, it has a KEY='VALUE' format and can be easily read by bash

Contextualization variables

There are no fixed contextualization variables in OpenNebula, no standard. Following variables were found on various places and revisions of the OpenNebula documentation. Where multiple similar variables are specified, only first found is taken.

```
DSMODE
```

Datasource mode configuration override. Values: local, net, disabled.

```
DNS
ETH<x>_IP
ETH<x>_NETWORK
ETH<x>_MASK
ETH<x>_GATEWAY
ETH<x>_DOMAIN
ETH<x>_DNS
```

Static network configuration.

```
HOSTNAME
```

Instance hostname.

```
PUBLIC_IP
IP_PUBLIC
ETH0_IP
```

If no hostname has been specified, cloud-init will try to create hostname from instance's IP address in 'local' dsmode. In 'net' dsmode, cloud-init tries to resolve one of its IP addresses to get hostname.

```
SSH_KEY
SSH_PUBLIC_KEY
```

One or multiple SSH keys (separated by newlines) can be specified.

```
USER_DATA
USERDATA
```

cloud-init user data.

Example configuration

This example cloud-init configuration (*cloud.cfg*) enables OpenNebula datasource only in 'net' mode.

```
disable_ec2_metadata: True
datasource_list: ['OpenNebula']
datasource:
  OpenNebula:
    dsmode: net
    parseuser: nobody
```

Example VM's context section

```
CONTEXT=[
  PUBLIC_IP="$NIC[IP]",
  SSH_KEY="$USER[SSH_KEY]
$USER[SSH_KEY1]
$USER[SSH_KEY2] ",
  USER_DATA="#cloud-config
# see https://help.ubuntu.com/community/CloudInit

packages: []

mounts:
- [vdc,none,swap,sw,0,0]
runcmd:
- echo 'Instance has been configured by cloud-init.' | wall
" ]
```

Alt cloud

The datasource altcloud will be used to pick up user data on RHEVm and vSphere.

RHEVm

For RHEVm v3.0 the userdata is injected into the VM using floppy injection via the RHEVm dashboard "Custom Properties".

The format of the Custom Properties entry must be:

```
floppyinject=user-data.txt:<base64 encoded data>
```

For example to pass a simple bash script:

```
% cat simple_script.bash
#!/bin/bash
echo "Hello Joe!" >> /tmp/JJV_Joe_out.txt

% base64 < simple_script.bash
IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gSm9lISIgPj4gL3RtcC9KSlZfSm9lX291dC50eHQK
```

To pass this example script to cloud-init running in a RHEVm v3.0 VM set the "Custom Properties" when creating the RHEMv v3.0 VM to:

```
floppyinject=user-data.
↪txt:IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gSm9lISIgPj4gL3RtcC9KSlZfSm9lX291dC50eHQK
```

NOTE: The prefix with file name must be: `floppyinject=user-data.txt:`

It is also possible to launch a RHEVm v3.0 VM and pass optional user data to it using the Delta Cloud.

For more information on Delta Cloud see: http://deltacloud.apache.org

vSphere

For VMWare's vSphere the userdata is injected into the VM as an ISO via the cdrom. This can be done using the vSphere dashboard by connecting an ISO image to the CD/DVD drive.

To pass this example script to cloud-init running in a vSphere VM set the CD/DVD drive when creating the vSphere VM to point to an ISO on the data store.

Note: The ISO must contain the user data.

For example, to pass the same `simple_script.bash` to vSphere:

Create the ISO

```
% mkdir my-iso
```

NOTE: The file name on the ISO must be: `user-data.txt`

```
% cp simple_scirpt.bash my-iso/user-data.txt
% genisoimage -o user-data.iso -r my-iso
```

Verify the ISO

```
% sudo mkdir /media/vsphere_iso
% sudo mount -o loop JoeV_CI_02.iso /media/vsphere_iso
% cat /media/vsphere_iso/user-data.txt
% sudo umount /media/vsphere_iso
```

Then, launch the vSphere VM the ISO user-data.iso attached as a CDROM.

It is also possible to launch a vSphere VM and pass optional user data to it using the Delta Cloud.

For more information on Delta Cloud see: http://deltacloud.apache.org

No cloud

The data source `NoCloud` and `NoCloudNet` allow the user to provide user-data and meta-data to the instance without running a network service (or even without having a network at all).

You can provide meta-data and user-data to a local vm boot via files on a vfat or iso9660 filesystem. The filesystem volume label must be `cidata`.

These user-data and meta-data files are expected to be in the following format.

```
/user-data
/meta-data
```

Basically, user-data is simply user-data and meta-data is a yaml formatted file representing what you'd find in the EC2 metadata service.

Given a disk ubuntu 12.04 cloud image in 'disk.img', you can create a sufficient disk by following the example below.

```
## create user-data and meta-data files that will be used
## to modify image on first boot
$ { echo instance-id: iid-local01; echo local-hostname: cloudimg; } > meta-data

$ printf "#cloud-config\npassword: passw0rd\nchpasswd: { expire: False }\nssh_pwauth:␣
↪True\n" > user-data

## create a disk to attach with some user-data and meta-data
$ genisoimage  -output seed.iso -volid cidata -joliet -rock user-data meta-data

## alternatively, create a vfat filesystem with same files
## $ truncate --size 2M seed.img
## $ mkfs.vfat -n cidata seed.img
## $ mcopy -oi seed.img user-data meta-data ::

## create a new qcow image to boot, backed by your original image
$ qemu-img create -f qcow2 -b disk.img boot-disk.img

## boot the image and login as 'ubuntu' with password 'passw0rd'
## note, passw0rd was set as password through the user-data above,
## there is no password set on these images.
$ kvm -m 256 \
   -net nic -net user,hostfwd=tcp::2222-:22 \
   -drive file=boot-disk.img,if=virtio \
   -drive file=seed.iso,if=virtio
```

Note: that the instance-id provided (`iid-local01` above) is what is used to determine if this is "first boot". So if you are making updates to user-data you will also have to change that, or start the disk fresh.

Also, you can inject an `/etc/network/interfaces` file by providing the content for that file in the `network-interfaces` field of metadata.

Example metadata:

```
instance-id: iid-abcdefg
network-interfaces: |
  iface eth0 inet static
  address 192.168.1.10
  network 192.168.1.0
  netmask 255.255.255.0
  broadcast 192.168.1.255
  gateway 192.168.1.254
hostname: myhost
```

MAAS

TODO

For now see: http://maas.ubuntu.com/

CloudStack

Apache CloudStack expose user-data, meta-data, user password and account sshkey thru the Virtual-Router. For more details on meta-data and user-data, refer the CloudStack Administrator Guide.

URLs to access user-data and meta-data from the Virtual Machine. Here 10.1.1.1 is the Virtual Router IP:

```
http://10.1.1.1/latest/user-data
http://10.1.1.1/latest/meta-data
http://10.1.1.1/latest/meta-data/{metadata type}
```

Configuration

Apache CloudStack datasource can be configured as follows:

```
datasource:
  CloudStack: {}
  None: {}
datasource_list:
  - CloudStack
```

OVF

TODO

For now see: https://bazaar.launchpad.net/~cloud-init-dev/cloud-init/trunk/files/head:/doc/sources/ovf/

OpenStack

TODO

Vendor Data

The OpenStack metadata server can be configured to serve up vendor data which is available to all instances for consumption. OpenStack vendor data is, generally, a JSON object.

cloud-init will look for configuration in the `cloud-init` attribute of the vendor data JSON object. cloud-init processes this configuration using the same handlers as user data, so any formats that work for user data should work for vendor data.

For example, configuring the following as vendor data in OpenStack would upgrade packages and install `htop` on all instances:

```
{"cloud-init": "#cloud-config\npackage_upgrade: True\npackages:\n - htop"}
```

For more general information about how cloud-init handles vendor data, including how it can be disabled by users on instances, see https://bazaar.launchpad.net/~cloud-init-dev/cloud-init/trunk/view/head:/doc/vendordata.txt

Fallback/None

This is the fallback datasource when no other datasource can be selected. It is the equivalent of a *empty* datasource in that it provides a empty string as userdata and a empty dictionary as metadata. It is useful for testing as well as for when you do not have a need to have an actual datasource to meet your instance requirements (ie you just want to run modules that are not concerned with any external data). It is typically put at the end of the datasource search list so that if all other datasources are not matched, then this one will be so that the user is not left with an inaccessible instance.

Note: the instance id that this datasource provides is `iid-datasource-none`.

Modules

Apt Configure

Internal name: `cc_apt_configure`

Apt Pipelining

Internal name: `cc_apt_pipelining`

Bootcmd

Internal name: `cc_bootcmd`

Byobu

Internal name: `cc_byobu`

Ca Certs

Internal name: `cc_ca_certs`

Chef

Internal name: `cc_chef` **Summary:** module that configures, starts and installs chef.

Description: This module enables chef to be installed (from packages or from gems, or from omnibus). Before this occurs chef configurations are written to disk (validation.pem, client.pem, firstboot.json, client.rb), and needed chef folders/directories are created (/etc/chef and /var/log/chef and so-on). Then once installing proceeds correctly if configured chef will be started (in daemon mode or in non-daemon mode) and then once that has finished (if ran in non-daemon mode this will be when chef finishes converging, if ran in daemon mode then no further actions are possible since chef will have forked into its own process) then a post run function can run that can do finishing activities (such as removing the validation pem file).

It can be configured with the following option structure:

```
chef:
   directories: (defaulting to /etc/chef, /var/log/chef, /var/lib/chef,
                 /var/cache/chef, /var/backups/chef, /var/run/chef)
   validation_cert: (optional string to be written to file validation_key)
                    special value 'system' means set use existing file
   validation_key: (optional the path for validation_cert. default
                    /etc/chef/validation.pem)
   firstboot_path: (path to write run_list and initial_attributes keys that
                    should also be present in this configuration, defaults
                    to /etc/chef/firstboot.json)
   exec: boolean to run or not run chef (defaults to false, unless
                                         a gem installed is requested
                                         where this will then default
                                         to true)

chef.rb template keys (if falsey, then will be skipped and not
                       written to /etc/chef/client.rb)

chef:
  client_key:
  environment:
  file_backup_path:
  file_cache_path:
  json_attribs:
  log_level:
  log_location:
  node_name:
  pid_file:
  server_url:
  show_time:
  ssl_verify_mode:
  validation_cert:
  validation_key:
  validation_name:
```

`cloudinit.config.cc_chef.`**`handle`**(*name*, *cfg*, *cloud*, *log*, *_args*)
  Handler method activated by cloud-init.

Debug

Internal name: `cc_debug` **Summary:** helper to debug cloud-init *internal* datastructures.

Description: This module will enable for outputting various internal information that cloud-init sources provide to either a file or to the output console/log location that this cloud-init has been configured with when running.

It can be configured with the following option structure:

```
debug:
   verbose: (defaulting to true)
   output: (location to write output, defaulting to console + log)
```

Note: Log configurations are not output.

`cloudinit.config.cc_debug.`**`handle`**(*name*, *cfg*, *cloud*, *log*, *args*)
  Handler method activated by cloud-init.

Disable Ec2 Metadata

Internal name: `cc_disable_ec2_metadata`

Disk Setup

Internal name: `cc_disk_setup`

Emit Upstart

Internal name: `cc_emit_upstart`

Final Message

Internal name: `cc_final_message`

Foo

Internal name: `cc_foo`

Growpart

Internal name: `cc_growpart`

Grub Dpkg

Internal name: `cc_grub_dpkg`

Keys To Console

Internal name: `cc_keys_to_console`

Landscape

Internal name: `cc_landscape`

Locale

Internal name: `cc_locale`

Mcollective

Internal name: `cc_mcollective`

Migrator

Internal name: `cc_migrator`

Mounts

Internal name: `cc_mounts`

Package Update Upgrade Install

Internal name: `cc_package_update_upgrade_install`

Phone Home

Internal name: `cc_phone_home`

Power State Change

Internal name: `cc_power_state_change`

Puppet

Internal name: `cc_puppet`

Resizefs

Internal name: `cc_resizefs`

Resolv Conf

Internal name: `cc_resolv_conf`

Rightscale Userdata

Internal name: `cc_rightscale_userdata`

Rsyslog

Internal name: `cc_rsyslog` rsyslog module allows configuration of syslog logging via rsyslog Configuration is done under the cloud-config top level 'rsyslog'.

Under 'rsyslog' you can define:

- configs: [default=[]] this is a list. entries in it are a string or a dictionary. each entry has 2 parts:
  - content
  - filename

  if the entry is a string, then it is assigned to 'content'. for each entry, content is written to the provided filename. if filename is not provided, its default is read from 'config_filename'

  Content here can be any valid rsyslog configuration. No format specific format is enforced.

  For simply logging to an existing remote syslog server, via udp: configs: ["*. @192.168.1.1"]

- remotes: [default={}] This is a dictionary of name / value pairs. In comparison to 'config's, it is more focused in that it only supports remote syslog configuration. It is not rsyslog specific, and could convert to other syslog implementations.

  Each entry in remotes is a 'name' and a 'value'.

  - name: an string identifying the entry. good practice would indicate using a consistent and identifiable string for the producer. For example, the MAAS service could use 'maas' as the key.
  - value consists of the following parts: * optional filter for log messages

    default if not present: .

    * optional leading '@' or '@@' (indicates udp or tcp respectively). default if not present (udp): @ This is rsyslog format for that. if not present, is '@'.
    * ipv4 or ipv6 or hostname ipv6 addresses must be in [::1] format. (@[fd00::1]:514)
    * optional port port defaults to 514

- config_filename: [default=20-cloud-config.conf] this is the file name to use if none is provided in a config entry.
- config_dir: [default=/etc/rsyslog.d] this directory is used for filenames that are not absolute paths.
- service_reload_command: [default="auto"] this command is executed if files have been written and thus the syslog daemon needs to be told.

Note, since cloud-init 0.5 a legacy version of rsyslog config has been present and is still supported. See below for the mappings between old value and new value:

> old value -> new value 'rsyslog' -> rsyslog/configs 'rsyslog_filename' -> rsyslog/config_filename 'rsyslog_dir' -> rsyslog/config_dir

the legacy config does not support 'service_reload_command'.

Example config: #cloud-config rsyslog:

> **configs:**
>
> - ". @@192.158.1.1"
> - content: ". @@192.0.2.1:10514" filename: 01-example.conf
> - content: | . @@syslogd.example.com
>
> **remotes:** maas: "192.168.1.1" juju: "10.0.4.1"
>
> config_dir: config_dir config_filename: config_filename service_reload_command: [your, syslog, restart, command]

Example Legacy config: #cloud-config rsyslog:

> - ". @@192.158.1.1"
>
> rsyslog_dir: /etc/rsyslog-config.d/ rsyslog_filename: 99-local.conf

Runcmd

Internal name: `cc_runcmd`

Salt Minion

Internal name: `cc_salt_minion`

Scripts Per Boot

Internal name: `cc_scripts_per_boot`

Scripts Per Instance

Internal name: `cc_scripts_per_instance`

Scripts Per Once

Internal name: `cc_scripts_per_once`

Scripts User

Internal name: `cc_scripts_user`

Scripts Vendor

Internal name: `cc_scripts_vendor`

Seed Random

Internal name: `cc_seed_random`

Set Hostname

Internal name: `cc_set_hostname`

Set Passwords

Internal name: `cc_set_passwords`

Ssh

Internal name: `cc_ssh`

Ssh Authkey Fingerprints

Internal name: `cc_ssh_authkey_fingerprints`

Ssh Import Id

Internal name: `cc_ssh_import_id`

Timezone

Internal name: `cc_timezone`

Ubuntu Init Switch

Internal name: `cc_ubuntu_init_switch` **Summary:** reboot system into another init.

Description: This module provides a way for the user to boot with systemd even if the image is set to boot with upstart. It should be run as one of the first `cloud_init_modules`, and will switch the init system and then issue a reboot. The next boot will come up in the target init system and no action will be taken.

This should be inert on non-ubuntu systems, and also exit quickly.

It can be configured with the following option structure:

```
init_switch:
  target: systemd (can be 'systemd' or 'upstart')
  reboot: true (reboot if a change was made, or false to not reboot)
```

Note: Best effort is made, but it's possible this system will break, and probably won't interact well with any other mechanism you've used to switch the init system.

`cloudinit.config.cc_ubuntu_init_switch.`**`handle`**(*name*, *cfg*, *cloud*, *log*, *args*)
    Handler method activated by cloud-init.

Update Etc Hosts

Internal name: `cc_update_etc_hosts`

Update Hostname

Internal name: `cc_update_hostname`

Users Groups

Internal name: `cc_users_groups`

Write Files

Internal name: `cc_write_files`

Yum Add Repo

Internal name: `cc_yum_add_repo`

Merging User-Data Sections

Overview

This was implemented because it has been a common feature request that there be a way to specify how cloud-config yaml “dictionaries” provided as user-data are merged together when there are multiple yamls to merge together (say when performing an #include).

Since previously the merging algorithm was very simple and would only overwrite and not append lists, or strings, and so on it was decided to create a new and improved way to merge dictionaries (and there contained objects) together in a way that is customizable, thus allowing for users who provide cloud-config user-data to determine exactly how there objects will be merged.

For example.

```
#cloud-config (1)
run_cmd:
  - bash1
  - bash2

#cloud-config (2)
run_cmd:
  - bash3
  - bash4
```

The previous way of merging the following 2 objects would result in a final cloud-config object that contains the following.

```
#cloud-config (merged)
run_cmd:
  - bash3
  - bash4
```

Typically this is not what users want, instead they would likely prefer:

```
#cloud-config (merged)
run_cmd:
  - bash1
  - bash2
  - bash3
  - bash4
```

This way makes it easier to combine the various cloud-config objects you have into a more useful list, thus reducing duplication that would have had to occur in the previous method to accomplish the same result.

Customizability

Since the above merging algorithm may not always be the desired merging algorithm (like how the previous merging algorithm was not always the preferred one) the concept of customizing how merging can be done was introduced through a new concept call ‘merge classes’.

A merge class is a class defintion which provides functions that can be used to merge a given type with another given type.

An example of one of these merging classes is the following:

```
class Merger(object):
    def __init__(self, merger, opts):
        self._merger = merger
        self._overwrite = 'overwrite' in opts

    # This merging algorithm will attempt to merge with
    # another dictionary, on encountering any other type of object
    # it will not merge with said object, but will instead return
    # the original value
    #
    # On encountering a dictionary, it will create a new dictionary
    # composed of the original and the one to merge with, if 'overwrite'
    # is enabled then keys that exist in the original will be overwritten
    # by keys in the one to merge with (and associated values). Otherwise
    # if not in overwrite mode the 2 conflicting keys themselves will
    # be merged.
    def _on_dict(self, value, merge_with):
        if not isinstance(merge_with, (dict)):
            return value
        merged = dict(value)
        for (k, v) in merge_with.items():
            if k in merged:
                if not self._overwrite:
                    merged[k] = self._merger.merge(merged[k], v)
                else:
                    merged[k] = v
            else:
                merged[k] = v
        return merged
```

As you can see there is a '_on_dict' method here that will be given a source value and a value to merge with. The result will be the merged object. This code itself is called by another merging class which 'directs' the merging to happen by analyzing the types of the objects to merge and attempting to find a know object that will merge that type. I will avoid pasting that here, but it can be found in the *mergers/__init__.py* file (see *LookupMerger* and *UnknownMerger*).

So following the typical cloud-init way of allowing source code to be downloaded and used dynamically, it is possible for users to inject there own merging files to handle specific types of merging as they choose (the basic ones included will handle lists, dicts, and strings). Note how each merge can have options associated with it which affect how the merging is performed, for example a dictionary merger can be told to overwrite instead of attempt to merge, or a string merger can be told to append strings instead of discarding other strings to merge with.

How to activate

There are a few ways to activate the merging algorithms, and to customize them for your own usage.

1. The first way involves the usage of MIME messages in cloud-init to specify multipart documents (this is one way in which multiple cloud-config is joined together into a single cloud-config). Two new headers are looked for, both of which can define the way merging is done (the first header to exist wins). These new headers (in lookup order) are 'Merge-Type' and 'X-Merge-Type'. The value should be a string which will satisfy the new merging format defintion (see below for this format).
2. The second way is actually specifying the merge-type in the body of the cloud-config dictionary. There are 2 ways to specify this, either as a string or as a dictionary (see format below). The keys that are looked up for this definition are the following (in order), 'merge_how', 'merge_type'.

String format

The string format that is expected is the following.

```
classname1(option1,option2)+classname2(option3,option4)....
```

The class name there will be connected to class names used when looking for the class that can be used to merge and options provided will be given to the class on construction of that class.

For example, the default string that is used when none is provided is the following:

```
list()+dict()+str()
```

Dictionary format

In cases where a dictionary can be used to specify the same information as the string format (ie option #2 of above) it can be used, for example.

```
{'merge_how': [{'name': 'list', 'settings': ['extend']},
               {'name': 'dict', 'settings': []},
               {'name': 'str', 'settings': ['append']}]}
```

This would be the equivalent format for default string format but in dictionary form instead of string form.

Specifying multiple types and its effect

Now you may be asking yourself, if I specify a merge-type header or dictionary for every cloud-config that I provide, what exactly happens?

The answer is that when merging, a stack of 'merging classes' is kept, the first one on that stack is the default merging classes, this set of mergers will be used when the first cloud-config is merged with the initial empty cloud-config dictionary. If the cloud-config that was just merged provided a set of merging classes (via the above formats) then those merging classes will be pushed onto the stack. Now if there is a second cloud-config to be merged then the merging classes from the cloud-config before the first will be used (not the default) and so on. This way a cloud-config can decide how it will merge with a cloud-config dictionary coming after it.

Other uses

In addition to being used for merging user-data sections, the default merging algorithm for merging 'conf.d' yaml files (which form an initial yaml config for cloud-init) was also changed to use this mechanism so its full benefits (and customization) can also be used there as well. Other places that used the previous merging are also, similarly, now extensible (metadata merging, for example).

Note, however, that merge algorithms are not used *across* types of configuration. As was the case before merging was implemented, user-data will overwrite conf.d configuration without merging.

More information

Useful external references

- The beauty of cloudinit

- Introduction to cloud-init (video)

Hacking on cloud-init

This document describes how to contribute changes to cloud-init.

Do these things once

- If you have not already, be sure to sign the CCA:
  - Canonical Contributor Agreement
- Clone the LaunchPad repository:

  git clone YOUR_USERNAME@git.launchpad.net:cloud-init cd cloud-init

  If you would prefer a bzr style *git clone lp:cloud-init*, see the Instructions on LaunchPad for more information.
- Create a new remote pointing to your personal LaunchPad repository:

```
git remote add YOUR_USERNAME YOUR_USERNAME@git.launchpad.net:~YOUR_USERNAME/cloud-
↪init
```

Do these things for each feature or bug

- Create a new topic branch for your work:

```
git checkout -b my-topic-branch
```

- Make and commit your changes (note, you can make multiple commits, fixes, more commits.):

```
git commit
```

- Check pep8 and test, and address any issues:

```
make test pep8
```

- Push your changes to your personal LaunchPad repository:

```
git push -u YOUR_USERNAME my-topic-branch
```

- Use your browser to create a merge request:
  - Open the branch on LaunchPad
    - It will typically be at `https://code.launchpad.net/~YOUR_USERNAME/cloud-init/+git/cloud-init/+ref/BRANCHNAME` for example https://code.launchpad.net/~larsks/cloud-init/+git/cloud-init/+ref/feature/move-to-git
  - Click 'Propose for merging'
  - Select `cloud-init` as the target repository
  - Select `master` as the target reference path

Then, someone on cloud-init-dev (currently Scott Moser and Joshua Harlow) will review your changes and follow up in the merge request.

Feel free to ping and/or join `#cloud-init` on freenode (irc) if you have any questions.

Python Module Index

Index