
Yaybu Documentation

Release 3.0

John Carr, Doug Winter

September 04, 2013

CONTENTS

Yaybu is a push based configuration management tool written in Python with the goal of helping you tame your servers. You describe your infrastructure in a simple and flexible YAML-like language and Yaybu works out what needs to happen to deploy your updates.

Contents:

INSTALLING YAYBU

1.1 Latest stable release

1.1.1 Ubuntu

The latest release is packaged as deb packages and is available via a PPA for recent versions of Ubuntu:

```
sudo add-apt-repository ppa:yaybu-team/stable
sudo apt-get update
sudo apt-get install python-yaybu
```

1.1.2 OSX

A `.dmg` is available from the [releases](#) page at GitHub.

Drag the Yaybu icon into your Applications folder. When you first run Yaybu it will prompt you to install command line tools. This will simply create a symlink from `/usr/local/bin/yaybu` to command line tools embedded inside the Yaybu bundle.

You can drop `Yaybufile` files onto the Yaybu dock icon to automatically start a Yaybu shell for a project.

1.2 Nightlies

1.2.1 Ubuntu

An unstable ‘nightly’ PPA is available for lucid and precise. You can use it like this:

```
sudo add-apt-repository ppa:yaybu-team/nightly
sudo apt-get update
sudo apt-get install python-yaybu
```

1.2.2 OSX

The latest build is available from [here](#). Install it like you would install a stable version.

It’s automatic update feed is pointed at the nightlies channel.

QUICKSTART

Here are some quick and simple Yaybu examples to show you what you can do right now and we are working on.

2.1 Yaybufile

To use Yaybu you first need to write a Yaybufile. This describes the infrastructure you want to deploy.

Here is an example that provisions 2 compute nodes with different hosting providers and sets up subdomains for them. Yaybu is quite happy talking to Amazon EC2, BigV and Gandi DNS all from the same deployment:

```
new Provisioner as instance1:
  new Compute as server:
    driver:
      id: BIGV
      key: yourusername
      secret: yourpassword
      account: youraccountname

    image: precise
    name: test_at_bigv

    user: root
    password: aez5Eep4

  resources:
    - File:
      name: /etc/heartbeat.conf
      template: heartbeat.conf.j2
      template_args:
        partner: {{ instance2.public_ip }}

new Provisioner as instance2:
  new Compute as server:
    driver:
      id: EC2
      key: yourusername
      secret: yourpassword

    image: ami-57afb223
    name: test_at_ec2

    user: root
    public_key: instance2.pub
```

```
    private_key: instance2.priv

resources:
  - File:
    name: /etc/heartbeat.conf
    template: heartbeat.conf.j2
    template_args:
      partner: {{ instance1.public_ip }}

new Zone as dns:
  driver:
    id: GANDI
    key: yourgandikey

  domain: example.com

records:
  - name: instance1
    data: {{ instance1.server.public_ip }}
  - name: instance2
    data: {{ instance2.server.public_ip }}
```

2.2 Yaybu commands

Currently the following commands are available:

yaybu up Apply the configuration specified in your Yaybufile

yaybu destroy If your configuration creates external resources like virtual machines, then this command will destroy it.

yaybu expand Print out a YAML dump of your configuration after all variables have been expanded and any ifs/fors/etc have been applied.

yaybu ssh SSH into a server using the connection details specified in your configuration file.

You can do `yaybu help COMMAND` to learn more about each of these.

2.3 Yaybu parts

Parts are the building blocks that you connect together to describe your services and how to deploy them. There are several core ones at the moment.

2.3.1 Compute

The `Compute` part can be used to create and destroy services in various cloud services supported by libcloud.

2.3.2 Provisioner

The `Provisioner` part provides idempotent configuration of UNIX servers that can be accessed by SSH. It can be connected to `Compute` part to create and deploy to a new cloud server, or it can be pointed at a static set of SSH connection details to deploy to a dedicated server.

The part needs connection details, these are provided through the `server` parameter:

```
new Provisioner as provisioner:
  server:
    fqdn: example.com
    port: 22
    username: root
    password: penguin55
    private_key: path/to/id_rsa
```

The part deploys a list of resources provided by the `resources` parameter. These are idempotent - when used correctly they only make changes that need making, which means that you can see quite clearly what has been changed by an update deployment and it is safe to run repeatedly.

For detailed documentation of the resources you can see the [online documentation](#).

2.3.3 Zone

The `Zone` part uses the libcloud DNS API to manage DNS entries in various cloud services.

2.4 Keeping secrets secret

You can reference encrypted yay files in your Yaybufile:

```
include "mysecrets.yay.gpg"
```

Any include of a `.gpg` file is automatically decrypted, using your `gpg-agent` to prompt for any passphrases that are required.

Additionally the file `~/ .yaybu/defaults.yay.gpg` is automatically loaded when Yaybu starts. This is useful for storing your credentials/tokens outside of your code repository and easily injected them into multiple projects.

For vim users, [vim-gnupg](#) is a gret way to transparently edit your GPG armored configuration files.

COMMAND LINE

The main command line in `yaybu` is `yaybu`. If you run it with no arguments it will drop you into an interactive shell where you can run the various subcommands.

Most subcommands will need a config file. By default `yaybu` will search for a `Yaybufile` in the current directory. It will also check in parent directories. If you want to force it to use a different config file you can use the `-c` option:

```
yaybu -c myconfig.yay test
```

Your configuration can declare variables that it takes on the command line. This is covered in *Runtime arguments*. If you have defined an argument called `instance` then you can use it with a command like this:

```
yaybu up instance=test1234
```

3.1 expand

Running `yaybu expand` evaluates your configuration without making any changes and prints to stdout a YAML or JSON representation. This is useful for checking that your loops, conditionals and variables have been evaluated as you expected by inspection.

3.2 test

The `yaybu test` command will run various tests on your configuration to try and find errors before you do an actual deployment. We can't guarantee success but we can:

- Check any assets you depend on exist
- Check your templates for errors
- Check that any dependencies or relationships you have created between parts or resources are valid
- Check your credentials for all the services you will be accessing

3.3 up

`yaybu up` is the command that will actually deploy your configuration.

In order to protect your infrastructure this will automatically perform a self-test of your configuration before starting.

If you haven't defined any parts in your configuration then this command will not do anything.

This command takes `--resume` or `--no-resume`. These flags control whether or not yaybu remembers trigger states between deployments. This is covered in more detail in the *Provisioner* section.

3.4 destroy

When you run `yaybu destroy` yaybu will examine all the parts you have defined in your configuration and ask them to destroy themselves:

```
$ yaybu destroy
[*] Destroying load balancer 'test_lb'
[*] Destroying node 'test1'
[*] Destroying node 'test2'
```

3.5 ssh

The `yaybu ssh` takes an expression and solves it against the configuration. For example, if you had a list of servers you could:

```
yaybu ssh myservers[2]
```

This would start an SSH connection to the 3rd server in the list (the first server is at position 0).

If Yaybu is aware of a particular username, port, password or SSH key needed to access that server it will use it. Otherwise it will fall back to using for SSH agent.

RUNTIME ARGUMENTS

By setting `yaybu.options` you can allow some parts of your configuration to be set at runtime. These are then available in the `yaybu.argv` dictionary.

4.1 Defining arguments

4.1.1 Strings

The `string` argument type is the simplest. You need to specify a name and can optionally set a default:

```
yaybu:
  options:
    - name: username
      type: string
      default: john
```

A `string` is actually the default type of argument. So you don't need to specify the `type`:

```
yaybu:
  options:
    - name: username
      default: john
```

4.1.2 Integer

The `integer` argument validates that the argument provided by your end user is indeed a valid integer. It can be defined like this:

```
yaybu:
  options:
    - name: num_servers
      type: integer
      default: 1
```

4.1.3 Boolean

The `boolean` argument type takes a value of `no`, `0`, `off` or `false` and interprets it as a negative. `yes`, `1`, `on` or `true` is interpreted as a positive. Other values trigger validation. You can use it like this:

```
yaybu:
  options:
    - name: on_off_toggle
      type: boolean
      default: on
```

4.2 Using arguments

The arguments defined via `yaybu.options` are available at runtime using the `yaybu.argv` mapping.

One use of this is to combine it with the *Compute* part to create a configuration that can be deployed multiple times with no changes:

```
yaybu:
  options:
    - name: instance
      default: cloud

new Compute as server:
  name: myproject-{{ yaybu.argv.instance }}
  driver:
    id: EC2
    key: secretkey
    secret: secretsecret
  image: imageid
  size: t1.micro
```

If i were to run this configuration several times:

```
yaybu up
yaybu up instance=take2
yaybu up instance=take3
```

Then i would have 3 instances running:

- myproject-cloud
- myproject-take2
- myproject-take3

COMPUTE INSTANCES

The `Compute` part can be used to create and destroy services in various cloud services supported by `libcloud` as well as various local VM tools.

Creating a simple compute node will look something like this:

```
new Compute as server:
  name: test123456

  driver:
    id: BIGV
    key: yourusername
    secret: yourpassword
    account: youraccountname

  image: precise

  user: root
  password: aez5Eep4
```

In this example we are creating a server via `BigV`, but because our cloud support is underpinned by `libcloud` we support many hosting providers.

5.1 Options

Any compute instances you create must have a unique name. This lets `yaybu` keep track of it between `yaybu apply` invocations.

Use the `driver` argument to configure a `libcloud` driver for your hosting service. Specific driver sub arguments are discussed in the sections below.

You can choose an base image using the `image` argument. For the common case an image id is enough:

```
new Compute as server:
  image: ami-f7445d83
```

You can choose an instance size by passing a size name:

```
new Compute as server:
  size: t1.micro
```

Some servers don't have the concept of size but you can control the resources assigned in a more granular way:

```
new Computer as server:
  size:
    processors: 5
```

See the driver specific options below for more information on what tweaks you can pass to a backend.

You must choose a username that can be used to log in with.

If you provide a `public_key` file and are using a driver that supports it Yaybu will automatically load it into the created instance to enable key based authentication.

If you provide a `password` and the backend supports it then Yaybu will automatically set the account password for the newly created instance.

The `Compute` part does not look at the `private_key` attribute, but as it is common to use the `Compute` part directly with a `Provisioner` part, which does check for it, you will often see it specified:

```
new Provisioner as vml:
  new Compute as server:
    private_key: path/to/privatekey
```

5.2 Supported services

5.2.1 BigV

Our `BigV` support is implemented via [the libcloud library](#) but is currently residing in the Yaybu codebase. As you can set the password for an instance when it is created there is no preparation to do to create a bigv instance, other than creating a bigv account.

Your Yaybufile looks like this:

```
new Provisioner as vml:
  new Compute as server:
    name: test123456

    driver:
      id: BIGV
      key: yourusername
      secret: yourpassword
      account: youraccountname

    image: precise

    user: root
    password: aez5Eep4

  resources:
    - Package:
      name: git-core
```

This example will create a new vm called `test123456`. You will be able to log in as root using the password `aez5Eep4` (though you should use `pwgen` to come up with something better).

5.2.2 EC2

Provisioning of AWS instances is supported out of the box using libcloud. You will need to have set up an SSH key in the Amazon control panel and either have the path to the private part of that key or have added it to your ssh-agent.

You'll need something like this in your `Yaybufile`:

```
new Compute as server:
  name: myappserver

  driver:
    id: EC2_EU_WEST
    key: mykey
    secret: mysecret

  size: t1.micro
  image: ami-4f504f3b

  user: ubuntu
  ex_keyname: mykey
  private_key: mykey.pem
```

`ex_keyname` is the name of the SSH key pair in the amazon console. `private_key` is the corresponding private key.

We recently merged a patch upstream to do away with `ex_keyname`. In future Yaybu will be able to automatically upload a `public_key` for you in the same way it can for other backends.

5.2.3 VMWare

You'll need a copy of VMWare Workstation, VMWare Fusion or VMWare Player. You'll need a base image to use. My checklist when creating mine is:

- Is `openssh-server` installed?
- Is there a user with passphraseless `sudo` access to root?
- Have I deleted the `/etc/udev/rules.d/70-persistent-net.rules`?

When you are done, shut down the VM and get the path to its VMX file.

Now your `Yaybufile` looks like this:

```
new Compute as server:
  name: mytest vm

  driver:
    id: VMWARE

  image:
    id: ~/vmware/ubuntu/ubuntu.vmx

  user: ubuntu
```

5.3 Community supported services

By using libcloud to support the services in the previous section, the following services are also available. Please adopt your favourite and help improve documentation for it.

5.3.1 Cloudstack

The driver id for `CloudStack` is `CLOUDSTACK`:

```
new Compute as server:
  name: new_cloudstack_server

  driver:
    id: CLOUDSTACK
    host: yourcloudstackhost.com
    path: /api/2.0
    key: yourkey
    secret: yoursecret

  image: yourimageid
  size: yoursized
```

Note: The `CloudStack` `libcloud` driver could be updated to allow the user to inject SSH keys, but this is not currently in progress.

5.3.2 Digital Ocean

The driver if for `Digital Ocean` is `DIGITAL_OCEAN`:

```
new Compute as server:
  name: new_digital_ocean_server

  driver:
    id: DIGITAL_OCEAN
    key: yourkey
    secret: yoursecret

  image: yourimageid
  size: yoursized
```

Note: The `Digital Ocean` `libcloud` driver could be updated to allow the user to inject SSH keys, but this is not currently in progress.

5.3.3 Gandi

The driver id for `Gandi` is `GANDI`:

```
new Compute as server:
  name: new_gandi_server

  driver:
    id: GANDI
    key: yourkey
    secret: yoursecret

  image: yourimageid
  size: yoursized
```

5.3.4 GoGrid

5.3.5 IBM SCE

5.3.6 Linode

5.3.7 OpenStack

5.3.8 Rackspace

5.3.9 SoftLayer

5.3.10 And more

The libcloud project supports a lot of compute services. The goal is that any cloud service supported by libcloud can be controlled using Yaybu, and any fixes to improve that support will be pushed upstream.

5.4 Adding support for your other hosting services

Depending on what you are doing there are different requirements.

If you have preprepared images and simply want to stop and start them then the only requirement is that you are using a version of libcloud that supports that service (and exposes it as a public driver).

If you want to use your hosting service in conjunction with a Provisioner part you will additionally need:

- SSH to be installed and working in the base image you choose.
- **You have credentials that can obtain root access**
 - Either the service lets you set a password/SSH key at create time
 - Or the base image has credentials baked into it that you can use

PROVISIONER

The `Provisioner` part provides idempotent configuration of UNIX servers that can be accessed by SSH. It can be connected to `Compute` part to create and deploy to a new cloud server, or it can be pointed at a static set of SSH connection details to deploy to a dedicated server.

The part needs connection details, these are provided through the `server` parameter:

```
new Provisioner as provisioner:
  server:
    fqdn: example.com
    port: 22
    username: root
    password: penguin55
    private_key: path/to/id_rsa

  resources:
    - File:
      name: /etc/my.cnf
      template: mytemplate.j2
      template_args:
        hello: world
```

To provision to a server, `Yaybu` needs to be able to access it. In particular you **MUST** make sure that:

- `Yaybu` has passwordless access over ssh to the server.
- `Yaybu` has passwordless access to `sudo`. The best way to achieve this is to ensure you are in the appropriate group ('`admin`' or '`sudo`' on Ubuntu for example, depending on which version). Then add the `NOPASSWD:` directive to the appropriate group.

6.1 Options

You specify a list of `resources` to apply to a designated `server`.

The `resources` are specified as a list of simple files, directories, users, etc that are executed in order:

```
resources:
  - File:
    name: /etc/my.cnf
    static: staticfile.cnf

  - User:
    name: django
```

You can pass the following settings to the `server` argument:

fqdn A fully qualified domain name to connect to (via SSH). An IP can also be used if required.

port The port to connect to. This is optional, and port 22 will be used if not provided.

username The ssh username to login as. If this isn't root then Yaybu will attempt to use sudo when it requires root access to perform a task.

password The ssh password to login with.

private_key An RSA or DSA private key that can be used to log in to the target server.

resources The provisioner part expresses server configuration in units called "resources". These are things like files, init.d services or unix accounts.

If you do not provide a `private_key` or a `password` Yaybu will fallback to trying keys in your ssh keyring. If you provide both then it will prefer to use a password.

6.2 Built-in resources

This section describes the built-in resources you can use to describe your server configuration.

6.2.1 File

A provider for this resource will create or amend an existing file to the provided specification.

For example, the following will create the `/etc/hosts` file based on a static local file:

```
extend resources:
  - File:
    name: /etc/hosts
    owner: root
    group: root
    mode: 644
    static: my_hosts_file
```

The following will create a file using a jinja2 template, and will back up the old version of the file if necessary:

```
extend resources:
  - File:
    name: /etc/email_addresses
    owner: root
    group: root
    mode: 644
    template: email_addresses.j2
    template_args:
      foo: foo@example.com
      bar: bar@example.com
    backup: /etc/email_addresses.{year}-{month}-{day}
```

The available parameters are:

name The full path to the file this resource represents.

owner A unix username or UID who will own created objects. An owner that begins with a digit will be interpreted as a UID, otherwise it will be looked up using the python 'pwd' module.

group A unix group or GID who will own created objects. A group that begins with a digit will be interpreted as a GID, otherwise it will be looked up using the python 'grp' module.

mode A mode representation as an octal. This can begin with leading zeros if you like, but this is not required. DO NOT use yaml Octal representation (0o666), this will NOT work.

static A static file to copy into this resource. The file is located on the yaybu path, so can be colocated with your recipes.

template A jinja2 template, used to generate the contents of this resource. The template is located on the yaybu path, so can be colocated with your recipes

template_args The arguments passed to the template.

6.2.2 Directory

A directory on disk. Directories have limited metadata, so this resource is quite limited.

For example:

```
extend resources:
  - Directory:
      name: /var/local/data
      owner: root
      group: root
      mode: 0755
```

The available parameters are:

name The full path to the directory on disk

owner The unix username who should own this directory, by default this is 'root'

group The unix group who should own this directory, by default this is 'root'

mode The octal mode that represents this directory's permissions, by default this is '755'.

parents Create parent directories as needed, using the same ownership and permissions, this is False by default.

6.2.3 Link

A resource representing a symbolic link. The link will be from *name* to *to*. If you specify owner, group and/or mode then these settings will be applied to the link itself, not to the object linked to.

For example:

```
extend resources:
  - Link:
      name: /etc/init.d/exampled
      to: /usr/local/example/sbin/exampled
      owner: root
      group: root
```

The available parameters are:

name The name of the file this resource represents.

owner A unix username or UID who will own created objects. An owner that begins with a digit will be interpreted as a UID, otherwise it will be looked up using the python 'pwd' module.

group A unix group or GID who will own created objects. A group that begins with a digit will be interpreted as a GID, otherwise it will be looked up using the python 'grp' module.

to The pathname to which to link the symlink. Dangling symlinks ARE considered errors in Yaybu.

6.2.4 Execute

Execute a command. This command *is* executed in a shell subprocess.

For example:

```
extend resources:
  - Execute:
      name: core_packages_apt_key
      command: apt-key adv --keyserver keyserver.ubuntu.com --recv-keys {{source.key}}
```

A much more complex example. This shows executing a command if a checkout synchronises:

```
extend resources:
  for bi in flavour.base_images:
    - Execute:
        name: base-image-{{bi}}
        policy:
          apply:
            when: sync
            on: /var/local/checkouts/ci
        command: ./vmbuilder-{{bi}}
        cwd: /var/local/checkouts/ci
        user: root
```

The available parameters are:

name The name of this resource. This should be unique and descriptive, and is used so that resources can reference each other.

command If you wish to run a single command, then this is the command.

commands If you wish to run multiple commands, provide a list

cwd The current working directory in which to execute the command.

environment The environment to provide to the command, for example:

```
extend resources:
  - Execute:
      name: example
      command: echo $FOO
      environment:
        FOO: bar
```

returncode The expected return code from the command, defaulting to 0. If the command does not return this return code then the resource is considered to be in error.

user The user to execute the command as.

group The group to execute the command as.

umask The umask to use when executing this command

unless A command to run to determine if this execute should be actioned

creates The full path to a file that execution of this command creates. This is used like a “touch test” in a Makefile. If this file exists then the execute command will NOT be executed.

touch The full path to a file that yaybu will touch once this command has completed successfully. This is used like a “touch test” in a Makefile. If this file exists then the execute command will NOT be executed.

6.2.5 Checkout

This represents a “working copy” from a Source Code Management system. This could be provided by, for example, Subversion or Git remote repositories.

Note that this is ‘a checkout’, not ‘to checkout’. This represents the resource itself on disk. If you change the details of the working copy (for example changing the branch) the provider will execute appropriate commands (such as `svn switch`) to take the resource to the desired state.

For example:

```
extend resources:
  - Checkout:
      name: /usr/src/myapp
      repository: https://github.com/myusername/myapp
      scm: git
```

The available parameters are:

name The full path to the working copy on disk.

repository The identifier for the repository - this could be an http url for subversion or a git url for git, for example.

branch The name of a branch to check out, if required.

tag The name of a tag to check out, if required.

revision The revision to check out or move to.

scm The source control management system to use, e.g. subversion, git.

scm_username The username for the remote repository

scm_password The password for the remote repository.

user The user to perform actions as, and who will own the resulting files. The default is root.

group The group to perform actions as. The default is to use the primary group of `user`.

mode A mode representation as an octal. This can begin with leading zeros if you like, but this is not required. DO NOT use yaml Octal representation (0o666), this will NOT work.

6.2.6 Package

Represents an operating system package, installed and managed via the OS package management system. For example, to ensure these three packages are installed:

```
extend resources:
  - Package:
      name: apache2
```

The available parameters are:

name The name of the package. This can be a single package or a list can be supplied.

version The version of the package, if only a single package is specified and the appropriate provider supports it (the Apt provider does not support it).

purge When removing a package, whether to purge it or not.

6.2.7 User

A resource representing a UNIX user in the password database. The underlying implementation currently uses the “useradd” and “usermod” commands to implement this resource.

This resource can be used to create, change or delete UNIX users.

For example:

```
extend resources:
  - User:
      name: django
      fullname: Django Software Owner
      home: /var/local/django
      system: true
      disabled-password: true
```

The available parameters are:

name The username this resource represents.

password The encrypted password, as returned by crypt(3). You should make sure this password respects the system’s password policy.

fullname The comment field for the password file - generally used for the user’s full name.

home The full path to the user’s home directory.

uid The user identifier for the user. This must be a non-negative integer.

gid The group identifier for the user. This must be a non-negative integer.

group The primary group for the user, if you wish to specify it by name.

groups A list of supplementary groups that the user should be a member of.

append A boolean that sets how to apply the groups a user is in. If true then yaybu will add the user to groups as needed but will not remove a user from a group. If false then yaybu will replace all groups the user is a member of. Thus if a process outside of yaybu adds you to a group, the next deployment would remove you again.

system A boolean representing whether this user is a system user or not. This only takes effect on creation - a user cannot be changed into a system user once created without deleting and recreating the user.

shell The full path to the shell to use.

disabled_password A boolean for whether the password is locked for this account.

disabled_login A boolean for whether this entire account is locked or not.

6.2.8 Group

A resource representing a unix group stored in the /etc/group file. `groupadd` and `groupmod` are used to actually make modifications.

For example:

```
extend resources:
  - Group:
      name: zope
      system: true
```

The available parameters are:

name The name of the unix group.

gid The group ID associated with the group. If this is not specified one will be chosen.

system Whether or not this is a system group - i.e. the new group id will be taken from the system group id list.

password The password for the group, if required

6.2.9 Service

This represents service startup and shutdown via an init daemon.

The available parameters are:

name A unique name representing an initd service. This would normally match the name as it appears in `/etc/init.d`.

priority Priority of the service within the boot order. This attribute will have no effect when using a dependency or event based init.d subsystem like `upstart` or `systemd`.

start A command that when executed will start the service. If not provided, the provider will use the default service start invocation for the init.d system in use.

stop A command that when executed will start the service. If not provided, the provider will use the default service stop invocation for the init.d system in use.

restart A command that when executed will restart the service. If not provided, the provider will use the default service restart invocation for the init.d system in use. If it is not possible to automatically determine if the restart script is available the service will be stopped and started instead.

reconfig A command that when executed will make the service reload its configuration file.

running A comamnd to execute to determine if a service is running. Should have an exit code of 0 for success.

pidfile Where the service creates its pid file. This can be provided instead of `running` as an alternative way of checking if a service is running or not.

6.3 Dependencies between resources

Resources are always applied in the order they are listed in the `resources` property. You can rely on this to build repeatable and reliable processes. However this might not be enough. There are a couple of other ways to express relationships between resources.

One example is when you want to run a script only if you have deployed a new version of your code:

```
resources:
  - Checkout:
      name: /usr/local/src/mycheckout
      repository: git://github.com/example/example_project

  - Execute:
      name: install-requirements
      command: /var/sites/myapp/bin/pip install -r /usr/local/src/mycheckout/requirements.txt
      policy:
        execute:
          when: sync
          on: Checkout [/usr/local/src/mycheckout]
```

When the `Checkout` step pulls in a change from a repository, the `Execute` resource will apply its `execute` policy.

You can do the same for monitoring file changes too:

```
resources:
- File:
  name: /etc/apache2/security.conf
  static: apache2/security.conf

- Execute:
  name: restart-apache
  commands:
  - apache2ctl configtest
  - apache2ctl graceful
  policy:
    execute:
      when: apply
      on: File[/etc/apache2/security.conf]
```

Sometimes you can't use File (perhaps buildout or maven or similar generates a config file for you), but you still want to trigger a command when a file changes during deployment:

```
resources:
- Execute:
  name: buildout
  command: buildout -c production.cfg
  watches:
  - /var/sites/mybuildout/parts/apache.cfg

- Execute:
  name: restart-apache
  commands:
  - apache2ctl configtest
  - apache2ctl graceful
  policy:
    execute:
      when: watched
      on: File[/var/sites/mybuildout/parts/apache.cfg]
```

This declares that the buildout step might change a File (the `apache.cfg`). Subsequent step can then subscribe to `File[/var/sites/mybuildout/parts/apache.cfg]` as though it was an ordinary file.

All of these examples use a trigger system. When a trigger has been set yaybu will remember it between invocations. Consider the following example:

```
resources:
- File:
  name: /etc/apache2/sites-enabled/mydemosite

- Directory:
  name: /var/local/tmp/this/paths/parent/dont/exist

- Execute:
  name: restart-apache2
  command: /etc/init.d/apache2 restart
  policy:
    execute:
      when: apply
      on: File[/etc/apache2/sites-enabled/mydemosite]
```

When it is run it will create a file in the `/etc/apache2/sites-enabled` folder. Yaybu knows that the `Execute[restart-apache2]` step must be run later. It will record a trigger for the `Execute` statement in `/var/run/yaybu/`. If the `Directory[]` step fails and yaybu terminates then the next time yaybu is execute it

will instruct you to use the `--resume` or `--no-resume` command line option. If you `--resume` it will remember that it needs to restart `apache2`. If you choose `--no-resume` it will not remember, and `apache` will not be restarted.

6.4 Examples

6.4.1 Deploy to an existing server or VM

To deploy to your current computer by SSH you can use a Yaybufile like this:

```
new Provisioner as provisioner:
```

```
resources:
  - File:
      name: /some_empty_file

  - Execute:
      name: hello_world
      command: touch /hello_world
      creates: /hello_world
```

```
server:
  fqdn: localhost
  username: root
  password: penguin55
  private_key: path/to/key
```


MANAGING CLOUD LOAD BALANCERS

Yaybu can manage your load balancers using a `LoadBalancer` part. They run as soon as all of the inputs become valid, as opposed to when the program encounters them.

A basic setup looks like this:

```
new LoadBalancer as lb:
  name: myprojectlb

  driver:
    id: ELB
    key: yourawskey
    secret: yourawssecret

  # Listen on port 80 for http access
  port: 80
  protocol: http
  algorithm: round-robin

  members:
    - {{ server1 }}
```

7.1 Options

You must specify a name when creating a `LoadBalancer` part. Some backends will use this as a unique id for the load balancer. Take care to avoid duplicating load balancer names in different configurations!

The `driver` section contains the settings used by libcloud to initialize a driver. This typically includes account information - a access key and secret, a username and password, or similar.

You must specify a `port` for the load balancer to listen on.

The load balancer needs to know what `protocol` it is balancing. For example, if it is handling SSL connections it can act as an SSL terminator but to do this it needs to know it is an SSL protocol. Not all balancers support all protocols, and Yaybu doesn't expose SSL support at the moment. You can set `protocol` to one of:

- http
- https
- tcp
- ssl

Some load balancers let you choose an `algorithm`. This is the method by which the load balancer distributes traffic. It can be one of:

random Incoming connections are assigned to a backend at random

round-robin Incoming connections are passed to a backend in a circular fashion without any considering of priority.

least-connections Incoming connections are passed to the backend with the least number of active connections with the assumption that it must have the most free capacity.

weighted-round-robin Same as `round-robin`, but also factors in a weight factor for each member

weighted-least-connections Same as `least-connections`, but also factors in a weight factor for each member

The `members` input is a list of all compute resources that load will be spread over. There are a few variations here.

If you are doing load balancing for port 80 and forwarding to port 80 on the backend VM's then you can:

```
new LoadBalancer as lb:
  <snip>
  members:
    - {{ server1 }}
    - {{ server2 }}
```

In this example `server1` and `server2` are `Compute` parts defined elsewhere in your configuration.

However if you are using different ports on the backend servers you can:

```
new LoadBalancer as lb:
  <snip>
  members:
    - instance: {{ server1 }}
      port: 8080
```

Not all backends support this, and an error will be raised before deployment starts if it is not.

There are 2 main types of cloud load balancer. The first accepts IP addresses and ports. If you pass a `Compute` node to this type of load balancer Yaybu will determine it's IP automatically. But you can pass ip addresses manually:

```
new LoadBalancer as lb:
  <snip>
  members:
    - ip: 192.168.0.1
      port: 8080
```

Other load balancers expect to be give a list of compute instance ids. Again, Yaybu will do the right thing if given `Compute` parts. But you can also give it `id` values directly:

```
new LoadBalancer as lb:
  <snip>
  members:
    - id: ec2123ab
      port: 8080
```

7.2 Outputs

The part exposes a number of output variables to other Yaybu parts.

Each load balancer that is created has a unique `id`. In some cases this may be the same as the `name`.

A load balancer has a `public_ip`. This is the public facing method of accessing the load balancer.

7.3 Supported services

Using libcloud to implement this part allows us to support a number of DNS services. Some of these receive more extensive real world testing than others and are listed in this section.

7.3.1 Elastic Load Balancing

The driver id for Elastic Load Balancing is ELB:

```
new LoadBalancer as lb:
  name: my-load-balancer

  driver:
    id: ELB
    key: myaccesskey
    secret: myaccesssecret
    region: eu-west-1

  port: 80
  protocol: http
  algorithm: round-robin

  # The default is just a
  ex_memebers_availability_zones:
    - a
    - b

  members:
    - id: ec2123
```

For this driver:

- After creating a balancer you cannot change its settings (you can continue to add and remove members).
- `protocol` must be either `tcp` or `http`.
- `algorithm` must be ?.....?
- `members` are managed by instance id. You cannot set the backend port.
- `ex_members_availability_zones` is an ELB specific extension that controls which Amazon availability zones a balancer is in.

7.4 Community supported services

By using libcloud to support the services in the previous section, the following services are also available:

7.4.1 Brightbox

The driver id for brightbox is BRIGHTBOX:

```
new LoadBalancer as lb:
  name: my-load-balancer

  driver:
    id: BRIGHTBOX
    key: acc-43ks4
    secret: mybrightboxsecret

  port: 80
  protocol: http
  algorithm: round-robin

  members:
    - id: ec2123
```

For the Brightbox loadbalancer:

- `protocol` must be `http` or `tcp`
- `algorithm` must be `round-robin` or `least-connections`
- `members` are managed by instance id, and you cannot set the backend port (your backends must listen on the same port as your load balancer).

7.4.2 Cloudstack

The driver id for cloudstack is not currently set upstream, so it is currently unavailable.

For the CloudStack loadbalancer:

- After creating a balancer you cannot change its setting (you can continue to add and remove members).
- `protocol` must be `tcp`
- `algorithm` must be `round-robin` or `least-connections`
- `members` are managed by instance id. You cannot set the backend port.

7.4.3 GoGrid

The driver id for GoGrid is GOGRID:

```
new LoadBalancer as lb:
  name: my-load-balancer

  driver:
    id: GOGRID
    key: myaccesskey
    secret: myaccesssecret

  port: 80
  protocol: http
  algorithm: round-robin

  members:
    - id: ec2123
```

For this driver:

- `protocol` must be `http`

- `algorithm` must be `round-robin` or `least-connections`
- `members` are managed by `ip`. Each backend can use a different port.

7.4.4 Ninfold

The driver id for Ninfold is `NINEFOLD`:

```
new LoadBalancer as lb:
  name: my-load-balancer

  driver:
    id: NINEFOLD
    key: myaccesskey
    secret: myaccesssecret

  port: 80
  protocol: http
  algorithm: round-robin

  members:
    - id: ec2123
```

Ninfold uses CloudStack, so see that section for additional notes.

7.4.5 Rackspace

The driver id for Rackspace load balancing is `RACKSPACE_UK`:

```
new LoadBalancer as lb:
  name: my-load-balancer

  driver:
    id: RACKSPACE_UK
    key: myaccesskey
    secret: myaccesssecret

  port: 80
  protocol: http
  algorithm: round-robin

  members:
    - id: ec2123
```

For this driver:

- After creating a balancer you can later change its settings.
- The list of supported `protocol` options is dynamic and fetched from Rackspace at runtime.
- `algorithm` must be one of `random`, `round-robin`, `least-connections`, `weighted-round-robin` or `weighted-least-connections`.
- `members` are managed by `ip/port` pairs.

MANAGING CLOUD BASED DNS

Yaybu can manage your DNS using a `Zone` part. A basic setup looks like this:

```
new Zone as mydns:
  driver:
    id: GANDI
    key: yourgandikey

  domain: example.com

  records:
    - name: mail
      data: 173.194.41.86
      type: A

    - name: www
      data: www.example.org
      type: CNAME
```

In this example, when you run `yaybu apply` this part will look for a zone named `example.com` and create it if it does not exist. It will ensure that all the `records` given exist and are of the right `type` and have the right `data`.

8.1 Options

Use the `driver` argument to find and initialize a libcloud DNS driver. You must specify an `id` so that the right service is targeted. Other variables include `users` and `secrets` and are described in the service-specific notes below.

You must specify a `domain`. If a zone for this domain doesn't exist it will be created.

You must provide a list of DNS `records` to publish in the zone. At the very least you will specify a `name` and `data` but other options are available:

name For example `www` or `pop`. You do not need to specify a fully qualified domain name.

type The type of DNS record - for example `A` or `CNAME`.

data The data to put in the DNS record. This varies between record types, but is typically an IP address for `A` records or a fully qualified domain name for a `CNAME` record.

ttl How long this record can be cached for, specified in seconds. Specifying `86400` seconds would mean that if a DNS record was changed some DNS servers could be returning the old value for up to 24 hours.

8.2 Supported services

Using `libcloud` to implement this part allows us to support a number of DNS services. Some of these receive more extensive real world testing than others and are listed in this section.

8.2.1 Gandi

The driver id for `Gandi` is `GANDI`:

```
new Zone as dns:
  driver:
    id: GANDI
    key: yourgandikey

  domain: example.com

  records:
    - name: www
      data: 192.168.0.1
```

TTL can only be set on records.

Gandi supports the following record types:

- NS
- MX
- A
- AAAA
- CNAME
- TXT
- SRV
- SPF
- WKS
- LOC

8.2.2 Route53

The driver id for `Route53` is `ROUTE53`:

```
new Zone as dns:
  domain: example.com

  driver:
    id: ROUTE53
    key: youraccountkey
    secret: youraccountsecret

  records:
    - name: www
      data: 192.168.0.1
```

TTL can only be set on records.

Route53 supports the following record types:

- NS
- MX
- A
- AAAA
- CNAME
- TXT
- SRV
- PTR
- SOA
- SPF
- TXT

8.3 Community supported services

By using *libcloud* to support the services in the previous section, the following services are also available:

8.3.1 HostVirtual

The driver id for `HostVirtual` is `HOSTVIRTUAL`:

```
new Zone as dns:
  domain: example.com

  driver:
    id: HOSTVIRTUAL
    key: yourkey
    secret: yoursecret

  records:
    - name: www
      data: 192.168.0.1
```

TTL can be set by zone and by record.

HostVirtual supports the following record types:

- A
- AAAA
- CNAME
- MX
- TXT
- NS
- SRV

8.3.2 Linode

The driver id for [Linode](#) is `LINODE`:

```
new Zone as dns:
  domain: example.com

  driver:
    id: LINODE
    key: yourlinodeikey
    secret: yourlinodesecret

  records:
    - name: www
      data: 192.168.0.1
```

TTL can be set by zone and by record.

Linode supports the following record types:

- NS
- MX
- A
- AAAA
- CNAME
- TXT
- SRV

8.3.3 RackSpace

The driver id for [Rackspace DNS](#) is `RACKSPACE_UK` or `RACKSPACE_US`:

```
new Zone as dns:
  domain: example.com

  driver:
    id: RACKSPACE_UK
    user_id: rackspace_user_id
    key: rackspace_secret_key

  records:
    - name: www
      data: 192.168.0.1
```

TTL can be set by zone and by record.

Rackspace supports the following record types:

- A
- AAAA
- CNAME
- MX
- NS

- TXT
- SRV

8.3.4 Zerigo

The driver id for *Zerigo* is ZERIGO:

```
new Zone as dns:
  domain: example.com

  driver:
    id: ZERIGO
    key: youraccountkey
    secret: youraccountsecret

  records:
    - name: www
      data: 192.168.0.1
```

TTL can be set by zone and by record.

Zerigo supports The following record types:

- A
- AAAA
- CNAME
- MX
- REDIRECT
- TXT
- SRV
- NAPTR
- NS
- PTR
- SPF
- GEO
- URL

SYNCING STATIC FILES TO CLOUD SERVICES

The `StaticContainer` part allows static assets to be synchronised from one container to another. The primary use case is to upload assets from your local drive to the cloud.

A simple invocation looks like this:

```
new StaticContainer as my_static_files:
  source: local/path

  destination:
    id: S3
    key: yourawskey
    secret: yourawssecret
    container: target_container
```

This will sync the contents of a local folder to a destination container.

If the source and destination have incompatible approaches to hashing `StaticContainer` will automatically generate and store a manifest in the target destination.

Any service that can be used as a destination can also be used as a source, so this also works:

```
new StaticContainer as my_static_files:
  source:
    id: S3
    key: yourawskey
    secret: yourawssecret
    container: source_container

  destination:
    id: S3
    key: yourawskey
    secret: yourawssecret
    container: target_container
```

9.1 Options

There are 2 main options for `StaticContainer`. The source and destination.

`source` can either be a simple string with a path to local files or it can describe a libcloud driver:

```
source:
  id: S3
  key: yourawskey
  secret: yourawssecret
  container: source_container
```

The destination must be a set of driver parameters as above.

The exact options vary based on the driver that you use, and this is covered in more detail below.

9.2 Supported drivers

Using libcloud to implement this part allows us to support a number of DNS services. Some of these receive more extensive real world testing than others and are listed in this section.

9.2.1 Local files

You can synchronise from and to any folder that is accessible locally use the LOCAL driver:

```
new StaticContainer as my_static_files:
  source: ~/source

  destination:
    id: LOCAL
    key: yourawskey
    secret: yourawssecret
    container: target_container
```

9.2.2 S3

The driver id for S3 is S3:

```
new StaticContainer as my_static_files:
  source: ~/source

  destination:
    id: S3
    key: yourawskey
    secret: yourawssecret
    container: target_container
```

9.3 Community supported drivers

By using libcloud to support the services in the previous section, the following services are also available:

9.3.1 Azure Blobs

9.3.2 CloudFiles

9.3.3 Google Storage

9.3.4 Nimbus

9.3.5 Ninefold

COMBINING PARTS

You can combine the parts in different ways using the `yay` language.

10.1 Create and provision a cloud server

You can use a *Compute* part to provide the `server` key of the *Provisioner* part:

```
new Provisioner as vml:
  new Compute as server:
    name: mytestvml
    driver:
      id: VMWARE
    image:
      id: /home/john/vmware/ubuntu/ubuntu.vmx
    user: ubuntu

  resources:
    - Package:
      name: git-core
```

When the *Provisioner* part tries to access `server.fqdn` the *Compute* part will automatically find an existing `mytestvml` or create a new one if needed.

10.2 Create a new instance and automatically set up DNS

You can use the IP from a *Compute* part in other parts just by using it like any other variable:

```
new Compute as server:
  name: mytestserver
  driver:
    id: EC2
    key: secretkey
    secret: secretsecret
  image: imageid
  size: t1.micro

new Zone as dns:
  driver:
    id: ROUTE53
    key: secretkey
    secret: secret
```

```
domain: mydomain.com
records:
  - name: www
    type: A
    data: {{ server.public_ip }}
```

10.3 Create and provision interdependent cloud servers

You can refer to server A from the configuration for server B and vice versa and Yaybu will satisfy the dependencies automatically:

```
new Provisioner as vm1:
  new Compute as server:
    name: mytestvm1
    driver:
      id: VMWARE
    image:
      id: /home/john/vmware/ubuntu/ubuntu.vmx
    user: ubuntu

  resources:
    - File:
      name: /etc/foo
      template: sometemplate.j2
      template_args:
        vm2_ip: {{ vm2.server.public_ips[0] }}

new Provisioner as vm2:
  new Compute as server:
    name: mytestvm2
    driver:
      id: VMWARE
    image:
      id: /home/john/vmware/ubuntu/ubuntu.vmx
    user: ubuntu

  resources:
    - File:
      name: /etc/foo
      template: sometemplate.j2
      template_args:
        vm1_ip: {{ vm1.server.public_ips[0] }}
```

here a templated File on mytestvm1 needs the IP address of mytestvm2. mytestvm2 needs the IP address of mytestvm1. Yaybu is able to work out that it should activate both *Compute* parts first, then proceed to provision both template files to the instances.

LANGUAGE TOUR

Yay is a non-strict language that supports lazy evaluation. It is a sort of mutant child of YAML and Python, with some of the features of both.

There are some significant differences from YAML and this absolutely does not attempt to implement the more esoteric parts of YAML.

A particularly significant restriction is that keys may not contain whitespace. keys in a configuration language are expected to be simple bare terms. This also helpfully keeps the magic smoke firmly inside our parser.

It is important to understand that for any line of input it is imperative “pythonish” or declarative “yamlish”. It actually works well and we find it very easy to read, for example:

```
a: b
if a == 'b':
    c: d
```

It is pretty clear that some of those lines are declarative and some are imperative. When in pythonish mode it works just as you would expect from python, when in yamlish mode it works as a declarative language for defining terms.

11.1 Mappings

A mapping is a set of key value pairs. They key is a string and the value can be any type supported by Yay. All Yay files will contain at least one mapping:

```
site-domain: www.yaybu.com
number-of-zopes: 12
in-production: true
```

You can nest them as well, as deep as you need to. Like in Python, the relationships between each item is based on the amount of indentation:

```
interfaces:
  eth0:
    interfaces: 192.168.0.1
    dhcp: yes
```

11.2 List

You can create a list of things by creating an intended bulleted list:

```
packages:
- python-yay
- python-yaybu
- python-libvirt
```

If you need to express an empty list you can also do:

```
packages: []
```

11.3 Variable Expansion

If you were to specify the same Yaybu recipe over and over again you would be able to pull out a lot of duplication. You can create templates with placeholders in and avoid that. Lets say you were deploying into a directory based on a customer project id:

```
projectcode: MyCustomer-145

resources:
- Directory:
  name: /var/local/sites/{{projectcode}}

- Checkout:
  name: /var/local/sites/{{projectcode}}/src
  repository: svn://mysvnserver/{{projectcode}}
```

If you variables are in mappings you can access them using `.` as seperator. You can also access specific items in lists with `[]`:

```
projects:
- name: www.foo.com
  projectcode: Foo-1
  checkout:
    repository: http://github.com/isotoma/foo
    branch: master

resources:
- Checkout:
  repository: /var/local/sites/{{projects[0].checkout.repository}}
```

Sometimes you might only want to optionally set variables in your configuration. Here we pickup `project.id` if its set, but fall back to `project.name`:

```
project:
  name: www.baz.com

example_key: {{project.id else project.name}}
```

11.4 Including Files

You can import a recipe using the `yay extends` feature. If you had a template `foo.yay`:

```
resources:
- Directory:
  name: /var/local/sites/{{projectcode}}
```

```
- Checkout:
  name: /var/local/sites/{{projectcode}}/src
  repository: svn://mysvnserver/{{projectcode}}
```

You can reuse this recipe in `bar.yay` like so:

```
include "foo.yay"

include foo.bar.includes

projectcode: MyCustomer-145
```

11.5 Search paths

You can add a directory to the search path:

```
search "/var/yay/includes"

search foo.bar.searchpath
```

11.6 Configuration

```
::
```

configure openers:

foo: bar baz: quux

configure basicauth: zip: zop

11.7 Ephemeral keys

These will not appear in the output:

```
for a in b
  set c = d.foo.bar.baz
  set d = dsds.sdsd.sewewe
  set e = as.ew.qw
foo: c
```

11.8 Extending Lists

If you were to specify resources twice in the same file, or indeed across multiple files, the most recently specified one would win:

```
resources:
  - foo
  - bar
```

```
resources:
  - baz
```

If you were to do this, resources would only contain baz. Yay has a function to allow appending to predefined lists: `append`:

```
resources:
  - foo
  - bar

extend resources:
  - baz
```

11.9 Conditions

```
foo:
  if averylongvariablename == anotherverylongvariablename and \
     yetanothervariable == d and e == f:

    bar:
      quux:
        foo:
          bar: baz

  elif blah == something:
    moo: mah

  else:
    - baz
```

11.10 For Loops

You might want to have a list of project codes and then define multiple resources for each item in that list. You would do something like this:

```
projectcodes:
  MyCustomer-100
  MyCustomer-72

extend resources:

  for p in projectcodes:
    - Directory:
      name: /var/local/sites/{{p}}

    for q in p.qcodes:
      - Checkout:
        name: /var/local/sites/{{p}}/src
        repository: svn://mysvnserver/{{q}}
```

You can also have conditions:

```
fruit:
  - name: apple
    price: 5
  - name: lime
    price: 10
```

```
cheap:
  for f in fruit if f.price < 10:
    - {{f}}
```

You might need to loop over a list within a list:

```
staff:
  - name: Joe
    devices:
      - macbook
      - iphone

  - name: John
    devices:
      - air
      - iphone

stuff:
  for s in staff:
    for d in s.devices:
      {{d}}
```

This will produce a single list that is equivalent to:

```
stuff:
  - macbook
  - iphone
  - air
  - iphone
```

You can use a for against a mapping too - you will iterate over its keys. A for over a mapping with a condition might look like this:

```
fruit:
  # recognised as decimal integers since they look a bit like them
  apple: 5
  lime: 10
  strawberry: 1

cheap:
  for f in fruit:
    if fruit[f] < 10:
      {{f}}
```

That would return a list with apple and strawberry in it. The list will be sorted alphabetically: mappings are generally unordered but we want the iteration order to be stable.

11.11 Select

The select statement is a way to have conditions in your configuration.

Lets say `host.distro` contains your Ubuntu version and you want to install difference packages based on the distro. You could do something like:

```
packages:
  select distro:
    karmic:
```

```
    - python-setuptools
lucid:
    - python-distribute
    - python-zc.buildout
```

11.12 Function calls

Any sandboxed python function can be called where an expression would exist in a yay statement:

```
set foo = sum(a)
for x in range(foo):
    - x
```

11.13 Class bindings

Classes can be constructed on-the-fly:

```
parts:
  web:
    new Compute:
      foo: bar
      % for x in range(4)
        baz: x
```

Classes may have special side-effects, or provide additional data, at runtime.

Each name for a class will be looked up in a registry for a concrete implementation that is implemented in python.

11.14 Macros

Macros provided parameterised blocks that can be reused, rather like a function.

you can define a macro with:

```
macro mymacro:
  foo: bar
  baz: {{thing}}
```

You can then call it later:

```
foo:
  for q in x:
    call mymacro:
      thing: {{q}}
```

11.15 Prototypes

Prototypes contain a default mapping which you can then override. You can think of a prototype as a class that you can then extend.

In their final form, they behave exactly like mappings:

```

prototype DjangoSite:
    set self = here

    name: www.example.com

    sitedir: /var/local/sites/{{ self.name }}
    rundir: /var/run/{{ self.name }}
    tmpdir: /var/tmp/{{ self.name }}

    resources:
        - Directory:
            name: {{ self.tmpdir }}

        - Checkout:
            name: {{ self.sitedir}}
            source: git://github.com/

some_key:
    new DjangoSite:
        name: www.mysite.com

```

11.16 Here

Here is a reserved word that expands to the nearest parent node that is a mapping.

You can use it to refer to siblings:

```

some_data:
    sitename: www.example.com
    sitedir: /var/www/{{ here.sitename }}

```

You can use it with `set` to refer to specific points of the graph:

```

some_data:
    set self = here

    nested:
        something: goodbye
        mapping: {{ self.something }}           # Should be 'hello'
        other_mapping: {{ here.something }}    # Should be 'goodbye'

    something: hello

```


PROTECTING YOUR SECRETS, KEYS AND CERTIFICATES

Yaybu natively supports the use of GPG as a way to protect both secret variables in your configuration files and the use of encrypted assets when using the *Provisioner* part.

12.1 Installing GPG

On an Ubuntu machine GPG can be installed with:

```
sudo apt-get install gnupg
```

On OSX you can install a pre-built binary produced by the [GPGTools](#) team, or you can install it using [brew](#):

```
brew install gnupg
```

12.2 Creating a GPG key

If you want to encrypt your secrets for multiple recipients you will need a GPG key. We tend to follow the advice of Debian when [creating new keys](#) and as such:

- You should go for a 4096 bit key
- You should avoid SHA1 as your preferred hash

You can generate a signing and encryption key has follows:

```
paul@jolt:~$ gpg --gen-key
gpg (GnuPG) 1.4.10; Copyright (C) 2008 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

gpg: directory `/home/paul/.gnupg' created
gpg: new configuration file `/home/paul/.gnupg/gpg.conf' created
gpg: WARNING: options in `/home/paul/.gnupg/gpg.conf' are not yet active during this run
gpg: keyring `/home/paul/.gnupg/secring.gpg' created
gpg: keyring `/home/paul/.gnupg/pubring.gpg' created
Please select what kind of key you want:
  (1) RSA and RSA (default)
  (2) DSA and Elgamal
  (3) DSA (sign only)
```

```
(4) RSA (sign only)
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) 4096
Requested keysize is 4096 bits
Please specify how long the key should be valid.
    0 = key does not expire
    <n> = key expires in n days
    <n>w = key expires in n weeks
    <n>m = key expires in n months
    <n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y
```

You need a user ID to identify your key; the software constructs the user ID from the Real Name, Comment and E-mail Address in this form:

```
"Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"
```

```
Real name: Paul Ubbot
E-mail address: pubbot@example.com
Comment:
You selected this USER-ID:
    "Paul Ubbot <pubbot@example.com>"
```

Change (N)ame, (C)omment, (E)-mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.

We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, use the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy! (Need 284 more bytes)

```
+++++
.....+++++
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, use the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
```

```
.....+++++
.....+++++
gpg: /home/paul/.gnupg/trustdb.gpg: trustdb created
gpg: key D770E8A9 marked as ultimately trusted
public and secret key created and signed.
```

```
gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u
pub 4096R/D770E8A9 2013-08-28
    Key fingerprint = 746B 2477 FB6F CCC6 46C2 D5D2 288C EF6D D770 E8A9
uid                               Paul Ubbot <pubbot@example.com>
sub 4096R/49BEE9E3 2013-08-28
```

You now have a GPG key.

Ideally you should sign the keys of the people you are working with to build a web of trust, however there is no requirement to do so. There are excellent resources online for holding a [key signing event](#).

In order to encrypt for you collaborators will need a copy of the public portion of your key. You can publish your key like so:

```
gpg --keyserver pgp.mit.edu --send-key D770E8A9
```

Anyone can retrieve your public key like so:

```
gpg --keyserver pgp.mit.edu --recv-keys D770E8A9
```

12.3 Encrypting your configuration

You might have a `secrets.yay` that looks like this:

```
secrets:
  aws: somepassword
  rackspace: abetterpassw0rd
```

You can encrypt it for your new key like this:

```
gpg -e -r D770E8A9 secrets.yay
```

You can use e-mail addresses as well:

```
gpg -e -r pubbot@example.com secrets.yay
```

In both cases a `secrets.yay.gpg` will be generated, which you can then reference from your Yaybufile:

```
include "secrets.yay.gpg"

new Compute as myserver:
  driver:
    id: EC2
    key: myawskey
    secret: {{ secrets.aws }}
  <snip>
```

12.4 Encrypting your provisioner assets

The *Provisioner* part is GPG aware. If you were copying a file to a server that was a secret you could encrypt it as above and then refer to it from `File` parts:

```
new Provisioner as p:
  resources:
    - File:
      name: /etc/defaults/foobar
      static: foobar.gpg
```

In this situation Yaybu would notify you when it changed the file, but it wouldn't show a diff as it knows the file is encrypted and so secret.

12.5 Integration with VIM

We are big fans of the `vim-gnupg` plugin which allows you to:

```
vi secrets.yay.gpg
```

It will transparently decrypt the file, allow you to edit the text contents, then when you save it will re-encrypt it. It will preserve the same recipients, which is very useful if you are working with a team.

CHANGE SOURCES

13.1 EXPERIMENTAL: Provisioning on commit (via Travis CI)

Travis CI has a mechanism to encrypt secrets. It also has a hook that is run on success. This means we can have yaybu perform system orchestration tasks on commit + successful CI run without having to run any of our own servers.

Here is a simple Yaybufile:

```
yaybu:
  options:
    - name: BIGV_KEY
    - name: BIGV_SECRET
    - name: BIGV_ACCOUNT
    - name: BIGV_ROOT_PASSWORD
      default: penguin55

new Provisioner as myexample:
  new Compute as server:
    driver:
      id: BIGV
      key: {{ yaybu.argv.BIGV_KEY }}
      secret: {{ yaybu.argv.BIGV_SECRET }}

    image: precise

    name: myexample

    user: root
    password: {{ yaybu.argv.BIGV_ROOT_PASSWORD }}

  resources:
    - Package:
      name: git-core

    - Checkout:
      name: /tmp/yaybu
      scm: git
      repository: https://github.com/yaybu/example
```

The `yaybu.options` section allows us to define arguments that can be passed to yaybu via the command line. You can define defaults to use if no such argument is passed in.

Now we can encrypt these details using the travis command line tool:

```
travis encrypt BIGV_KEY=myusername --add env.global
travis encrypt BIGV_SECRET=password --add env.global
travis encrypt BIGV_ACCOUNT=myaccount --add env.global
travis encrypt BIGV_ROOT_PASSWORD=password --add env.global
```

And here is what your `.travis.yml` looks like:

```
language: python
python:
  - "2.6"

env:
  global:
    - secure: <YOUR_ENCRYPTED_STRINGS>

script:
  - true # This is where you would normally run your tests

after_success:
  - sudo add-apt-repository yaybu-team/yaybu
  - sudo apt-get update
  - sudo apt-get install python-yaybu
  - yaybu up BIGV_KEY=$BIGV_KEY BIGV_SECRET=$BIGV_SECRET BIGV_ACCOUNT=$BIGV_ACCOUNT BIGV_ROOT_PASSWORD=$BIGV_ROOT_PASSWORD
```

13.2 EXPERIMENTAL: Provisioning on commit

This uses a new command, `yaybu run`. This puts yaybu into a mode where it continues to run, rather than deploying then exiting. Parts can set up listeners to respond to external events like commits or monitoring systems.

To deploy on commit you can use a Yaybufile like this:

```
new GitChangeSource as changesource:
  polling-interval: 10
  repository: https://github.com/isotoma/yaybu

new Provisioner as myexample:
  new Compute as server:
    driver:
      id: EC2_EU_WEST
      key: mykey
      secret: mysecret

    size: t1.micro
    image: ami-4f504f3b

    ex_keyname: mysshkey
    name: myexample

    user: ubuntu
    private_key: mysshkey.pem

  resources:
    - Package:
      name: git-core

    - Checkout:
      name: /tmp/yaybu
```

```
scm: git
repository: {{ changesource.repository }}
revision: {{ changesource.master }}
```

The `GitChangeSource` part polls and sets `{{changesource.master}}` with the SHA of the current commit.

This example `changesource` polls to learn if a new commit has occurred. This is only because the part is an example implementation - it could easily be a webhook or zeromq push event.

The `Checkout` resource uses the `master` property of `changesource`. Yaybu can use this dependency information to know that the `Provisioner` that owns the `Checkout` is stale and needs applying every time `master` changes.

If your `Yaybufile` contained another `Provisioner` that didn't have such a `Checkout` (perhaps its the database server) then Yaybu would equally know *not* to deploy to it on commit.

HACKING ON YAYBU

If you are going to hack on Yaybu please stop by IRC and say hi! We are on OFTC in #yaybu.

The source code is available on GitHub - please fork it and send us pull requests!

The main components you might want to hack on are:

`yaybu` The main app. You'll need to change this to add new CLI subcommands or add new `Parts`.
`yay` The configuration language runtime. You will need to change this to improve parsing, the runtime graph, file transports, etc.
`yaybu.app` This contains a small OSX application and build scripts to package Yaybu for OSX. You will probably need to fork this to fix OSX specific bugs.

To get a development environment with required dependencies:

```
virtualenv .  
./bin/pip install -r requirements.txt
```

NOTE: Currently the testrunner will try and run a set of integration tests against an ubuntu chroot. These tests are only run on ubuntu systems with the following packages installed:

```
sudo apt-get install fakechroot fakeroot debootstrap cowdancer
```

To run the test:

```
./bin/nose2
```

Then write a configuration file called `Yaybufile`:

And run it with:

```
./bin/yaybu up
```


INDICES AND TABLES

- *genindex*
- *modindex*
- *search*