Modest .INI administration

Release 0.1.1

Priam Kanealii

September 2012

CONTENTS

CHAPTER

ONE

WELCOME

Minimin is a handful of administrative conventions expressed in INI-style configuration files and implemented using Fabric. Its goal is to help keep administration light, modular, and easy to reason about. Fabric executes remote tasks via SSH, and general system configuration is managed by Puppet.

Minimin is not a configuration management system - it is a way to think about, and ease the pain of, incorporating management components into an existing environment.

1.1 Audience

If you:

- juggle personal and professional projects in exclusive environments and want to reduce the expense of context switching,
- find your occasional-but-valuable commands getting lost among personal notes, project documentation, todo-lists and bookmarks,
- wish it were easier to share your workflow (or as much of it as you're willing) with others,
- use (or want to make more use of) SSH as the default pipeline to remote servers but aren't motivated to manage keys or an `ssh_config`,
- don't want to mess with Puppet's SSL client-server "pull-based" system, or don't have a happy recipe to bootstrap modules..

..then you may find something helpful here.

1.2 Admin environment

Three conceptual domains comprise an administrator's environment:

1. **user**: Everything specific to an administrator's user account; files that needn't or shouldn't be shared with team members or the public. For example, exactly *where* an admin keeps off-line project repositories probably isn't important to anyone else and private SSH keys should definitely be kept private.
2. **project**: Specific sets of responsibilities, methods, and tasks define projects. Project information is shared between team members and may be available to others for the purposes of monitoring progress, making backups, or education. Projects should use some form of (preferably distributed) version control.
3. **support**: Files that are shared by several projects or available publicly constitute support. Support includes everything in this module, additional pre-configured commands that you share with friends, third-party software packages, libraries, etc.. Support files should be easily accessible, preferably in a local cache.

1.3 Layout example

In the following environment, an `admin` user keeps private files in `/home/admin`, two project repositories in `/proj/websvc` and `/proj/common`, and third party support in `/support`. Project repositories contain Minimin configurations in their respective `etc/*.ini` files, and Puppet modules are found in the `common` project and as part of the support from a `vendor`. Support packages fall into one of three categories: sources, RPMs, and Solaris pacakges. All these locations are configured in `/home/admin/.fabricrc`:

```
/
+-- home
|   +-- admin
|       +-- .fabricrc                       (1)
|       +-- .ssh                            (6)
|           +-- config                      (7)
|           +-- id_admin_root_websvr
|           +-- id_admin_root_websvr.pub
+-- proj
|   +-- websvc
|   |   +-- etc                             (2)
|   |       +-- site.ini                    (3)
|   +-- common
|       +-- etc                             (2)
|       |   +-- share.ini                   (3)
|       +-- puppet
|           +-- modules                     (4)
|               +-- ...
+-- support
    +-- source                              (5)
    |   +-- tmux-1.6.tar.gz
    +-- rpm                                 (5)
    |   +-- zeromq-2.1.9-1.fc15.x86_64.rpm
    +-- pkg                                 (5)
    |   +-- nc-110-sol10-x86-local.gz
    +-- vendor
        +-- puppet
            +-- modules                     (4)
                +-- ...
```

The admin's Fabric config (1, `/home/admin/.fabricrc`) tells us where to find administrative components in our directory tree. It uses `option = value` entries to define:

- Minimin config file inclusion paths (2):

  ```
  admin_cfg_path = /proj/websvc/etc /proj/common/etc
  ```

- Minimin config files to include (3):

  ```
  admin_cfg_include = site.ini share.ini
  ```

- Puppet module homes (4):

  ```
  admin_puppet_path = /proj/common/puppet/modules \
                      /support/vendor/puppet/modules
  ```

- paths to find software packages (5):

  ```
  admin_support_path = /support/source /support/rpm /support/pkg
  ```

- a home for our SSH keys (6):

```
admin_ssh_home = /home/admin/.ssh
```

- and an SSH host configuration (7):

```
use_ssh_config = True
ssh_config_path = /home/admin/.ssh/config
```

1.4 INI configs

The Minimin configuration - the result of all the `*.ini` files above - tells us how to apply these components to our target nodes (hosts). Our node names map to an OS/platform identifier to determine the relevancy of configured commands (in the `[exe]` section) and "manual" installs (in the `[install]` section). Roles group nodes for the convenience of executing ad-hoc commands in bulk and applying a common Puppet configuration:

```
[node]
websvr = f15
mailsvr = s10

[role]
common.nodes = websvr mailsvr
common.puppet.mods = master qnd
common.puppet.modpath = /etc/puppet/modules
common.puppet.site = /etc/puppet/modules/master/manifests/site.pp

web.nodes = websvr

mail.nodes = mailsvr

[exe]
pkg-grep.txt = look for a package name matching {str}
pkg-grep.f15 = rpm -qa | grep {str}
pkg-grep.s10 = pkginfo | grep -i {str}

[install]
url-sfw10 = ftp://ftp.sunfreeware.com/pub/freeware/intel/10

nc.txt.s10 = Netcat, the all-purpose network tool SMCnc
nc.url.s10 = %(url-sfw10)s/nc-110-sol10-x86-local.gz
nc.src.s10 = nc-110-sol10-x86-local.gz
nc.dst.s10 = /tmp/nc-110-sol10-x86-local.gz
nc.pth.s10 = /usr/sbin:/usr/bin
nc.req.s10 = which pkgadd pkgrm gunzip
nc.chk.s10 = /usr/local/bin/nc -h 2>&1 | grep 'v1.10'
nc.del.s10 = pkgrm SMCnc
nc.exe.s10 = #!/bin/sh
    cd /tmp && \
    gunzip nc-110-sol10-x86-local.gz && \
    pkgadd -d nc-110-sol10-x86-local all
```

Two nodes, `websvr` and `mailsvr`, run Fedora 15 (`f15`) and Solaris 10 (`s10`), respectively. The platform identifiers are arbitrary strings.

The `common` role includes both nodes. The `common` role requires two Puppet modules, `master` and `qnd`, both of which should be found in `admin_puppet_path`. A configuration update copies these modules to each node's `/etc/puppet/modules` directory and applies the

`/etc/puppet/modules/master/manifests/site.pp` manifest on each node. Updates can be applied toward the group of both nodes via the `common` role or to a single node via the `web` or `mail` role.

One command, `pkg-grep`, conveniently wraps native methods to find installed packages.

Finally, a "manual" installation of Netcat is configured for Solaris 10. `nc-110-sol10-x86-local.gz` should be located in `admin_support_path`. An attempt to install the pacakge will first run a check for existence (`nc.chk.s10`). If it fails, the package will be copied to the destination (`nc.dst.s10`) and the installation script (`nc.exe.s10`) will be run using a specific path (`nc.pth.s10`).

1.5 Motivation

What we're trying to achieve with the layout and configuration above:

- **Ease of adoption**: We want team members to understand and be able to contribute to the site configuration as quickly as possible.

  The simple INI format reduces the potential for syntax errors, is friendly to line parsers, and lends itself well to visual pattern recognition, copied-and-pasted chunks, and manual editing in general.

  The tight-coupling between config files makes it easy to make sense of (and troubleshoot) the admin environment. "Where are my config files?" Look in `.fabricrc`. "Why can't I contact a node?" Start with `ssh_config_path`. "Where is that installer?" Double check your `admin_support_path`. "How do I run that command?" Consult the `[exe]` section.

- **Minimal overhead**: We want to install as little as possible and write as little as possible to accomplish as much as possible.

  Fabric (which includes SSH support) alone will get you started with ad-hoc tasks and Puppet will handle general system configuration. By evolving your custom Puppet modules and contributing to your commands in `[exe]`, these tools may be all you need. [1]

  In terms of layout, separating a user's environment (SSH keys) from shared projects (site-specific, group-accessible files) from publicly available third party packages should be considered minimal, but required, overhead.

- **Domain-less configuration**: We want to be able to take our configurations, or portions thereof, with us. We want to share them between sites. We don't want them locked in a binary format in a running service. We don't want to define our systems in terms of screenshots.

  Puppet modules are one way of sharing reusable system configuration components, and layered INI files make it easy to splice and dice configuration sections and options. Admins can phase components in our out of an environment by editing the appropriate options in their `.fabricrc` (ex: `admin_cfg_path` and `admin_cfg_include`) or at runtime using `fab --set ...`.

- **Encouragement**: Finally, we want to make it easy to "do the right things" by reducing the startup expense for each administrative component. And the layout itself should be leveraged for convenience to encourage enthusiasm in it's upkeep.

1.6 Use cases

"OK. What *really* motivated this?" In a nutshell: the desire to free *administration* from the *domain*.

[1] The admin's *environment* is light - both conceptually and in terms of required packages. Administered nodes can be laden with any and all software available to native package managers or found in the `admin_support_path`.

1.6.1 Reading down, writing up

Barriers between networks at different security levels can lead to unncessarily divergent methods of administration despite their similarities; no matter where you're at, you're likely to manage DNS, NTP, and SNMP, and you're likely to wonder what processes are resource hogs, and how much space is available.

1.6.2 VPNs and LANs

Unplug here, go there, same head honcho.

CHAPTER

TWO

GETTING STARTED

Familiarize yourself with Fabric and Puppet.

2.1 Installation

Install this package via:

- Mercurial:

  ```
  hg clone https://www.bitbucket.org/sourpoi/minimin /tmp/minimin
  ```

- pip:

  ```
  pip install minimin
  ```

- source:

  ```
  wget http://pypi.python.org/packages/...
  ```

For this quickstart, we'll assume that the extracted package is located at `/tmp/minimin` and that our target "fabfile" is `/tmp/minimin/lib/minimin/fabfile.py` (we'll set this as the default fabfile in *Admin user's configuration*).

2.2 Layout

We'll be using a layout where our admin user's home is at `/home/admin`, our primary project is at `/tmp/myproj`, a common project with shared configurations is at `/tmp/common`, and 3rd-party support is in `/tmp/support`:

```
/
+-- home
|   +-- admin
|       +-- .fabricrc
|       +-- .ssh
|           +-- config
|           +-- id_some_key
|           +-- id_some_key.pub
+-- tmp
    +-- myproj
    |   +-- etc
    |   |   +-- site.ini
    |   +-- puppet
```

```
|       +-- modules
|           +-- myproj
|               +-- manifests
|                   +-- site.ini
+-- common
|   +-- etc
|   |   +-- common.ini
|   +-- puppet
|       +-- modules/
+-- support
```

2.2.1 Skeleton

Create a sample environment with directories for third-party support, shared and reusable configurations, private configurations, and the admin user's home:

```
mkdir -p /tmp/support                       # public, third-party packages
mkdir -p /tmp/common/{etc,puppet/modules}   # common, shared configs
mkdir -p /tmp/myproj/{etc,puppet/modules}   # project, site-specific configs
mkdir -p /home/admin                        # admin's home
touch /home/admin/.fabricrc                 # admin's Fabric config
mkdir /home/admin/.ssh                      # admin's SSH home
touch /home/admin/config                    # admin's SSH config
```

2.2.2 Admin user's configuration

Include some *suggested* environment settings in the Fabric `rcfile` (`~/.fabricrc`):

```
cat >> /home/admin/.fabricrc <<EOF
fabfile = /tmp/minimin/lib/minimin/fabfile.py
ssh_config_path = /home/admin/.ssh/config
use_ssh_config = True
warn_only = True
EOF
```

Include the **required** Fabric environment setting in the Fabric `rcfile` (`~/.fabricrc`):

```
cat >> /home/admin/.fabricrc <<EOF
admin_support_path = /tmp/support
admin_puppet_path = /tmp/myproj/puppet/modules /tmp/common/puppet/modules
admin_cfg_path = /tmp/myproj/etc /tmp/common/etc
admin_cfg_include = site.ini common.ini
admin_ssh_home = /home/admin/.ssh
EOF
```

2.2.3 Project configuration

First let's satisfy our `site.ini` inclusion. We intend this to be a private, project-specific configuration so we put it in `/tmp/myproj/etc` (instead of `/tmp/common/etc`). Our `admin_cfg_path` will first look for `site.ini` in `/tmp/myproj/etc`:

```
cat > /tmp/myproj/etc/site.ini <<EOF
[node]
websvr = f15
```

```
mailsvr = s10

[role]
common.nodes = websvr mailsvr
common.puppet.mods = myproj
common.puppet.modpath = /etc/puppet/modules
common.puppet.site = /etc/puppet/modules/myproj/manifests/site.pp

[exe]
kick-web.txt = restart the web server
kick-web.f15 = service httpd restart

kick-mail.txt = restart the mail server
kick-mail.s10 = svcadm restart sendmail
EOF
```

2.2.4 Common/shared configuration

Next let's include our shared configuration `common.ini` in `/tmp/common/etc`. This contains options that aren't considered private, may be shared between projects, and may have come from a public source:

```
cat > /tmp/common/etc/common.ini <<EOF
[exe]
pkg-grep.txt = look for a package name matching {str}
pkg-grep.f15 = rpm -qa | grep {str}
pkg-grep.s10 = pkginfo | grep -i {str}

[install]
url-sfw10 = ftp://ftp.sunfreeware.com/pub/freeware/intel/10

nc.txt.s10 = Netcat, the all-purpose network tool SMCnc
nc.url.s10 = %(url-sfw10)s/nc-110-sol10-x86-local.gz
nc.src.s10 = nc-110-sol10-x86-local.gz
nc.dst.s10 = /tmp/nc-110-sol10-x86-local.gz
nc.pth.s10 = /usr/sbin:/usr/bin
nc.req.s10 = which pkgadd pkgrm gunzip
nc.chk.s10 = /usr/local/bin/nc -h 2>&1 | grep 'v1.10'
nc.del.s10 = pkgrm SMCnc
nc.exe.s10 = #!/bin/sh
    cd /tmp && \
    gunzip nc-110-sol10-x86-local.gz && \
    pkgadd -d nc-110-sol10-x86-local all
```

2.3 Workflow

1. Install Minimin. See *Installation*.
2. Set up the environment. See *Layout*.
3. Generate SSH keys, configure nodes to use them. See *SSH key distribution*.
4. Install Puppet on nodes. See *Puppet installation* and *[install]*.
5. Apply Puppet configurations to nodes. See *Applying Puppet configuration* and *[role]*. Use a quick-n-dirty Puppet module to jumpstart service configurations. See *Quick-n-dirty Puppet modules*.

6. Query nodes, manage services using Fabric. See *Ad-hoc commands, queries, and service management* and *[exe]*.

2.3.1 SSH key distribution

Generate an SSH key to use on both nodes by providing the local username (`admin`), username on the remote node (`root`), and an indicator of the key's purpose (`SHARED`):

```
fab sshkeygen:admin,root,SHARED
```

This will create:

```
/home/admin/.ssh/id_admin_root_SHARED
                 id_admin_root_SHARED.pub
```

Note the suggested `ssh_config` additions, and add the following to `/home/admin/.ssh/config`:

```
cat >> /home/admin/.ssh/config <<EOF
Host *
IdentitiesOnly yes
GSSAPIAuthentication no

Host websvr
HostName 192.168.1.2
IdentityFile /home/admin/.ssh/id_admin_root_SHARED
User root

Host mailsvr
HostName 192.168.1.3
IdentityFile /home/admin/.ssh/id_admin_root_SHARED
User root
EOF
```

Next, configure your target nodes to respect your SSH key. This generally consists of:

1. Configuring the firewall to permit SSH access and restarting the firewall.
2. Configuring `/etc/ssh/sshd_config` to use public keys and restarting `sshd`.
3. Appending your key(s) to the appropriate user's `~/.ssh/authorized_keys` file.

The details will vary between operating systems, but if you have SSH access and can login manually via root password, you might try using a known-good `sshd_config` and the included "pure-ssh" push/append included.

1. Create a basic `sshd_config` to distribute:

   ```
   cat >> /tmp/sshd_config <<EOF
   Protocol 2
   Port 22
   HostbasedAuthentication yes
   HostKey /etc/ssh/ssh_host_rsa_key
   PasswordAuthentication yes
   PermitRootLogin yes
   AuthorizedKeysFile .ssh/authorized_keys
   ChallengeResponseAuthentication no
   GSSAPIAuthentication no
   Subsystem sftp /usr/libexec/openssh/sftp-server
   AllowTcpForwarding no
   X11Forwarding no
   EOF
   ```

2. Reconfigure/restart `sshd` and append your new SSH key to root's `authorized_keys`:

```
fab -uroot -Rcommon -- cp /etc/ssh/sshd_config /etc/ssh/sshd_config.orig
fab -uroot -Rcommon -- ssh-keygen -trsa -f /etc/ssh/ssh_host_rsa_key
fab -uroot -Rcommon sshput:/tmp/sshd_config,/etc/ssh/sshd_config
fab -uroot -Rcommon -- service sshd restart

PUBKEY='/home/admin/.ssh/id_admin_root_SHARED'
AUTHKEYS='/root/.ssh/authorized_keys'
fab -uroot -Rcommon sshput:$PUBKEY,$AUTHKEY,append=True
fab -uroot -Rcommon -- chmod 600 $AUTHKEY

fab sshup
```

2.3.2 SSH headache

Firewall access and SSH support on each node is part of the minimal overhead, but it can be tedious. Many OS installs do not include firewall access, SSH public key or SSH root access by default. You can ease the pain by:

1. enabling SSH during the OS install with a *kickstart <http://fedoraproject.org/wiki/Anaconda/Kickstart>* file or installation server,
2. speeding up manual password entries using screen or tmux copy-paste, or
3. scripting the password entries using expect or key distribution with `ssh-copy-id`.

2.3.3 Puppet installation

If a package manager is configured and the node has internet access (for example, on our `websvr` above running Fedora 15), you might be able to install Puppet by running:

```
fab -Hwebsvr -- yum -y install puppet
```

However, for our `mailsvr` node running Solaris 10, we'll configure manual installs. This is a little involved, and requires the following packages to satisfy dependencies `libiconv` -> `gcc` -> `ruby` -> `facter` -> `puppet`:

```
[install]
url-ibs10 = http://www.ibiblio.org/pub/packages/solaris/freeware/intel/10
url-ruby18 = http://ftp.ruby-lang.org/pub/ruby/1.8
url-puplab = http://downloads.puppetlabs.com

libiconv.txt.s10 = libiconv x86 SMCliconv
libiconv.url.s10 = %(url-ibs10)s/libiconv-1.11-sol10-x86-local.gz
libiconv.src.s10 = libiconv-1.11-sol10-x86-local.gz
libiconv.dst.s10 = /tmp/libiconv-1.11-sol10-x86-local.gz
libiconv.chk.s10 = pkginfo -l SMCliconv | grep VERSION | grep '1.11'
libiconv.exe.s10 = #!/bin/sh
    cd /tmp && \
    gunzip libiconv-1.11-sol10-x86-local.gz && \
    pkgadd -d libiconv-1.11-sol10-x86-local all

gcc86.txt.s10 = GNU x86 C compiler SMCgcc
gcc86.url.s10 = %(url-ibs10)s/gcc-3.4.6-sol10-x86-local.gz
gcc86.src.s10 = gcc-3.4.6-sol10-x86-local.gz
gcc86.dst.s10 = /tmp/gcc-3.4.6-sol10-x86-local.gz
gcc86.req.s10 = libiconv.s10
gcc86.chk.s10 = /usr/local/bin/gcc -v 2>&1 | grep 'gcc version 3.4.6'
```

```
gcc86.exe.s10 = #!/bin/sh
    cd /tmp && \
    gunzip gcc-3.4.6-sol10-x86-local.gz && \
    pkgadd -d gcc-3.4.6-sol10-x86-local all

ruby.txt.s10 = Ruby general-purpose programming language from %(url-sfw10)s
ruby.url.s10 = %(url-ruby18)s/ruby-1.8.7-p72.tar.gz
ruby.src.s10 = ruby-1.8.7-p72.tar.gz
ruby.dst.s10 = /tmp/ruby-1.8.7-p72.tar.gz
ruby.req.s10 = gcc86.s10
ruby.chk.s10 = /usr/local/bin/ruby -v | grep "ruby 1.8.7"
ruby.pth.s10 = /usr/sbin:/usr/bin:/usr/sfw/bin:/usr/ccs/bin
; installs to /usr/local/bin/ruby
ruby.exe.s10 = #!/bin/sh
    cd /tmp && \
    gunzip ruby-1.8.7-p72.tar.gz && \
    tar -xvf ruby-1.8.7-p72.tar && \
    rm -r ruby-1.8.7-p72.tar && \
    cd ruby-1.8.7-p72 && \
    ./configure && \
    make && \
    make install && \
    ln -s /usr/local/bin/ruby /usr/bin/ruby

facter.txt.s10 = Facter host-assessment tool from %(url-sfw10)s
facter.url.s10 = %(url-puplab)s/facter/facter-1.6.6rc2.tar.gz
facter.src.s10 = facter-1.6.6rc2.tar
facter.dst.s10 = /tmp/facter-1.6.6rc2.tar
facter.req.s10 = ruby.s10
facter.chk.s10 = /usr/local/bin/facter -v | grep "1.6.6"
facter.pth.s10 = /usr/sbin:/usr/bin:/usr/local/bin
facter.exe.s10 = #!/bin/sh
    cd /tmp && \
    tar -xf facter-1.6.6rc2.tar && \
    cd facter-1.6.6rc2 &&
    ./install.rb && \
    ln -s /usr/local/bin/facter /usr/bin/facter

; Note underscore to dash change after tarball extract.
puppet.txt.s10 = Puppet configuration manager from source
puppet.url.s10 = %(url-puplab)s/puppet/puppet_2.7.10.orig.tar.gz
puppet.src.s10 = puppet_2.7.10.orig.tar.gz
puppet.dst.s10 = /tmp/puppet_2.7.10.orig.tar.gz
puppet.req.s10 = ruby.s10 facter.s10
puppet.chk.s10 = /usr/local/bin/puppet -V | grep "2.7.10"
puppet.pth.s10 = /usr/sbin:/usr/bin:/usr/local/bin
puppet.exe.s10 = #!/bin/sh
    cd /tmp && \
    gunzip puppet_2.7.10.orig.tar.gz && \
    tar -xf puppet_2.7.10.orig.tar && \
    cd puppet-2.7.10 && \
    ./install.rb && \
    ln -s /usr/local/bin/puppet /usr/bin/puppet
```

With all of the above in our Minimin config, we can install Puppet on `mailsvr` using:

```
fab -Hmailsvr install:libiconv
fab -Hmailsvr install:gcc86
```

```
fab -Hmailsvr install:ruby
fab -Hmailsvr install:facter
fab -Hmailsvr install:puppet
```

Note: The URL options (`url-ibs10`, `url-ruby18` and `url-puplab`) aren't used for anything but documentation (yet).

More information on install configuration in *[install]*.

2.3.4 Applying Puppet configuration

To apply Puppet configurations, run updates on all nodes sharing a role:

```
fab -Rcommon apply
```

..or specify one or more hosts (nodes) as well as the role type:

```
fab -Rcommon -Hwebsvr apply
```

A dummy role named `nodes` contains all the configured nodes. If the puppet options are defined for the dummy role `nodes`:

```
[role]
nodes.puppet.mods = myproj
nodes.puppet.modpath = /etc/puppet/modules
nodes.puppet.site = /etc/puppet/modules/myproj/manifests/site.pp
```

..then you may use the `nodes` role as you would any other:

```
fab -Rnodes -Hwebsvr apply
```

More information on Puppet configuration using roles in *[role]*.

Rationale behind the role dependency in *Role dependency*.

2.3.5 Ad-hoc commands, queries, and service management

List configured commands and descriptions from the `[exe]` section:

```
fab showexe
```

Run the commands on specific nodes:

```
fab -Hwebsvr exe:kick-web
fab -Hmailsvr exe:kick-mail
```

Given our configuration above, we "just know" that `kick-web` won't work on node `mailsvr` because `[exe]` doesn't have the option `kick-web.s10` and `kick-mail` won't work on `websvr` because `[exe]` doesn't have the option `kick-mail.f15`.

A configured command may require keyword arguments:

```
[exe]
net-ping.txt = ping {node} 3 times and show stats
net-ping.f15 = /bin/ping -c3 {node}
net-ping.s10 = /usr/sbin/ping -s {node} 56 3
```

Since `net-ping` is configured for all our node suffixes, we can `ping` a common target (`192.168.1.1`) from both nodes like this:

```
fab -Rcommon exe:net-ping,node=192.168.1.1
```

Include the required keywords in the description of the configured command.

More information on Puppet configuration using roles in *[exe]*.

2.3.6 On my laptop

Here's my typical workflow:

```
fab vbmlist                      # available VirtualBox VMs
fab vbmstart:{vmname}            # start a VM
fab vbmscan:192.168.1.2-40       # reconcile VM DCHP results with ssh_config
fab sshup                        # confirm access
fab showcfg:flat | grep role.    # remind me what I'm working on
fab install:{pkg}                # install something on our nodes
fab -H{node} config              # apply Puppet configuration to a test node
fab -R{role} config              # apply Puppet configuration role nodes
fab showexe | grep {cmd}         # describe pre-configured {cmd}
fab -R{role} exe:{cmd},opt=val   # check some aspect of our nodes
```

CHAPTER

THREE

MINIMIN CONFIGURATION

3.1 Fabric's `rcfile` (`.fabricrc`)

The default location for a user's Fabric `rcfile` is `~/.fabricrc`.

We use it to set the following paths (with example values):

```
; directories to 3rd party support files
admin_support_path = /export/public/src /export/private/lib
; directories holding Puppet modules
admin_puppet_path = /etc/puppet/modules /opt/puppet/modules
; directories containing minimin .ini files
admin_cfg_path = /home/admin/etc /opt/project
; administrator's SSH home directory (ssh_home)
admin_ssh_home = /home/admin/.ssh
```

And we specify our active Minimin config files, to be found in one of the diretories in `admin_cfg_path`:

```
; active minimin config files (found in admin_cfg_path)
admin_cfg_include = conf1.ini conf2.ini
```

We also set some native Fabric options - a default fabfile so we don't have to be in the same directory as one or provide it explicitly, and an `ssh_config` file to explicitly map our configured node names to their respective network addresses and keys:

```
; active/default fabfile
fabfile = /home/admin/proj/minimin/lib/minimin/fabfile.py
; personal/administrative ssh_config file
ssh_config_path = /home/admin/.ssh/config
; activate the ssh_config
use_ssh_config = True
```

3.2 Minimin INI config

3.2.1 Included config files

INI-style config files must be in one of the `admin_cfg_path` directories. `admin_cfg_include` configs are included first, followed by those provided on the commandline using `--set cfg_include=`.

All the config files are combined into a single, master configuration.

3.2.2 [node]

Our configuration begins with a list of nodes. Each node specifies an OS/platform:

```
[node]
websvr = f15
mailsvr = s10
```

The platform is a somewhat arbitraty identifier used to determine the relevancy of options in other sections:

```
[exe]
pkg-rm.txt = remove package {pkg} *
pkg-rm.f15 = /bin/rpm -e {pkg}
pkg-rm.s10 = /usr/sbin/pkgrm -n {pkg}
```

3.2.3 [role]

Roles bundle a list of nodes and their common requirements:

```
[role]
; targeted node(s)
{rolename}.nodes = {node1} {node2}
; modules to copy to the node(s)
{rolename}.puppet.mods = {module1} {module2}
; target module dir
{rolename}.puppet.modpath = /etc/puppet/modules
; target site manifest
{rolename}.puppet.site = /etc/puppet/modules/prod/manifest/site.pp
```

An arbitrary `{rolename}` is used to identify a list of nodes (`{rolename}.nodes`) on the command line using `fab -R{rolename}`.

The main purpose is to determine:

1. a list of nodes' Puppet module dependencies (`{rolename}.puppet.mods`), located in `admin_puppet_path`,
2. where those modules should be located on each node (`{rolename}.puppet.modpath`), and
3. what site manifest should be applied on each node (`{rolename}.puppet.site`, which should be a `site.pp` manifest in one of the required modules).

You may provide a role with no nodes, perhaps for testing:

```
[role]
test1.puppet.mods = testmodA testmodB
test1.puppet.modpath = /etc/puppet/test/modules
test1.puppet.site = /etc/puppet/test/modules/testmodA/manifests/site.pp
```

..and then test the role-based configuration on a single host:

```
fab -Rtest1 -Htestsvr1 config
```

Likewise, you don't need to provide the Puppet options if all you want is a convenient node list:

```
[role]
new.nodes = node1A node1B
```

3.2.4 [exe]

Command "aliases" are configured in the `exe` section:

```
[exe]
{alias}.txt = {description}
{alias}.{os1} = {cmd1}
{alias}.{os2} = {cmd2}
```

The `{alias}`, for now, follows a `{category}-{action}` format, though this is only for convenient grouping by category.

Commands may require keyword values as `{key}`:

```
[exe]
pkg-info.txt = information on package {pkg}
pkg-info.f15 = /bin/rpm -qi {pkg}
pkg-info.s10 = /usr/bin/pkginfo -l {pkg}
```

The keywords should be used in the description to make `fab showexe` summaries more useful.

3.2.5 [install]

Manual installs require a number of options per package (`{pkgname}`) and OS/platform suffix (`{os}`):

```
[install]
; helper option interpolated later
{os}.pth = /sbin:/usr/sbin:/bin:/usr/bin

; description with version and formal package name/id
{pkgname}.txt.{os} = a fantastic application MyApp 1.2-3
; pacakge file in admin_support_path
{pkgname}.src.{os} = myapp-1.2-3.tgz
; destination filename on node
{pkgname}.dst.{os} = /tmp/myapp-1.2-3.tgz
; executable path (interpolated)
{pkgname}.pth.{os} = %({os}.pth)s
; cmd to check for existence of package
{pkgname}.chk.{os} = myapp -V
; cmd to check for support requirements
{pkgname}.req.{os} = which tar
; deletion script
{pkgname}.del.{os} = #!/bin/sh
    rm -rf /opt/myapp-1.2-3
; installer script
{pkgname}.exe.{os} = #!/bin/sh
    cd /tmp && \
    tar -xzf myapp-1.2-3.tgz && \
    sh /tmp/myapp-1.2-3/install-me
```

The URL option isn't used yet, but it may be handy for documentation (note the interpolation of our URL value as well as the default path):

```
[install]
url-sfw10 = ftp://ftp.sunfreeware.com/pub/freeware/intel/10

nc.txt.s10 = Netcat, the all-purpose network tool SMCnc
nc.url.s10 = %(url-sfw10)s/nc-110-sol10-x86-local.gz
```

```
nc.src.s10 = nc-110-sol10-x86-local.gz
nc.dst.s10 = /tmp/nc-110-sol10-x86-local.gz
nc.pth.s10 = /usr/sbin:/usr/bin
nc.req.s10 = which pkgadd pkgrm gunzip
nc.chk.s10 = /usr/local/bin/nc -h 2>&1 | grep 'v1.10'
nc.del.s10 = pkgrm SMCnc
nc.exe.s10 = #!/bin/sh
    cd /tmp && \
    gunzip nc-110-sol10-x86-local.gz && \
    pkgadd -d nc-110-sol10-x86-local all
```

CHAPTER

FOUR

QUICK-N-DIRTY PUPPET MODULES

The goal of a quick-n-dirty Puppet module is to reduce the common "package, file, service" pattern to a bare minimum by..

1. describing classes in an admin's configuration:

```
[qnd]
ntp.config.default  = mysite:ntp:0644:ntp_conf:/etc/ntp.conf:stratnum:servers
ntp.config.solaris  = mysite:ntp:0644:ntp_conf:/etc/inet/ntp.conf
ntp.service.default = ntp:ntpd
ntp.service.solaris = ntp:ntp
ntp.install.redhat  = ntp:ntp
ntp.install.solaris = ntp:CSWntp:pkgutil
```

2. ..and generating a quick-n-dirty (here, `qnd`) module:

```
puppet
+-- modules
    +-- qnd
        +-- ntp
            +-- default.pp      # class qnd::ntp::default
                +-- config.pp   # class qnd::ntp::config
                +-- install.pp  # class qnd::ntp::install
                +-- service.pp  # class qnd::ntp::service
```

3. ..that supports these kinds of declarations:

```
class {'qnd::ntp::install': }
class {'qnd::ntp::config': $ntp_conf_stratnum => '10' }
class {'qnd::ntp::service': $ntp_subs => File['ntp_conf']}
```

The idea is that the descriptions in the admin's configuration provide a quick reference for common and concise declarations, and important configuration information is separated from the class implementations (which can be regenerated at will).

4.1 Defining values

Start by adding a `[qnd]` section to your admin configuration that follows this convention:

```
[qnd]
ntp.config.default  = mysite:ntp:0644:ntp_conf:/etc/ntp.conf:stratnum:servers
ntp.config.solaris  = mysite:ntp:0644:ntp_conf:/etc/inet/ntp.conf
ntp.service.default = ntp:ntpd
```

```
ntp.service.solaris = ntp:ntp
ntp.install.redhat  = ntp:ntp
ntp.install.solaris = ntp:CSWntp:pkgutil
```

Options follow a `submodule.classname.platform` pattern. `submodule` is a custom, administrative name for the target service across all platforms. `classname` is one of `config`, `service`, or `install` depending on which class you're configuring. `platform` is a string that Puppet should be able to apply to a conditional `$::operatingsystem` switch (`default` is acceptable).

Values are colon-separated parameters that are passed to the underlying class generators' `add()` method (see `minimin.fabqnd`). The first parameter is a unique identifier for the class (type, and following parameters are values specific to the class appropriate for the corresponding platform.

A line-by-line explanation:

1. `ntp.config.default = mysite:ntp:0644:ntp_conf:/etc/ntp.conf:stratnum:servers`

   The default `ntp` configuration (`ntp.config.default`) uses one configuration template found in the `mysite` module named `ntp_conf`; this template should exist at `mysite/templates/ntp_conf`. The user and mode of the file are `ntp` and `0644`. Content will be written to `/etc/ntp.conf`. The names `stratnum` and `servers` will create two template variables named `ntp_conf_stratnum` and `ntp_conf_servers`. The template determines whether these variables are required (or even used).

   The point of having a template home outside the quick-n-dirty module is to keep actual configuration in a module that is actively maintained; the quick-n-dirty module will need to be re-created if the admin's `[qnd]` options change.

   For the purposes of our configuration, we're tailoring our `default` values to a Red Hat/Fedora-based platform.

2. `ntp.config.solaris = mysite:ntp_conf:/etc/inet/ntp.conf`

   As above, with the destination of the `ntp_conf` content on Solaris being written to `/etc/inet/ntp.conf`.

   No template variables are provided, since all but the first set of variables defined for a unique template identifier (`ntp_conf`) are ignored; we are trying to enforce uniform configuration behavior.

3. `ntp.service.default = ntp:ntpd`

   The `ntp` portion of `ntp:ntpd` sets a service identifier, the default value of which is `ntpd`. This means that the platform's native service management facility should know how to start and stop something named `ntpd`. `ntpd` satisfies Red Hat and Fedora's `service` facility.

4. `ntp.service.solaris = ntp:ntp`

   Again, we override the service name for Solaris (the second `ntp` in `ntp:ntp`), since `svcadm` recognizes `ntp` instead of `ntpd`.

5. `ntp.install.redhat = ntp:ntp`

   For Red Hat, we define a native (probably `yum`) package installation for the package named `ntp`.

6. `ntp.install.solaris = ntp:CSWntp:pkgutil`

   For Solaris, we specify a third-party package installer `pkgutil` that will understand how to fetch `CSWntp`.

4.2 Generating the module

To create our quick-n-dirty module in the `common` directory documented in *Layout*, we'll generate and run a script that will create the module in `/proj/common/puppet/modules/qnd`:

```
fab --hide=running,status qndgen:/proj/common/puppet/modules/qnd > /tmp/qnd.sh
sh /tmp/qnd.sh
```

Note that the `qndgen` command looks for the `[qnd]` section in our administrative configuration and creates submodules for all the options in this section. In the example above, we only defined values for `ntp`-prefixed options, so we only expect the `qnd/ntp/` submodule directory to be created.

4.2.1 Submodule files

The following files will be created after the script is run:

```
/
+-- proj
    +-- common
        +-- puppet
            +-- modules
                +-- qnd
                    +-- ntp
                        +-- default.pp  # class qnd::ntp::default
                        +-- config.pp   # class qnd::ntp::config
                        +-- install.pp  # class qnd::ntp::install
                        +-- service.pp  # class qnd::ntp::service
```

Managed components are organized as 'submodules'. Each submodule provides classes for installation, configuration, and service management independently via parameterized classes. 'Independently' means that a service class does not implicitly require the configuration class, the configuration class does not implicitly require the installation class, etc..

`default.pp` defines operating system-specific values for internal use by `install.pp`, `config.pp`, and `service.pp`.

4.3 Enable the module

So far we've only used our admin config to define quick-n-dirty NTP behavior and generate a Puppet module with classes that implement the behavior, but we haven't enabled the ability to use those classes in our site configuration.

Enable the quick-n-dirty module by..

1. ..verifying that the parent path leading to the module is in the admin's `.fabricrc` (in this case, the parent path is `/proj/common/puppet/modules`):

   ```
   admin_puppet_path = /proj/common/puppet/modules
   ```

2. ..and including the `qnd` module by name in the `[role]` section (see *Project configuration*):

   ```
   [role]
   common.nodes = websvr mailsvr
   common.puppet.mods = mysite qnd
   common.puppet.modpath = /etc/puppet/modules
   common.puppet.site = /etc/puppet/modules/mysite/manifests/site.pp
   ```

When we finally attempt to apply a Puppet configuration (see *Applying Puppet configuration*), `fab apply` will look in `/proj/common/puppet/modules` for `mysite` and `qnd` directories and copy both to the target node's `/etc/puppet/modules` before running `/etc/puppet/modules/mysite/manifests/site.pp`. Given the Puppet path and required modules for the `common` role, `/proj/common/puppet/modules/mysite/manifests/site.pp` must exist locally.

4.4 Use the classes

- `${svc}::config` - Config files, templates, and parameters. The default config file should provide a reasonable configuration, perhaps using a template that accomodates differences between operating systems. Since we don't expect a default template to accept enough custom parameters to satisfy every site, all files will accept a string to use as file content. The intention is for the site to use it's own config file templates (if necessary) with reasonable custom parameters and pass the resulting string representation (using `template('{sitemod}/{configfile}')`.
- `${svc}::service` - On the site level, the only (configurable) concerns are whether to enable or disable services (default `running`) and whether they should refresh themselves after a configuration change (default `${svc}_subscribe = []`,). The idea is that we probably want to run a service, but we can't take for granted that automatic restarts are appropriate.
- `${svc}::install` - This is only useful when a package manager is capable of verifying and installing a package by name.

Assuming all the above is intact, we can include `qnd::ntp` classes in the site configuration (`site.pp`). Here we invoke the default behavior defined above (see *Defining values*):

```
class {'qnd::ntp::install': }
class {'qnd::ntp::config': }
class {'qnd::ntp::service': }
```

Installation, configuration, and service management are independent - include only those classes that you want to use. To skip installation (perhaps you installed a package manually) and have a service restart after a change is made to a config file:

```
class {'qnd::ntp::config':  ntp_conf_stratnum => '4'}
class {'qnd::ntp::service': ntp_subs => File['ntp_conf']}
```

See the `minimin.fabqnd` for details on `config`, `service`, and `install` class parameters.

4.5 See also

- http://www.devco.net/archives/2009/09/28/simple_puppet_module_structure.php
- http://vstone.eu/puppet-module-patterns/
- http://www.example42.com/?q=understandExample42PuppetModules
- http://bombasticmonkey.com/2011/12/27/stop-writing-puppet-modules-that-suck/
- http://puppet-modules.git.puzzle.ch

CHAPTER

FIVE

MODULES

5.1 minimin.fabfile

Sample fabfile.py.

This may be used to set default Fabric behavior setting the `fabfile` option in `~/.fabricrc`:

```
fabfile = /home/admin/support/minimin.hg/lib/minimin/fabfile.py
```

See the source for details.

This may be used to set default Fabric behavior setting the `fabfile` option in `~/.fabricrc`:

```
fabfile = /home/admin/support/minimin.hg/lib/minimin/fabfile.py
```

See the source for details.

5.2 minimin.fabcfg

Shared configuration for Fabric tasks.

This module intends to serve as a singleton configuration object by providing the following attributes:

- *cfg_incl*: list of .INI files contributing to our effective configuration
- *cfg_path*: list of directories to find included .INI files (*cfg_incl*)
- *puppet_path*: list of directories to find Puppet modules
- *support_path*: list of directories to find support software/installers
- *ini*: *ConfigParser.SafeConfigParser* object, our effective configuration
- *inimap*: dictionary with nested *ini* dotted-options
- *extmap*: dictionary mapping nodes to their OS/arch extensions
- *nodes*: list of node names in our effective configuration
- *roledefs*: dictionary mapping role names to node lists
- *role*: active role (in *roledefs*)
- *sshcfg*: *ssh.SSHConfig* dictionary representing our `ssh_config`

class minimin.fabcfg.**IniMap**(*ini*)
Bases: object

A flat key, val mapper that pulls info from a dict.

gen(**match*)

minimin.fabcfg.**ext**()
Retrieve current node's OS/arch extension.

Returns node's OS/arch extension as mapped in .extmap.

Exception Raises KeyError if extension is not found.

5.3 minimin.fabmin

Minimin's basic adminisrative commands

Three commands are the basis of minimal administration - manual installs, one-off commands, and system configuration:

- install() - check, copy, and install packages in [install]
- exists() - run the package's .chk script to test for existence
- delete() - run the package's .del script to delete it
- exe() - interface to commands configured in [exe]
- config() - apply Puppet configurations defined in [role]

To help make sense of your configuration and environment:

- showcfg() - effective INI configuration
- showenv() - Fabric's env
- showexe() - list of commands configured in [exe]
- showpaths() - paths configured in .fabricrc/env.rcfile

minimin.fabmin.**config**(*noop=False*)
Configure nodes by copying Puppet modules and running 'puppet apply'.

Parameters

- *noop*: do a dry-run instead of applying the manifest (default False)

Applying the Puppet configuration will copy modules to the node's modulepath and run the site manifest on the node, distinguishing the node with using the node_name_value.

We only copy modules (using the convenient fabric.api.put() - we do not prune. This means that a new manifest tree might not take effect if, for example, a new manifests/ntp.pp is prefered over an existing but outdated manifests/ntp/init.pp when using class {'mod::ntp':}.

Todo

An rsync --delete might be useful here.

Example:

```
fab -Rrole config                 # apply to hosts sharing role
fab -Rrole -Hh1,h2 config         # apply to given hosts using role
fab -Rrole -Hh1 config:noop=True  # dry-run only, don't actually apply
```

minimin.fabmin.**delete**(*pkg*)

Delete a manually-installed package.

Parameters

- *pkg*: package name (first option segment in `[install]` section)

Returns output from `{pkg}.del.{os}` command

Example:

```
fab delete:monit
```

minimin.fabmin.**exe**(*cmd*, *final=False*, *sshlog=None*, ***kw*)

Execute a pre-configured command.

Parameters

- *cmd*: command option name
- *final*: skip command name look-up if True (optional, default False)
- *sshlog*: file to log SSH debugging info

Keywords `optcmd` command keywords (see `exeinfo()`)

This exists to fill the gap between elegant configuration management and ad-hoc, one-off commands by way of configuring those commands as simple INI-style options in this format:

```
[exe]
{cmd} = command summary
{cmd}.{os} = mycmd -a {arg1}
```

Example:

```
fab -Hh1,h2 exe:sys-who
fab -Hlocalhost exe:sys-who-f15  # unconfigured localhost requires os
```

minimin.fabmin.**exists**(*pkg*)

Check the existence of a manually-installed package.

Parameters

- *pkg*: package name (first option segment in `[install]` section)

Returns output from `{pkg}.chk.{os}` command

Example:

```
fab exists:monit
```

minimin.fabmin.**install**(*pkg*)

Install a package manually.

Parameters

- *pkg*: package name (first option segment in `[install]` section)

Returns output from `exe()` or None if check `{pkg}.chk.{os}` succeeds

Install options for package `{pkg}` on OS/architecture `{os}`:

```
[install]
{pkg} = Short description, include version in long form
{pkg}.url.{os} = http://host/app.tgz      ; download URL
{pkg}.src.{os} = app.tgz                  ; file in admin_support_path
{pkg}.dst.{os} = /tmp/app.tgz             ; absolute destination path
{pkg}.pth.{os} = /bin:/sbin               ; remote $PATH
{pkg}.req.{os} = which tar                ; test for requirements
{pkg}.chk.{os} = app -V | grep '1.0'      ; test for existence
{pkg}.exe.{os} = tar -C/tmp -xzf app.tgz  ; install script
{pkg}.del.{os} = rm -rf /tmp/app          ; deletion script
```

This function will:

1. Run `.req` and `.chk` scripts using the `.pth $PATH` on the node and continue the installation only if `.req` succeeds and `.chk` fails. If requirements are not met we abort, erring on the side of uniformity.
2. Copy the first matching `.src` file found in `env.admin_support_path` to remote `.dst`.
3. Run `.exe` using `.pth` on the node.

Example:

```
fab install:gcc
```

minimin.fabmin.**qndput**(*osname=None*, **pkgs*)

Copy quick-n-dirty installers to the expected package source paths.

Parameters

- *osname*: operating system name (third portion of quick-n-dirty option)
- *pkgs*: list of package to install (requires an `*.install.*` option)

Untested: use with a sense of humor.

minimin.fabmin.**showcfg**(**args*, ***kwargs*)

Show the effective configuration in raw format.

Parameters

- *flat*: replace default INI-style output with a flat list of opts prepended by section names (optional, default False)

Example:

```
fab --hide=running,status showcfg > /tmp/showcfg.ini
fab showcfg:flat | grep role
```

minimin.fabmin.**showenv**(**args*, ***kwargs*)

Show Fabric's environment settings (`fabric.api.env`).

Example:

```
fab --hide=running,status showenv
fab showenv | grep admin_
```

minimin.fabmin.**showexe**(**args*, ***kwargs*)

Print info on a configured command (or list all summaries).

Parameters

- *cmd*: command name (optional, default None)

Keywords keywords to populate command arguments (optional)

Example:

```
fab --hide=status,running showexe # list all command names and summaries
fab showexe:proc-listen,port=22   # keyword fills {port} placeholder
```

minimin.fabmin.**showpaths**(**args*, ***kwargs*)

Show our potential and actual file inclusions for a given role.

We use our configured paths and file inclusions (set in fabricrc or provided via `fab --set`) and provide the list of paths we look for followed by the list of files and directories we actually have. This is a quick way to double check running path sanity. Compare to:

```
fab showenv | grep admin_
```

Example:

```
fab --hide=running,status showpaths
```

minimin.fabmin.**support**(**srcs*)

Copy/sync support sourcepackages.

Parameters

- *srcs*: a list of specific packages to sync (optional)

Example:

```
fab -Rrole support
fab -Rrole support:awstats-7.0.tar.gz
```

5.4 minimin.fabssh

SSH commands

SSH helpers include:

- `sshkeygen()` - create local SSH keys as `id_localuser_remotuser_host`
- `sshput()` - use basic SSH to copy a file (when node is missing SCP/SFTP)
- `sshup()` - check access to nodes configured in `ssh_config`

minimin.fabssh.**sshfwdlocal**(*src*, *dst=None*)

Forward a local port to a port on a remote host.

Parameters

- *src*: source port (port on localhost)
- *dst*: destination port (port on remote host)

This is intended to make it easier to use local commands to manage remote services:

```
fab -Hcouch sshfwdlocal:5984
curl -XPUT 'http://localhost:5984/newdb'
```

This function relies on an `ssh_config` to determine remote user, host, and port.

Example:

```
fab -Hhost sshfwdlocal:8080
fab -Hhost sshfwdlocal:8080,80
```

minimin.fabssh.**sshkeygen**(**args*, ***kwargs*)

Create a new SSH key pair: write keys to SSH home and print config.

Parameters

- *localuser*: username of local admin (you)
- *remoteuser*: username of user on remote node
- *nodes*: list of nodes to generate keys for

For nodes `h1` and `h2`, create SSH keys in your SSH home (configured as the `admin_ssh_home` value in `~/.fabricrc`):

```
fab sshkeygen:me,root,h1,h2  # create id_me_root_h1 and id_me_root_h2
```

Using manual password logins to begin with, we copy the appropriate SSH server configuration (enabling the use keys) and our keys to the target nodes and restart the service:

```
fab -Hh1,h2 sshput:sshd_config,/etc/ssh/sshd_config,backup
fab -Hh1,h2 sshput:id_admin.pub,/root/.ssh/authorized_keys
fab -Hh1,h2 -- service sshd restart
```

Example:

```
fab sshkeygen:admin,root,n1,n2
fab sshkeygen:admin,root,SHARED
fab --set admin_ssh_home='/alt/ssh' sshkeygen:admin,root,n1
```

minimin.fabssh.**sshput**(*src*, *dst*, *append=False*)

Copy a file using local SSH (work around lack of sftp on target node).

Parameters

- *src*: path to local file
- *dst*: path of destination file on remote node
- *append*: (optional) append to file rather than overwriting it

This bare-bones copying method addresses SSH configurations on new nodes which do not support `sftp` or `scp` (and frustrate conveniences like Fabric's `put` and `append`).

Ideally, most file copies should be syncs via CMS.

This function relies on an `ssh_config` to determine remote user, host, and port.

Example:

```
fab -Hh1 sshput:id_admin.pub,/root/.ssh/authorized_keys,append
fab -Hh1 sshput:motd,/etc/motd
```

minimin.fabssh.**sshup**(**args*, ***kwargs*)

Check availability of configured SSH nodes/hosts.

For each host configured in the `ssh_config_path` (typically `~/.ssh/ssh_config`), retrieve the host key and attempt to run `uname -a` to confirm a connection.

Example:

```
fab sshup
```

5.5 minimin.fabvbm

Fabric VirtualBox commands

minimin.fabvbm.**vbmlist**(**args*, ***kwargs*)
Print a list of VirtualBox VMs (existing and running).

Example:

```
fab --hide=running,status vbmlist
fab vbmlist
```

minimin.fabvbm.**vbmscan**(**args*, ***kwargs*)
Scan addresses and match them to listening VM names and MACs.

Parameters

- *addrs*: range of host address to scan using Nmap

This function helps work with VMs that use DHCP and wind up with an address that isn't configured in `ssh_config`. We run a ping sweep over a range of addresses to collect MAC addresses in our ARP table, then we match MACs with those in our guest VM configurations.

See `sshup()` and `vbmlist()`.

Example:

```
fab --hide=running,status vbmscan:192.168.1-40
fab vbmscan:192.168.1.1-40
```

minimin.fabvbm.**vbmshow**(**args*, ***kwargs*)
Print information on a VirtualBox VM.

Parameters

- *vmname*: name of the virtual machine

Example:

```
fab vbmshow:vmname
```

minimin.fabvbm.**vbmstart**(**args*, ***kwargs*)
Start a VirtualBox VM in headless mode.

Parameters

- *vmname*: name of the virtual machine

This is mainly for convenience after a VM is set up and SSH is running.

See `vbmlist()`.

Example:

```
fab vbmstart:vmname
```

minimin.fabvbm.**vbmstop**(**args*, ***kwargs*)
Stop a VirtualBox VM.

Parameters

- *vmname*: name of the virtual machine

See `vbmlist()`.

Example:

```
fab vbmstop:vmname
```

5.6 minimin.fabutil

Helper functions.

minimin.fabutil.**firstreal**(*parents*, *child*)
First path pointing to child file in a list of parent directories.

Parameters

- *parents*: list of parent directories
- *child*: file to find in parent directories

Returns full path to child file

Exceptions

- *IOError*: if real path to a child doesn't exist in parents

minimin.fabutil.**fmtprint**(*items*, *spc*, ***kw*)
Print a sequence of pairs with a certain spacing.

Parameters

- *items*: a sequence of options and values to print
- *spc*: leading distance between the start of the line and value

Keywords if provided, keywords will format `{key}` items in string

minimin.fabutil.**lsplit**(*s*)
Split a string based on newlines, strip non-empty lines.

Parameters

- *s*: string with newlines

Returns list of non-empty strings

minimin.fabutil.**missing**(*paths*, *files*)
Return file names in files for which there are no matching path in paths.

minimin.fabutil.**path**(*parent*, *child*)
Absolute path after appending a child file/directory to a parent path.

Parameters

- *parent*: parent directory
- *child*: child file or directory

Returns resulting path of the child appended to the parent

minimin.fabutil.**paths**(*parents*, *children*)
List paths of child files found in parent paths.

Parameters

- *parents*: list of parent directories
- *children*: list of child files or directories to look for in parents

Yields (child, path) for each child in children, or (child, None) if no path was found

minimin.fabutil.**printonly**(*s*)

Todo Kill this and use puts() since we decided to document --hide?

minimin.fabutil.**realpaths**(*parents*, *children*)

List of paths in parents that point to children.

Parameters

- *parents*: list of parent directories
- *children*: list of child files or directories to look for in parents

Yields full paths pointing to children

Exceptions

- *IOError*: if real path to a child doesn't exist in parents

5.7 minimin.fabqnd

Puppet assistance with configuration, service management and installation.

5.7.1 config class

Optional config class parameters:

- ${pathid}: final, rendered string for the file (if set, ignores existing template and ${pathid}_tmpl)
- ${pathid}_path: alternate file path
- ${pathid}_tmpl: Puppet template, appended to default template
- ${pathid}_{tmplvar}: for every {tmplvar} associated with a {pathid}, expose it within the class as ${pathid}_{tmplvar}

The config class checks defaults for:

- $path_{pathid}: (optional) ... if not found, ignores (class parameters ${pathid}* are ignored if the $path_{pathid} was not defined in the defaults)

Todo

nested directories prior to actual file declaration.

5.7.2 service class

Optional service class parameters:

- ${svcid}_state: Puppet service state (default 'running')
- ${svcid}_subs: list of paths to subscribe the service to

The service class checks defaults for:

- $svc_{svcid}_subs: (optional) list of paths the service subscribes to

5.7.3 `install` class

Optional `install` class parameters:

- `${pkgid}_state`: Puppet package state (default 'present')
- `${pkgid}_provider`: useful for manual installs and testing
- `${pkgid}_source`: useful for manual installs and testing

The `install` class checks defaults for:

- `$pkg_{pkgid}`: name of the package to manage
- `$pkg_{pkgid}_provider`: (optional) name of the Puppet package provider/manager, useful for automated updates setting state to 'latest'

  Note `${pkgid}_source` is only available as a class parameter to prevent default versioning.

5.7.4 `default` class

The default class maps generic variables used by the other classes to actual values:

> $path_{pathid} -> default path to file $pkg_{pkgid} -> package name as understood by the provider $pkg_{pkgid}_provider -> provider as listed in the Puppet package type

5.7.5 Notes

- Lighter bootstraps are available using `puppet-module`:

```
gem install puppet-module
puppet-module generate $USER-mymod
```

- `qndmod()` helps bootstrap a new Puppet-managed service by creating a quick and dirty ("`qnd`") submodule; manifests that provide a simple layout that is easy to customize but works in the meantime.
- There's a lot of if-then-else boilerplate in here. It'd be nice to use something like a native Puppet ternary one-liner - I didn't want to involve custom Ruby functions on the quick-n-dirty level. Unfortunately these patterns didn't work (and I haven't investigated further):

```
('x?y:z' and '(x && y)||z')
'x = y or z' is 'x = bool'
```

Todo

Of `pathid`, `svcid` and `pkgid` only `pathid` is treated as an abstract identifier. `svcid` and `pkgid` are used as the actual system service and package names. While no checks are made on any of these strings, we assume a real ID wouldn't use un-Puppet-friendly characters. However, we might not be able to say the same about a services and packages which might have dashes which "screw up qualified variable access" or invalid dots. Maybe add actual IDs for these later. It's more readable, less redundant, and less likely than differing configuration file paths across operating systems.

---

class `minimin.fabqnd.`**`Config`**(**args*)

Bases: `minimin.fabqnd.Manifest`

`add`(*tmplmod=None*, *user=None*, *mode=None*, *_id=None*, *path=None*, **tmplvars*)

Add parameters to the 'config' class template.

Parameters

- *tmplmod*: name of the module containing templates/*_id*
- *user*: default username of the file owner (also used for group)
- *mode*: default file mode
- *_id*: common ID/template name for the config file across platforms
- *path*: platform-specific path to the config file
- *tmplvars*: list of variables expected in templates/*_id*

Example INI option and value:

```
ntp.config.default = sitemod:ntp:0644:ntp_conf:/etc/ntp.conf:stratnum:servers
```

tmpl_class = 'class {mod}::{sub}::config (\n{params}\n)\n{{\n include {mod}::{sub}::default\n{files}\n}}'

tmpl_file = "\n $path_{pathid} = ${mod}::{sub}::default::path_{pathid}\n\n if $path_{pathid} {{\n\n if ${pathid}_path

tmpl_param = ' ${pathid} = undef,\n ${pathid}_path = undef,\n ${pathid}_tmpl = undef'

tmpl_tmplvar = ' ${pathid}_{tmplvar} = undef'

class minimin.fabqnd.**Install**(**args*)

Bases: minimin.fabqnd.Manifest

add(*_id=None*, *name=None*, *provider=None*, *source=None*)

Add parameters to the 'install' class template.

Parameters

- *_id*: common ID for the package across platforms
- *name*: platform-specific package name
- *provider*: optional package provider name

Example INI option and value:

```
ntp.install.redhat  = ntp:ntp:yum
ntp.install.solaris = ntp:CSWntp:pkgutil
```

The class declaration also accepts a `{pkgid}_source` parameter that specifies the source for the `pkgid` as understood by the `provider`:

```
class {'qnd::ntp::install': $ntp_source => 'http://filesvr/ntp.rpm'}
```

tmpl_class = 'class {mod}::{sub}::install (\n{params}\n)\n{{\n include {mod}::{sub}::default\n{packages}\n}}'

tmpl_package = "\n $pkg_{pkgid} = ${mod}::{sub}::default::pkg_{pkgid}\n $pkg_{pkgid}_provider = ${mod}::{sub}::

tmpl_param = " ${pkgid}_state = 'present',\n ${pkgid}_source = undef,\n ${pkgid}_provider = undef"

class minimin.fabqnd.**Manifest**(*mod*, *sub*, *manifest*, *osname*)

Bases: object

append(*seq*, *tmpl*, ***kw*)

Append a rendered template to an internal sequence.

init_class_macros(**args*)

Create lists for macros required by the class template.

render()

Render the completed class string.

render_default()

set_default(*var*, *val*)
Append (osname, value) to funky variable key in defaults dictionary.

tmpl_val = ' {osname} => {val},'

tmpl_var = ' ${var} = $operatingsystem ? {{\n{vals}\n }}'

class minimin.fabqnd.**Service**(**args*)
Bases: minimin.fabqnd.Manifest

add(*_id=None*, *name=None*)
Add parameters to the 'service' class template.

Parameters

- *_id*: common ID for the service across platforms
- *name*: platform-specific service name

Example INI option and value:

```
ntp.service.default = ntp:ntpd
```

The class declaration accepts a `{name}_subs` parameter that should specify one or more configuration files by `_id` to subscribe to:

```
class {'qnd::ntp::service': $ntp_subs => File['ntp_conf']}
```

tmpl_class = 'class {mod}::{sub}::service (\n{params}\n)\n{{\n include {mod}::{sub}::default\n{services}\n}}'

tmpl_param = " ${svcid}_state = 'running',\n ${svcid}_subs = undef"

tmpl_service = "\n $svc_{svcid} = ${mod}::{sub}::default::svc_{svcid}\n\n if $svc_{svcid} {{\n\n service {{ '{svcid}':\n

minimin.fabqnd.**esc_print**(*s*)
Print a line, escape dollar characters.

minimin.fabqnd.**inigen**(*cfg*)
Use an .INI config to generate information for `subgen()`.

Parameters

- *cfg*: path to .INI config file (must include `[qnd]` section with the appropriate options and values)

Yields `(sub, manifest, osname), val` for each option and value

minimin.fabqnd.**join**(*iterable*) → string
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

minimin.fabqnd.**qndgen**(*modhome*)
Create a quick-n-dirty script to generate submodules.

Parameters

- *modhome*: path to module directory (omit trailing separator)

Example: fab –hide=running,status qndgen:/path/to/mymod > mksubs.sh

minimin.fabqnd.**subgen**(*mod*)
Yield Puppet code sor submodule classes.

Parameters

- *mod*: main module name (submodules will be created as `mod::sub`)

Yields tuple of strings (sub, manifest, class) for all submodules

CHAPTER

SIX

IMPLEMENTATION DETAILS

6.1 Minimin's INI configuration

XML is too verbose and persnickety, JSON and YAML have an overhead that might frustrate Notepad users, and nested formats aren't very friendly to line parsers.

The main trade-off with the INI file is that a lack of formal nesting requires us to fudge nests (as dotted option names) in the interest of consolidating related options (commands, installations, roles). This adds some redundancy in option prefixes, but this also makes it easier to `grep` the options. Lack of indentation and nesting makes it easier to copy and paste option blocks.

6.2 Fabric

6.2.1 Keyword restrictions

Fabric (specifically, `fabric/main.py:516`) prevents keywords `host`, `hosts`, `role`, `roles`, `exclude_hosts` from being passed to their intended functions. This is why we're using `node` as a keyword all over the place.

6.2.2 Localhost output

I didn't find an obvious or nice way to control output per function/task via configuration, decoration, or context managers. Specifically:

- `running` messages precede function/task calls (preempting decorated or context-managed output),
- the status `Done.` follows everything (by default),
- `with_settings` and `settings` effects get rolled back after a function is finished (missing the final status message),
- avoiding the restrictions of decorators and context managers with `fabric.state.output` would require every function to be output-aware or risk quietly overriding the intended output of those that weren't.

An output-to-file option might be a good idea.

The end result is that functions intended to provide clean output just document the need to `--hide` undesired messages (see `fab -d {func}`):

```
fab --hide=running,status {func}
```

6.2.3 `admin_ssh_home` and `env.ssh_config_path`

`admin_ssh_home` is only here to support `sshkeygen()`, so that we know where to put our keys. We don't want to assume that the home of our `env.ssh_config_path` is always where we store our keys.

6.2.4 SSH errors

- `EOF during negotiation` is probably the result of a bad `Subsystem sftp` configuration in the server's `sshd_config`. This is why `sshput` (for single files, no recursion) exists as a pure-SSH alternative to file copying.
- `Too many authentication failures` (check your server logs) is probably the result of too many attempts to use invalid SSH keys. We might mitigate by setting `IdentitiesOnly yes` in our client `ssh_config`, but it looks like Fabric loops over our keys in `fabric/network.py:280`. A (practical) workaround is to limit the number of public keys used in the `ssh_config`. Help troubleshoot Paramiko/SSH connections in Fabric using `ssh.util.log_to_file('logfile', 10)`.
- `KeyError: 'user@host:port'` from `fabric/network.py` might be due to a misconfigured `ssh_config`. Specifically, I had a trailing space after a `Port` number in `ssh_config`. Running a test `ssh host -- uname -a` worked, but the trailing space resulted in a `KeyError` when running `fab -Hhost -- uname -a`.

6.3 Puppet

6.3.1 Why, why not ..?

Puppet feels like the best choice for system configuration because of its idempotency, built-in capabilities, number of supported platforms, online resources, and option for commercial support. Features of Puppet that weren't attractive were easily avoided: SSL client-server setup may be replaced by SSH updates and native `puppet apply` calls and file serving may be handled by bundling configs as templates.

Some alternatives:

- Chef, Kokki and cdist lean toward "programming as configuration" by exposing either more of the underlying programming language (Chef/Ruby, Kokki/Python) or the framework itself (Cdist).
- Synctool is easy to get started with, but felt better suited for homogeneous environments.
- Ansible and Salt are attractive alternatives. Both use a simpler INI or YAML syntax for most configuration. Ansible leverages SSH and Salt implements it's own high-performance communication using ZeroMQ.

6.3.2 Modules everywhere

The quick-n-dirty implementation uses modules for everything (no `/etc/puppet/puppet.conf` or `/etc/puppet/manifests`) to reduce the number of locations we have to think about and the amount of bootstrapping we have to do. We copy all the modules we need to a common directory, and provide the module path and site manifest to `puppet apply`.

6.3.3 Preferring templates

The quick-n-dirty implementation assumes that configuration files are managed in `templates/`, regardless of whether or not the file or class (`config.pp`) uses configurable parameters because:

1. templates/ and modules can provide automagic path resolution which works well for our push-based, copy/apply approach, and
2. between templates/ and files/, a 'template' implies something that is dynamic will have its contents modified.

6.3.4 Role dependency

Fabric's roles are named lists of hosts, and host specification via command line, configuration, or decoration is generally additive. I wanted group attributes (like required Puppet modules) bundled with the host list to consolidate group information.

6.3.5 Dirty boilerplate

The quick-n-dirty submodules have a lot of boilerplate due to the if-then logic that determines whether to use operating system variables defined in default.pp or class parameter values.

6.4 Manual installs

We avoid Puppet for manual installs (those which don't use a package manager) because:

- it's difficult to parse manifests for external purposes (this typically requires fetching the results from node), and
- including the installation recipes gains us uniformity only if we can assume Puppet itself is trivial to install (not likely on non-Linux nodes).

Since we must support manual, non-Puppet installs anyway (if only to bootstrap Puppet), we mimic idempotence using the commands we'd have to provide Puppet's exec helpers unless, onlyif and refresh. The trade-off is between the convenience of using Puppet as a one stop shop for installation and configuration and the flexibility of INI-style syntax.

If possible, installations and updates should be configured in Puppet using a *package manager* (yum, apt, etc.).

6.4.1 Version comparisons

Sometimes you might want to compare the version of some required program, perhaps in an installer's req option. GNU's awk (gawk) has a match(string, regex, array) function, but the presence of perl is probably a safer bet:

```
[ $(java -version 2>&1 | perl -n -e '/"(.*)_.*"/ && print $1') \> 1.6 ]
```

Note that Bash's string comparison does not support "greater than or equal to." Bash supports greater than, less than, equal or not equal. In this case we're taking advantage of the fact that a java -version has three components, and that 1.7.0 is greater than 1.7.

6.5 Things TODO

Todo

Figure out how the fabmin functions exists, support, and install should work together. exists functionality is duplicated in the other two, and the support directory in the [role] and [install] configuration sections

is (or should be?) the same. For the time being, `install` and `support` are treated as separate concerns that *just so happen* to work with each other if they share the same support directory. The result is that after the initial SSH access is set up and Puppet is installed, we should run `fab support`, `config`, and manual `install` commands in that order.

Todo

It might be nice to allow roles to determine their list of nodes using other roles' node lists.

Todo

Add command `sec.setuid` to look for setuid programs.

Todo

Add command `sec.mount` to check mounted filesystem options.

Todo

Add pointers in source to links/references in docs so we can see where we (supposedly) implemented a feature.

Todo

Note reasons for different error output - `fabric.utils.error` (bound to group of everything that can be dismissed with `warn_only`, `log.warn` (hints that you're straying from the path of righteousness), and built-in exceptions (I wouldn't work no matter what) vs. `fabric.utils.abort` (which should probably be used more than the exceptions).

Todo

Add dependencies and order to manual installs, and add a list of required installs per role: `role_name.install = pkg1 pkg2`.

Todo

Logging support.

Todo

A fallback mechanism for suffixes (so we don't have to repeat for `.f16` what would still work from `.f15`). Maybe a format for suffixes? Dashes might help but look ugly. How many alpha-only IDs do we need? f-Fedora, r-Red Hat, d-Debian, s-Solaris, o-OpenIndiana, h-HPUX, s-Slackware.. uh oh.

Todo

More documentation support and support for Doxygen specifically.

Todo

TaskJuggler support.

Todo

Pitz support. A project's repository should probably include its technical issues, internal todo list, and "lessons learned" (a project should not include the full history of public issues and discussions). If issue management isn't within the comfort zone of normal documentation, then a text-based tracker might be helpful.

Todo

Uniform output from `[exe]` commands to make it easier to use programatically, in an interactive session, or as input to a graphing utility.

Todo

Ansible looks like it might be an alternative to Fabric and Puppet (and it uses a simpler syntax by default).

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

PYTHON MODULE INDEX

m

INDEX

R

S

T

U

V

W