# bsdploy Documentation

*Release 2.4.0.dev0*

**Tom Lazar**

**Dec 22, 2017**

# Contents

BSDploy is a comprehensive tool to remotely **provision**, **configure** and **maintain** FreeBSD jail hosts and jails.

Its main design goal is to lower the barrier to *repeatable jail setups*.

Instead of performing updates on production hosts you are encouraged to update the *description* of your setup, test it against an identically configured staging scenario until it works as expected and then apply the updated configuration to production with confidence.

CHAPTER 1

# Main Features

- **provision** complete jail hosts from scratch
- **describe** one or more jail hosts and their jails in a canonical configuration
- **declarative configuration** – apply Ansible playbooks to hosts and jails
- **imperative maintenance** – run Fabric scripts against hosts and jails
- configure ZFS pools and filesystems with whole-disk-encryption
- **modular provisioning** with plugins for VirtualBox and Amazon EC2 and an architecture to support more.

CHAPTER 2

# How it works

BSDploy takes the shape of a commandline tool by the name of `ploy` which is installed on a so-called *control host* (typically your laptop or desktop machine) with which you then control one or more *target hosts*. The only two things installed on target hosts by BSDploy are Python and `ezjail` – everything else stays on the control host.

CHAPTER 3

# Example Session

Here's what an abbreviated bootstrapping session of a simple website inside a jail on an Amazon EC2 instance could look like:

```
# ploy start ec-instance
[...]
# ploy configure jailhost
[...]
# ploy start webserver
[...]
# ploy configure webserver
[...]
# ploy do webserver upload_website
```

CHAPTER 4

# Best of both worlds

Combining a declarative approach for setting up the initial state of a system with an imperative approach for providing maintenance operations on that state has significant advantages:

1. Since the imperative scripts have the luxury of running against a well-defined context, you can keep them short and concise without worrying about all those edge cases.
2. And since the playbooks needn't concern themselves with performing updates or other tasks you don't have to litter them with awkward states such as `restarted` or `updated` or – even worse – with non-states such as `shell` commands.

CHAPTER 5

# Under the hood

BSDploy's scope is quite ambitious, so naturally it does not attempt to do all of the work on its own. In fact, BSDPloy is just a fairly thin, slightly opinionated wrapper around existing excellent tools.

In addition to the above mentioned Ansible and Fabric, it uses ezjail on the host to manage the jails and on the client numerous members of the ployground family for pretty much everything else.

CHAPTER 6

# Full documentation

The full documentation is hosted at RTD.

CHAPTER 7

# Dive in

## 7.1 Client requirements

BSDploy and its dependencies are written in Python and thus should run on pretty much any platform, although it's currently only been tested on Mac OS X and FreeBSD.

## 7.2 Server requirements

**Warning:** BSDploy is intended for initial configuration of a jail host before any jails have been installed. While technically possible, BSDploy is not intended for managing existing systems with non-BSDploy jails. Running against hosts that have not been bootstrapped by BSDploy can result in loss of data.

A FreeBSD system that wants to be managed by BSDploy will

- need to have ezjail installed
- as well as Python
- must have SSH access enabled (either for root or with `sudo` configured).
- have ZFS support (BSDploy does not support running on non-ZFS filesystems)

Strictly speaking, BSDploy only needs Python for the initial configuration of the jailhost. If you chose to perform that step yourself or use a pre-existing host, you won't need Python on the host, just ezjail.

Normally, BSDploy will take care of these requirements for you during *bootstrapping* but in situations where this is not possible, manually providing the abovementioned requirements should allow you to *apply BSDploy's host configuration* anyway.

BSDploy supports FreeBSD >= 9.2, including 10.3.

## 7.3 Client Installation

Since BSDploy is still early in development, there is currently only a packaged version for FreeBSD but no others such as i.e. homebrew, aptitude etc.) available yet.

You can however install beta releases from PyPI or a bleeding edge development version from github.

### 7.3.1 Installing on FreeBSD

BSDploy is available from FreeBSD ports as `sysutils/bsdploy`, for details check it at FreshPorts.

### 7.3.2 Installing from PyPI

BSDploy and its dependencies are written in Python, so you can install them from the official Python Packaging Index (a.k.a. PyPI).

The short version:

```
virtualenv .
source bin/activate
pip install bsdploy
```

(1) BSDploy has specific requirements in regards to Fabric and ansible (meaning, their latest version will not neccessarily work with the latest version of BSDploy until the latter is adjusted) it is therefore strongly recommended to install BSDploy into its own virtualenv.

To do so, you will need Python and virtualenv installed on your system, i.e.

- on **Mac OS X** using `homebrew` you would install `brew install pyenv-virtualenv`.
- on **FreeBSD** using `pkg` you would `pkg install py27-virtualenv`

(2) To use the version installed inside this virtualenv it is suggested to 'source' the python interpreter. This will add the `bin` directory of the virtualenv (temporarily) to your `$PATH` so you can use the binaries installed inside it just as if they were installed globally. Note, that the default `activate` works for bash, if you're using `tcsh` (the default on FreeBSD you will have to `source bin/activate.csh`)

### 7.3.3 Installing from github

To follow along the latest version of BSDploy you need Python and virtualenv plus – obviously – `git`. Then:

```
git clone https://github.com/ployground/bsdploy.git
cd bsdploy
make
```

This will check out copies of BSDploy's immediate dependencies (`ploy` and friends) and create the ploy* executables inside `bin`. You can either add the `bin` directory to your path or symlink them into somewhere that's already on your path, but as described above, it is recommended to source the `virtualenv` to have a 'global' installation of BSDploy:

```
source bin/activate
```

When keeping your checkout up-to-date it is usually a good idea to update the `ploy` packages (located inside `src`), as well. The best way to do so is to use the provided `develop` command after updating the bsdploy repository itself like so:

```
git pull
bin/develop up -v
```

The `-v` flag will show any git messages that arise during the update.

## 7.4 Quickstart

This quickstart provides the shortest possible path from an empty project directory to a running jail inside a provisioned host.

It is designed to be followed along, with all required commands and configuration provided in a copy & paste friendly format.

It takes several shortcuts on its way but it should give you a good idea about how BSDploy works and leave you with a running instance that you can use as a stepping stone for further exploration.

The process consists of:

- creating a VirtualBox based host (**provisioning**)
- then installing FreeBSD onto it (**bootstrapping**)
- then **configuring** the vanilla installation to our needs
- and finally creating a 'hello world' jail inside of it.

**Note:** Before you begin with this tutorial, make sure you have *installed bsdploy*.

### 7.4.1 Using VirtualBox

To give us the luxury of running against a well-defined context, this quickstart uses VirtualBox, a free, open source PC virtualization platform. If you don't have it installed on your system, head over to their downloads section and install it for your platform. We'll wait! If you can't be bothered, following along anyway should still be useful, though.

Since VirtualBox support is optional and BSDploy is fairly modular, you will need to install `ploy_virtualbox` to follow this quickstart like so:

```
% pip install ploy_virtualbox
```

### 7.4.2 Initializing the project environment

BSDploy has the notion of an environment, which is just fancy talk for a directory with specific conventions. Let's create one:

```
% mkdir ploy-quickstart
% cd ploy-quickstart
```

### 7.4.3 Configuring the virtual machine

The main configuration file is named `ploy.conf` and lives inside a top-level directory named `etc` by default:

```
% mkdir etc
```

Inside it create a file named `ploy.conf` with the following contents:

```
[vb-instance:ploy-demo]
vm-nic2 = nat
vm-natpf2 = ssh,tcp,,44003,,22
storage =
    --medium vb-disk:defaultdisk
    --type dvddrive --medium http://mfsbsd.vx.sk/files/iso/10/amd64/mfsbsd-se-10.3-
↪RELEASE-amd64.iso --medium_sha1 564758b0dfebcabfa407491c9b7c4b6a09d9603e
```

This creates a VirtualBox instance named `ploy-demo`. By default BSDploy provides it with a so-called "host only interface" but since that cannot be used to connect to the internet we explicitly configure a second one using NAT (mfsBSD will configure it via DHCP) and in addtion we create a port forwarding from `localhost` port `44003` to port `22` on the box - in essence allowing us to SSH into it via localhost.

Next, we assign a virtual disk named `defaultdisk` onto which we will install the OS. This special disk is created automatically by BSDploy if it doesn't exist yet (it's sparse by default, so it won't take up much space on your disk).

Finally, we configure a virtual optical drive to boot from the official mfsBSD 'special edition' installation image. By providing a download URL and checksum, BSDploy will automatically download it for us.

Now we can start it up:

```
% ploy start ploy-demo
```

This should download the mfsBSD image, fire up VirtualBox and boot our VM into mfsBSD.

### 7.4.4 Bootstrapping the host

To bootstrap the jailhost, we need to define it first. This is done with an `ez-master` entry in `ploy.conf`. So add this:

```
[ez-master:jailhost]
instance = ploy-demo
```

This creates an ezjail jailhost (`ez-master`) named `jailhost` and tells BSDploy that it lives / should live inside the provisioning instance named `ploy-demo` (our freshly created virtual machine).

But since none of this has happened yet, we need to tell BSDploy to make it so, like this:

```
% ploy bootstrap
```

This will ask you to provide a SSH public key (answer `y` if you have one in `~/.ssh/identity.pub`).

Next it will give you one last chance to abort before it commences to wipe the target drive, so answer `y` again.

To make sure that everything has worked so far, let's take a look at the host by logging into it via SSH. `bsdploy` provides a command for that, too:

```
% ploy ssh jailhost
FreeBSD 10.3-RELEASE (GENERIC) #0 r297264: Fri Mar 25 02:10:02 UTC 2016

Welcome to FreeBSD!
[...]
```

Let's take a quick look around. First, what packages have been installed?:

```
root@jailhost:~ # pkg info
gettext-runtime-0.19.3          GNU gettext runtime libraries and programs
indexinfo-0.2.2                 Utility to regenerate the GNU info page index
libffi-3.0.13_3                 Foreign Function Interface
pkg-1.4.3                       Package manager
python27-2.7.9                  Interpreted object-oriented programming language
```

Next, what's the ZFS scenario?:

```
root@jailhost:~ # zpool list
NAME     SIZE  ALLOC   FREE   FRAG  EXPANDSZ    CAP  DEDUP  HEALTH  ALTROOT
system  19.9G   931M  19.0G     2%         -     4%  1.00x  ONLINE  -
root@jailhost:~ # zfs list
NAME              USED  AVAIL  REFER  MOUNTPOINT
system            931M  18.3G    19K  none
system/root       931M  18.3G   876M  /
system/root/tmp    21K  18.3G    21K  /tmp
system/root/var  54.2M  18.3G  54.2M  /var
root@jailhost:~ #
```

A few things to note:

- `pkg` is installed and configured
- `python` has been installed
- there is one zpool which contains the system
- not much else

In other words, there's still work to do, so let's log out and continue...

### 7.4.5 Configuring the host

Now we can configure the vanilla installation. This step is performed internally using ansible playbooks, which are divided into different so-called *roles*. For the tutorial we will need the DHCP role (since Virtualbox provides DHCP to the VM) and the main jailhost role so add the following lines to the jailhost configuration in `ploy.conf` to make it look like so:

```
[ez-master:jailhost]
instance = ploy-demo
roles =
    dhcp_host
    jails_host
```

With this information, BSDploy can get to work:

```
% ploy configure jailhost
```

Let's log in once more and take another look:

```
% ploy ssh jailhost
[...]
```

Package-wise nothing much has changed – only `ezjail` has been installed:

```
root@jailhost:~ # pkg info
ezjail-3.4.1                    Framework to easily create, manipulate, and run␣
↪FreeBSD jails
gettext-runtime-0.19.3          GNU gettext runtime libraries and programs
indexinfo-0.2.2                 Utility to regenerate the GNU info page index
libffi-3.0.13_3                 Foreign Function Interface
pkg-1.4.3                       Package manager
python27-2.7.9                  Interpreted object-oriented programming language
root@jailhost:~ #
```

There is now a second zpool called `tank` and `ezjail` has been configured to use it:

```
root@jailhost:~ # zpool list
NAME     SIZE  ALLOC   FREE   FRAG  EXPANDSZ    CAP  DEDUP  HEALTH  ALTROOT
system  19.9G   934M  19.0G     2%         -     4%  1.00x  ONLINE  -
tank    75.5G   444M  75.1G      -         -     0%  1.00x  ONLINE  -
root@jailhost:~ # zfs list
NAME                  USED  AVAIL  REFER  MOUNTPOINT
system                933M  18.3G    19K  none
system/root           933M  18.3G   877M  /
system/root/tmp        21K  18.3G    21K  /tmp
system/root/var      56.6M  18.3G  56.6M  /var
tank                  443M  72.7G   144K  none
tank/jails            443M  72.7G  10.1M  /usr/jails
tank/jails/basejail   426M  72.7G   426M  /usr/jails/basejail
tank/jails/newjail   6.37M  72.7G  6.37M  /usr/jails/newjail
root@jailhost:~ #
```

But there aren't any jails configured yet:

```
root@jailhost:~ # ezjail-admin list
STA JID  IP              Hostname                       Root Directory
--- ---- --------------- ------------------------------ ------------------------
root@jailhost:~ #
```

Let's change that...

### 7.4.6 Creating a jail

Add the following lines to `etc/ploy.conf`:

```
[ez-instance:demo_jail]
ip = 10.0.0.1
```

and start the jail like so:

```
% ploy start demo_jail
```

Let's check on it first, by logging into the host:

```
ploy ssh jailhost
root@jailhost:~ # ezjail-admin list
STA JID  IP              Hostname                       Root Directory
--- ---- --------------- ------------------------------ ------------------------
ZR  1    10.0.0.1        demo_jail                      /usr/jails/demo_jail
```

Ok, we have a running jail, listening on a private IP – how do we communicate with it? Basically, there are two options (besides giving it a public IP): either port forwarding from the host or using a SSH proxy command.

Rather conveniently ploy_ezjail has defaults for the latter.

Log out from the jailhost and run this:

```
# ploy ssh demo_jail
FreeBSD 10.3-RELEASE (GENERIC) #0 r297264: Fri Mar 25 02:10:02 UTC 2016

Gehe nicht über Los.
root@demo_jail:~ #
```

and there you are, inside the jail.

But frankly, that's not very interesting. As a final step of this introduction, let's configure it to act as a simple webserver using an ansible playbook.

### 7.4.7 Configuring a jail

Like with the jailhost, we could assign roles to our demo jail, but another way is to create a playbook with the same name. If such a playbook exists, BSDploy will use that when you call `configure`. So, create a top-level file named `jailhost-demo_jail.yml` with the following content:

```
---
- hosts: jailhost-demo_jail
  tasks:
    - name: install nginx
      pkgng: name=nginx state=present
    - name: Setup nginx to start immediately and on boot
      service: name=nginx enabled=yes state=started
```

and apply it:

```
% ploy configure demo_jail
```

Ok, now we have a jail with a webserver running inside of it. How do we access it? Right, *port forwarding*...

#### Port forwarding

Port forwarding from the host to jails is implemented using `ipnat` and BSDploy offers explicit support for configuring it.

To do so, make a folder named `host_vars`:

```
% mkdir host_vars
```

and create the file `jailhost.yml` in it with the following content:

```
pf_nat_rules:
    - "rdr on em0 proto tcp from any to em0 port 80 -> {{ hostvars['jailhost-demo_jail
↪']['ploy_ip'] }} port 80"
```

To activate the rules, re-apply the jail host configuration with just the `pf-conf` tag. Ansible will figure out, that it needs to update them (and only them) and then restart the network. However, in practice running the entire configuration can take quite some time, so if you already know you only want to update some specific sub set of tasks you can pass in one or more tags. In this case for updating the ipnat rules:

```
% ploy configure jailhost -t pf-conf
```

Since the demo is running inside a host that got its IP address via DHCP we will need to find that out before we can access it in the browser.

To find out, which one was assigned run `ifconfig` like so:

```
% ploy ssh jailhost 'ifconfig em0'
em0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
    options=9b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_HWCSUM>
    ether 08:00:27:87:2e:40
    inet 192.168.56.108 netmask 0xffffff00 broadcast 192.168.56.255
    nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
    media: Ethernet autoselect (1000baseT <full-duplex>)
    status: active
```

Visit the IP in your browser and you should be greeted with the default page of `nginx`.

CHAPTER 8

# Tutorial

A more in-depth tutorial than the quickstart.

## 8.1 Tutorial

### 8.1.1 Overview

This tutorial will focus on setting up a host as an all-purpose webserver, with a dedicated jail for handling the incoming traffic and redirecting it to the appropriate jails.

In it, we will also demonstrate how to setup a staging/production environment. We will start with developing the host on a virtual instance and at the end will replay the final setup against a 'production' host.

We will

- configure a webserver jail (nginx) with a simple static overview site which contains links to the various services offered inside the jails. This step will also demonstrate the integration of Fabric scripts and configuring FreeBSD services and how to mount ZFS filesystems into jails.
- install an application that is more or less usable out-of-the-box and requires minimal configuration (transmission, a bit torrent client with a web frontend)
- create a 'production' configuration and apply the setup to it

### 8.1.2 Requirements

To follow along this tutorial, you will need to have *installed bsdploy* and a running jailhost. The easiest and recommended way to achieve this is to follow along the *quickstart*.

## 8.2 Webserver

The idea is that the webserver lives in its own jail and receives all HTTP traffic and then redirects the requests as required to the individual jails.

The webserver jail itself will host a simple website that lists and links the available services.

The website will be in a dedicated ZFS that will be mounted into the jail, so let's start with creating this.

In `etc/ploy.conf` from the quickstart you should have the following ezjail master:

```
[ez-master:jailhost]
instance = ploy-demo
roles =
    dhcp_host
    jails_host
```

### 8.2.1 ZFS mounting

To set up the ZFS layout we will replace the inline roles with a dedicated playbook, so let's delete the `roles =` entry and create a top-level file names `jailhost.yml` with the following contents:

```
---
- hosts: jailhost
  user: root
  roles:
    - dhcp_host
    - jails_host
```

Once we have a playbook in place, it becomes easy to add custom tasks:

```
tasks:
  - name: ensure ZFS file systems are in place
    zfs: name={{ item }} state=present mountpoint=/{{ item }}
    with_items:
    - tank/htdocs
    tags: zfs-layout
```

To apply the playbook it's easiest to call the `configure` command again. While Ansible is smart enough to only apply those parts that actually need to be applied again that process can become quite slow, as playbooks grow in size. So here, we tag the new task with `zfs-layout`, so we can call it explicitly:

```
ploy configure jailhost -t zfs-layout
```

**Note:** You should see an `INFO` entry in the output that tells you which playbook the `configure` command has used.

### 8.2.2 Exercise

In the quickstart, we've created a demo jail which conveniently already contains a webserver.

Repurpose it to create a jail named `webserver` which has nginx installed into it.

Do this by first terminating the demo jail, *then* renaming it, otherwise we will have an 'orphaned' instance which would interfere with the new jail:

```
ploy terminate demo_jail
```

Now we can mount the website ZFS into it like so:

```
[ez-instance:webserver]
master = jailhost
ip = 10.0.0.2
mounts =
    src=/tank/htdocs dst=/usr/local/www/data ro=true
```

**Note:** Mounting filesystems into jails gives us the ability to mount them read-only, like we do in this case.

Let's start the new jail:

```
ploy start webserver
```

### 8.2.3 Instance playbooks

Note that the webserver jail doesn't have a role assigned. Roles are useful for more complex scenarios that are being re-used. For smaller tasks it's often easier to simply create a one-off playbook for a particular host.

To associate a playbook with an instance you need to create a top-level YAML file with the same name as the instance – like we did above for the jailhost. For jail instances, this name contains both the name of the master instance *and* the name of the jail, so let's create a top-level file named `jailhost-webserver.yml` with the following contents:

```
---
- hosts: jailhost-webserver
  tasks:
    - name: install nginx
      pkgng: name=nginx state=present
    - name: enable nginx at startup time
      lineinfile: dest=/etc/rc.conf state=present line='nginx_enable=YES' create=yes
    - name: make sure nginx is running or reloaded
      service: name=nginx state=restarted
```

In the above playbook we demonstrate

- how to install a package with the `pkgng` module from ansible
- enable a service in `rc.conf` using the `lineinfile` command
- ensure a service is running with the `service` command

Let's give that a spin:

```
ploy configure webserver
```

**Note:** If the name of a jail is only used in a single master instance, `ploy` allows us to address it without stating the full name on the command line for convenience. IOW the above command is an alias to `ploy configure jailhost-webserver`.

### 8.2.4 "Publishing" jails

Eventhough the webserver is now running, we cannot reach it from the outside, we first need to explicitly enable access. While there are several possibilites to achieve this, we will use `ipnat`, just like in the quickstart.

So, create or edit `host_vars/jailhost.yml` to look like so:

```
pf_nat_rules:
    - "rdr on {{ ansible_default_ipv4.interface }} proto tcp from any to {{ ansible_
↪default_ipv4.interface }} port 80 -> {{ hostvars['jailhost-webserver']['ploy_ip'] }}
↪ port 80"
```

To activate the rules, re-apply the jail host configuration for just the `pf-conf` tag:

```
ploy configure jailhost -t pf-conf
```

You should now be able to access the default nginx website at the `http://192.168.56.100` address.

### 8.2.5 Use defaults

Currently the webserver serves the default site located at `/usr/local/www/nginx` which is a symbolic link to `nginx-dist`.

Now, to switch it the website located inside the ZFS filesystem we could either change the nginx configuration to point to it but in practice it can be a good idea to use default settings as much as possible and instead make the environment match the default. *Every custom configuration file you can avoid is a potential win.*

In this particular case, let's mount the website into the default location. First we need to remove the symbolic link that has been created by the nginx start up. Since this is truly a one-time operation (if we re-run the modified playbook against a fresh instance the symbolic link would not be created and wouldn't need to be removed) we can use ploy's ability to execute ssh commands like so:

```
ploy ssh webserver "rm /usr/local/www/nginx"
```

Now we can change the mountpoint in `ploy.conf`:

```
[ez-instance:webserver]
master = jailhost
ip = 10.0.0.2
mounts =
    src=/tank/htdocs dst=/usr/local/www/nginx ro=true
```

Unfortunately, currently the only way to re-mount is to stop and start the jail in question, so let's do that:

```
ploy stop webserver
ploy start webserver
```

Reload the website in your browser: you should now receive a `Forbidden` instead of the default site because the website directory is still empty.

### 8.2.6 Fabric integration

So far we've used ansible to configure the host and the jail. Its declarative approach is perfect for this. But what about maintenance tasks such as updating the contents of a website? Such tasks are a more natural fit for an *imperative* approach and `ploy_fabric` gives us a neat way of doing this.

Let's create a top-level file named `fabfile.py` with the following contents:

```
from fabric import api as fab

def upload_website():
        fab.put('htdocs/*', '/usr/jails/webserver/usr/local/www/nginx/')
```

Since the webserver jail only has read-access, we need to upload the website via the host (for now), so let's associate the fabric file with the host by making its entry in `ploy.conf` look like so:

```
[ez-master:jailhost]
instance = ploy-demo
fabfile = ../fabfile.py
```

Create a simple index page:

```
mkdir htdocs
echo "Hello Berlin" >> htdocs/index.html
```

Then upload it:

```
ploy do jailhost upload_website
```

and reload the website.

### 8.2.7 Exercise One

Requiring write-access to the jail host in order to update the website is surely not very clever.

Your task is to create a jail named `website_edit` that contains a writeable mount of the website and which uses a modified version of the fabric script from above to update the contents.

### 8.2.8 Exercise Two

Put the path to the website on the host into a ansible variable defined in ploy.conf and make the fabric script reference it instead of hard coding it.

You can access variables defined in ansible and `ploy.conf` in Fabric via its `env` like so:

```
ansible_vars = fab.env.instance.get_ansible_variables()
```

The result is a dictionary populated with variables from `group_vars`, `host_vars` and from within `ploy.conf`. However, it does *not* contain any of the Ansible facts. For details check ploy_fabric's documentation

## 8.3 Transmission

This part of the tutorial will demonstrate how to install and configure an existing web application.

### 8.3.1 Using Ansible Roles

Unlike with the webserver from the previous example we *will* create a custom configuration, so instead of littering our top level directory with yet more playbooks and templates we will configure this instance using a role.

Let's first create the required structure:

```
mkdir -p roles/transmission/tasks
mkdir -p roles/transmission/templates
mkdir -p roles/transmission/handlers
```

Populate them with a settings template in `roles/transmission/templates/settings.json`:

```
{
    "alt-speed-up": 50,
    "alt-speed-down": 200,
    "speed-limit-down": 5000,
    "speed-limit-down-enabled": true,
    "speed-limit-up": 100,
    "speed-limit-up-enabled": true,
    "start-added-torrents": true,
    "trash-original-torrent-files": true,
    "watch-dir": "{{download_dir}}",
    "watch-dir-enabled": true,
    "rpc-whitelist":  "127.0.0.*,10.0.*.*",
    "ratio-limit": 1.25,
    "ratio-limit-enabled": true
}
```

And in `roles/transmission/handlers/main.yml`:

```
---
- name: restart transmission
  service: name=transmission state=restarted
```

And finally in `roles/transmission/tasks/main.yml`:

```
- name: Ensure helper packages are installed
  pkgng: name={{ item }} state=present
  with_items:
  - transmission-daemon
  - transmission-web
- name: Setup transmission to start on boot
  service: name=transmission enabled=yes
- name: Configure transmission
  template: src=settings.json dest=/usr/local/etc/transmission/home/settings.json␣
↪backup=yes owner=transmission
  notify:
    - restart transmission
```

The above tasks should look pretty familiar by now:

- install the required packages (this time it's more than one and we demonstrate the `with_items` method)
- enable it in `rc.conf`
- Finally, as a new technique we upload a settings file as a template and...
- ... use ansible's *handlers* to make sure that the service is reloaded every time we change its settings.

### 8.3.2 Exercise One

Publish the transmission daemon's web UI at `http://192.168.56.100/transmission`.

**Note:** Proxying to transmission can be a bit finicky as it requires certain CRSF protection headers, so here's a small spoiler/hint.

This is the required nginx configuration to proxy to transmission:

```
location /transmission {
    proxy_http_version  1.1;
    proxy_set_header    Connection "";
    proxy_pass_header   X-Transmission-Session-Id;
    proxy_pass          http://transmissionweb;
    proxy_redirect      off;
    proxy_buffering     off;
    proxy_set_header    Host            $host;
    proxy_set_header    X-Real-IP       $remote_addr;
}
```

### 8.3.3 Exercise Two

Publish the downloads directory via nginx so users can download finished torrents from `http://192.168.56.100/downloads`.

Do this by configuring an additional jail that has read-only access to the download directory and publishes using its own nginx which is then targetted by the webserver jail.

## 8.4 Staging

So far we have developed our setup against a virtual machine which allowed us to experiment safely. Now it's time to deploy the setup in production.

Principally, there are two possible approaches to this with ploy:

- either define multiple hosts in one configuration or
- use two separate configurations for each scenario

In practice it's more practical to split the configuration, as it allows to re-use instances more easily.

As a rule-of-thumb, if you ever need to configure two scenarios simultenously or either of the hosts need to communicate with each other, you should use a single configuration with multiple hosts (i.e. if you're configuring a cluster) but if you're setting up mutually exclusive, isolated scenarios such as staging vs. production you should use two different configurations.

Another benefit of using two configurations is that you need to explicitly reference the non-default configuration which minimizes accidental modification of i.e. production hosts.

In this tutorial we will use a second VirtualBox instances to simulate a production environment but in actual projects you would more likely define either plain instances or EC2 instances.

To create a 'production' environment, create an additional configuration file `etc/production.conf` with the following contents:

```
[global]
extends = ploy.conf

[vb-instance:demo-production]
```

```
vm-nic2 = nat
vm-natpf2 = ssh,tcp,,44004,,22
storage =
    --type dvddrive --medium ../downloads/mfsbsd-se-9.2-RELEASE-amd64.iso
    --medium vb-disk:boot

[ez-master:jailhost]
instance = demo-production
```

Now we can start up the 'production' provider and run through the identical bootstrapping process as in the quickstart, except that we explicitly reference the production configuration:

```
ploy -c etc/production.conf start demo-production
```

However, before we can bootstrap we need to import the `bootstrap` into our custom fabric file – this is because bootstrapping is internally implemented as a fabric task.

Add the following import statement to your fabfile:

```
from bsdploy.fabfile_mfsbsd import bootstrap
```

Now we can follow through:

```
ploy -c etc/production.conf bootstrap
ploy -c etc/production.conf configure jailhost
```

CHAPTER 9

# Setup

How to setup a host from scratch or make an existing one ready for BSDploy:

## 9.1 Overview

The life cycle of a system managed by BSDploy is divided into three distinct segments:

1. provisioning
2. bootstrapping
3. configuration

Specifically:

- At the end of the provisioning process we have a system that has booted into an installer (usually mfsBSD) which is ready for bootstrapping
- At the end of the bootstrapping process we have installed and booted into a vanilla FreeBSD system with just the essential requirements to complete the actual configuration
- At the end of the configuration process we have a FreeBSD system with a pre-configured `ezjail` setup which can be managed by BSDploy.

Conceptually, the provider of a jailhost is a separate entity from the jailhost itself, i.e. in a VirtualBox based setup you could have the following:

```
[vb-instance:ploy-demo]
vm-ostype = FreeBSD_64
[...]

[ez-master:jailhost]
instance = ploy-demo
[...]

[ez-instance:webserver]
```

```
master = jailhost
[...]
```

Here we define a VirtualBox instance (`vb-instance`) named `ploy-demo` and a so-called master named `jailhost` which in turn contains a jail instance named `webserver`.

This approach allows us to hide the specific details of a provider and also to replace it with another one, i.e. *in a staging scenario* where a production provider such as a 'real' server or EC2 instance is replaced with a VirtualBox.

In the following we will document the entire provisioning process for each supported provider separately (eventhough there is a large amount of overlap), so you can simply pick the one that suits your setup and then continue with the configuration step which is then the same for all providers.

In a nutshell, though, given the previous example setup, what you need to do is this:

```
ploy start ploy-demo
ploy bootstrap jailhost
ploy configure jailhost
ploy start webserver
```

### 9.1.1 Project setup and directory structure

`ploy` and thus by extension also BSDploy have a "project based" approach, meaning that all configuration, playbooks, assets etc. for one given use case (a.k.a. "project") are contained in a single directory, as opposed to having a central configuration such as `/usr/local/etc/ploy.conf`.

The minimal structure of such a directory is to contain a subdirectory named `etc` in which the main configuration file `ploy.conf` is located.

Since BSDploy treats this directory technically as an *Ansible directory* you typically would also have additional top-level directories such as `roles`, `host_vars` etc.

Another (entirely optional) convention is to have a top-level directory named `downloads` where assets such as installer images or package archives are placed.

This approach was also chosen because most of the times project directories are version controlled and for example `downloads/` can then be safely ignored, because all of its contents are a) large, binary files and b) easily replaceable whereas `etc/` often contains sensitive, project specific data such as private keys, certificates etc. and may be even part of a different repository altogether.

### 9.1.2 Next step: Provisioning

Basically, unless you want to use one of the specific provisioners such as EC2 or VirtualBox, just use a plain instance:

#### Provisioning plain instances

The most simple provider simply assumes an already existing host. Here any configuration is purely descriptive. Unlike with the other providers we tell BSDploy how things are and it doesn't do anything about it.

For example:

```
[plain-instance:ploy-demo]
host = 10.0.1.2
```

At the very least you will need to provide a `host` (IP address or hostname).

Additionally, you can provide a non-default SSH `port` and a `user` to connect with (default is `root`).

### Local hardware

The most common scenario for using a plain instance is physical hardware that you have access to and can boot from a custom installer medium.

### Download installer image

First, you need to download the installer image and copy it onto a suitable medium, i.e. an USB stick.

As mentioned in the quickstart, BSDploy uses mfsBSD which is basically the official FreeBSD installer plus pre-configured SSH access. Also, it provides a small cross-platform helper for downloading assets via HTTP which also checks the integrity of the downloaded file:

```
mkdir downloads
ploy-download http://mfsbsd.vx.sk/files/images/9/amd64/mfsbsd-se-9.2-RELEASE-amd64.
↪img 9f354d30fe5859106c1cae9c334ea40852cb24aa downloads/
```

### Creating a bootable USB medium (Mac OSX)

For the time being we only provide instructions for Mac OS X, sorry! If you run Linux you probably already know how to do this, anyway :-)

- Run `diskutil list` to see which drives are currently in your system.
- insert your medium
- re-run `diskutil list` and notice which number it has been assigned (N)
- run `diskutil unmountDisk /dev/diskN`
- run `` `sudo dd if=downloads/mfsbsd-se-9.2-RELEASE-amd64.img of=/dev/diskN bs=1m ``
- run `diskutil unmountDisk /dev/diskN`
- now you can remove the stick and boot the target host from it

### Booting into mfsBSD

Insert the USB stick into the *target host* and boot from it. Log in as `root` using the pre-configured password `mfsroot`. Either note the name of the ethernet interface and the IP address it has been given by running `ifconfig` and (temporarily) set it in `ploy.conf` or configure them one-time to match your expectations.

### One-Time manual network configuration

BSDploy needs to access the installer via SSH and it in turn will need to download assets from the internet during installation, so we need to configure the interface, the gateway, DNS and sshd.

In the above mentioned example that could be:

```
ifconfig em0 netmask 255.255.255.0
route add default 10.0.1.1
```

Since usually, the router also performs as DNS you would edit `/etc/resolv.conf` to look like so:

```
nameserver 10.0.1.1
```

Finally restart sshd:

```
service sshd restart
```

To verify that all is ready, ssh into the host as user `root` with the password `mfsroot`. Once that works, log out again and you are ready to continue with *Bootstrapping*.

### Hetzner

The German ISP Hetzner provides dedicated servers with FreeBSD support. In a nutshell, boot the machine into their so-called *Rescue System* using their robot and choose *FreeBSD* as OS. The machine will boot into a modified version of mfsBSD.

The web UI will then provide you with a one-time root password – make sure it works by SSHing into the host as `root` and you are ready for continuing with *Bootstrapping*.

### vmWare

Since BSDploy (currently) doesn't support automated provisioning of vmWare instances (like it does for VirtualBox) you will need to manually create a vmWare instance and then follow the steps above for it, except that instead of downloading the image referenced there you need one specifically for booting into a virtual machine, IOW download like so:

```
mkdir downloads
ploy-download http://mfsbsd.vx.sk/files/iso/9/amd64/mfsbsd-se-9.2-RELEASE-amd64.iso␣
↪4ef70dfd7b5255e36f2f7e1a5292c7a05019c8ce downloads/
```

Then create a new virtual machine, set the above image as boot device and continue with *Bootstrapping*.

### Provisioning VirtualBox instances

BSDploy provides automated provisioning of VirtualBox instances via the ploy_virtualbox plugin.

**Note:** The plugin is not installed by default when installing BSDploy, so you need to install it additionally like so `pip install ploy_virtualbox`.

Unlike with *plain instances* the configuration doesn't just describe existing instances but is used to create them. Consider the following entry in `ploy.conf`:

```
[vb-instance:ploy-demo]
vm-nic2 = nat
vm-natpf1 = ssh,tcp,,44003,,22
storage =
    --medium vb-disk:defaultdisk
    --type dvddrive --medium http://mfsbsd.vx.sk/files/iso/10/amd64/mfsbsd-se-10.3-
↪RELEASE-amd64.iso --medium_sha1 564758b0dfebcabfa407491c9b7c4b6a09d9603e
```

VirtualBox instances are configured using the `vb-instance` prefix and you can set parameters of the virtual machine by prefixing them with `vm-`. For additional details on which parameters are available and what they mean, refer to the plugin's documentation and the documentation of the VirtualBox commandline tool VBoxManage, in particualar for VBoxManage createvm and VBoxManage modifyvm.

Having said that, BSDploy provides a number of convenience defaults for each instance, so in most cases you won't need much more than in the above example.

### Default hostonly network

Unless you configure otherwise, BSDploy will tell VirtualBox to

- create a host-only network interface named `vboxnet0`
- assign the first network interface to that
- create a DHCP server for the address range `192.168.56.100-254`

This means that a) during bootstrap the VM will receive a DHCP address from that range but more importantly b) you are free to assign your own static IPs from the range *below* (i.e. `192.168.56.10`) because the existence of the VirtualBox DHCP server will ensure that that IP is reachable from the host system. This allows you to assign known, good static IP addresses to all of your VirtualBox instances.

### Default disk setup

As you can see in the example above, there is a reference to a disk named `defaultdisk` in the `storage` parameter of the `vb-instance` entry. If you reference a disk of that name, BSDploy will automatically provision a virtual sparse disk of 100Gb size. In practice it's often best to leave that assignment in place (it's where the OS will be installed onto during bootstrap) and instead configure additional disks for data storage.

### Boot images

Also note, that we reference a mfsBSD boot image above and assign it to the optical drive. By providing an external URL with a checksum, `ploy_virtualbox` will download that image for us (by default into `~/.ploy/downloads/`) and connect it to the instance.

### First Startup

Unlike `VBoxManage` BSDploy does not provide an explicit *create* command, instead just start the instance and if it doesn't exist already, BSDploy will create it for you on-demand:

```
ploy start ploy-demo
```

Since the network interface is configured via DHCP, we cannot know under which IP the VM will be available. Instead the above snippet configures portforwarding, so regardless of the IP it gets via DHCP, we will access the VM via SSH using the host `localhost` and (in the example) port `44003`. Adjust these values to your needs and use them during *Bootstrapping*.

**Note:** In addtion to starting a VM you can also use the `stop` and `terminate` commands..

### Provisioning Amazon EC2 instances

BSDploy provides automated provisioning of Amazon EC2 instances via the ploy_ec2 plugin.

**Note:** The plugin is not installed by default when installing BSDploy, so you need to install it additionally like so `pip install ploy_ec2`.

Like with *Virtualbox instances* the configuration doesn't just describe existing instances but is used to create them.

The first step is always to tell ploy where to find your AWS credentials in `ploy.conf`:

```
[ec2-master:default]
access-key-id = ~/.aws/foo.id
secret-access-key = ~/.aws/foo.key
```

Then, to define EC2 instances use the `ec2-` prefix, for example:

```
1  [ec2-instance:production-backend]
2  region = eu-west-1
3  placement = eu-west-1a
4  keypair = xxx
5  ip = xxx.xxx.xxx.xxx
6  instance_type = c3.xlarge
7  securitygroups = production
8  image = ami-c847b2bf
9  user = root
10 startup_script = ../startup_script
```

Let's go through this briefly, the full details are available at the ploy_ec2 documentation:

- `2-3` here you set the region and placement where the instance shall reside
- `4` you can optionally provide a keypair (that you must create or upload to EC2 beforehand). If you do so, the key will be used to grant you access to the newly created machine. See the section below regarding the `startup_script`.
- `5` if you have an Elastic IP you can specify it here and `ploy` will tell EC2 to assign it to the instance (if it's available)
- `6` check the EC2 pricing overview for a description of the available instance types.
- `7` every EC2 instance needs to belong to a so-called security group. You can either reference an existing one or create one like so:

  ```
  [ec2-securitygroup:production]
  connections =
      tcp     22      22      0.0.0.0/0
      tcp     80      80      0.0.0.0/0
      tcp    443     443      0.0.0.0/0
  ```

- `8` For FreeBSD the currently best option is to use Colin Percival's excellent daemonology AMIs for FreeBSD. Simply pick the ID best suited for your hardware and region from the list and you're good to go!
- `9` The default user for which daemonology's startup script configures SSH access (using the given `keypair`) is named `ec2-user` but BSDploy's playbooks all assume `root`, so we explicitly configure this here. Note, that this means that we *must* change the `ec-user` name (this happens in our own `startup_script`, see below).

- `10 ploy_ec2` allows us to provide a local startup script which it will upload for us using Amazon's instance metadata mechanism. Here we reference it relative to the location of `ploy.conf`. The following example provides minimal versions of `rc.conf` and `sshd_config` which...
  - configure SSH access for root
  - install Python (needed for running the *configuration playbook*)
  - updates FreeBSD to the latest patch level upon first boot:

```
#!/bin/sh
cat << EOF > etc/rc.conf
ec2_configinit_enable="YES"
ec2_fetchkey_enable="YES"
ec2_fetchkey_user="root"
ec2_bootmail_enable="NO"
ec2_ephemeralswap_enable="YES"
ec2_loghostkey_enable="YES"
ifconfig_xn0="DHCP"
firstboot_freebsd_update_enable="YES"
firstboot_pkgs_enable="YES"
firstboot_pkgs_list="python27"
dumpdev="AUTO"
panicmail_enable="NO"
panicmail_autosubmit="NO"
sshd_enable="YES"
EOF

cat << EOF > /etc/ssh/sshd_config
Port 22
ListenAddress 0.0.0.0
Subsystem sftp /usr/libexec/sftp-server
PermitRootLogin without-password
UseDNS no
EOF
```

Now you can provision the instance by running:

```
# ploy start production-backend
```

This will take several minutes, as the machine is started up, updates itself and reboots. Be patient, it can easily take five minutes. To check if everything is done, use ploy's status command, once the instance is fully available it should say something like this:

```
# ploy status production-backend
INFO: Instance 'production-backend' (i-xxxxx) available.
INFO: Instance running.
INFO: Instances DNS name ec2-xxx-xx-xx-xx.eu-west-1.compute.amazonaws.com
INFO: Instances private DNS name ip-xxx-xx-xx-xx.eu-west-1.compute.internal
INFO: Instances public DNS name ec2-xx-xx-xx-xx.eu-west-1.compute.amazonaws.com
INFO: Console output available. SSH fingerprint verification possible.
```

Especially the last line means that the new instance is now ready.

You should now be able to log in via SSH:

```
ploy ssh production-backend
```

**Note:** Unlike with *plain* or *Virtualbox* instances, daemonology's configinit in conjunction with a `startup_script`

such as the example above already perform everything we need in order to be able to run the jailhost playbooks. In other words, you can skip the *Bootstrapping* step and continue straight to *Configuring a jailhost*.

But before continuing on to *Configuring a jailhost*, let's take a look around while we're still logged in and note what hard disks and network interfaces are available. I.e. on our example machine of `c3.xlarge` type, the interface is named `xn0` and we have two SSDs of 40Gb at `/dev/xbd1` and `/dev/xbd2`, but by default daemonology has already created a swap partition on the first slice (highly recommended, as most instance types don't have that much RAM), so we need to specify the second slice for our use.

This means, that to configure a jailhost on this EC2 instance we need to declare an `ez-master` entry in `ploy.conf` with the following values:

```
[ez-master:production]
instance = production-backend
bootstrap_data_pool_devices = xbd1s2 xbd2s2
```

In addition, since daemonology will also update the installation to the latest patch level, we will need to explicitly tell `ezjail` which version to install, since by default it uses the output of `uname` to compute the URL for downloading the base jail but that most likely won't exist (i.e `10.0-RELEASE-p10`). You can do this by specifying `ezjail_install_release` for the `ez-master` like so:

```
ezjail_install_release = 10.0-RELEASE
```

With this information you are now finally and truly ready to *configure the jailhost.*.

## 9.2 Bootstrapping

Bootstrapping in the context of BSDploy means installing FreeBSD onto a *previously provisioned host* and the smallest amount of configuration to make it ready for its final configuration.

The Bootstrapping process assumes that the target host has been booted into an installer and can be reached via SSH under the configured address and that you have configured the appropriate bootstrapping type (currently either `mfsbsd` or `daemonology`).

### 9.2.1 Bootstrapping FreeBSD 9.x

The default version that BSDploy assumes is 10.3. If you want to install different versions, i.e. 9.2 you must:

- use the iso image for that version:

```
% ploy-download http://mfsbsd.vx.sk/files/iso/9/amd64/mfsbsd-se-9.2-RELEASE-amd64.
↪iso 4ef70dfd7b5255e36f2f7e1a5292c7a05019c8ce downloads/
```

- set `bootstrap-host-key` in `ploy.conf` to content of `/etc/ssh/ssh_host_rsa_key.pub` in the mfsbsd image (each mfsbsd release has it's own hardcoded ssh host key)
- create a file named `files.yml` in `bootstrap-files` with the following contents:

```
---
'pkg.txz':
    url: 'http://pkg.freebsd.org/freebsd:9:x86:64/quarterly/Latest/pkg.txz'
    directory: '/mnt/var/cache/pkg/All'
    remote: '/mnt/var/cache/pkg/All/pkg.txz'
```

### 9.2.2 Bootstrap configuration

Since bootstrapping is specific to BSDploy we cannot configure it in the provisioning instance. Instead we need to create a specific entry for it in our configuration of the type `ez-master` and assign it to the provisioner.

I.e. in our example:

```
[ez-master:jailhost]
instance = ploy-demo
```

#### Required parameters

The only other required value for an `ez-master` besides its provisioner is the name of the target device(s) the system should be installed on.

If you don't know the device name FreeBSD has assigned, run `gpart list` and look for in the `Consumers` section towards the end of the output.

If you provide more than one device name, BSDploy will create a zpool mirror configuration, just make sure the devices are compatible.

There we can provide the name of the target device, so we get the following:

```
[ez-master:jailhost]
instance = ploy-demo
bootstrap-system-devices = ada0
```

Or if we have more than one device:

```
[ez-master:jailhost]
instance = ploy-demo
bootstrap-system-devices =
    ada0
    ada1
```

#### Optional parameters

You can use the following optional parameters to configure the bootstrapping process and thus influence what the jail host will look like:

- `bootstrap-system-pool-size`: BSDploy will create a zpool on the target device named `system` with the given size. This value will be passed on to `zfsinstall`, so you can provide standard units, such as `5G`. Default is `20G`.
- `bootstrap-swap-size`: This is the size of the swap space that will be created. Default is double the amount of detected RAM.
- `bootstrap-bsd-url`: If you don't want to use the installation files found on the installer image (or if your boot image doesn't contain any) you can provide an explicit alternative (i.e. `http://ftp4.de.freebsd.org/pub/FreeBSD/releases/amd64/9.2-RELEASE/`) and this will be used to fetch the system from.
- `bootstrap-host-key`: Since the installer runs a different sshd configuration than the final installation, we need to provide its ssh host key explicitly. However, if you don't provide one, BSDploy will assume the (currently hardcoded) host key of the 10.3 mfsBSD installer . If you are using newer versions you must update the value to the content of `/etc/ssh/ssh_host_rsa_key.pub` in the mfsbsd image.

- `firstboot-update`: By default bootstrapping will install and enable the firstboot-freebsd-update package. This will update the installed system automatically (meaning non-interactively) to the latest patchlevel upon first boot. If for some reason you do not wish this to happen, you can disable it by setting this value to `false`.
- `http_proxy`: If set, that proxy will be used for all `pkg` operations performed on that host, as well as for downloading any assets during bootstrapping (`base.tbz` etc.)

---

**Note:** Regarding the http proxy setting it is noteworthy, that `pkg` servers have rather restrictive caching policies in their response headers, so that most proxies' default configurations will produce misses. Here's an example for how to configure squid to produce better results:

```
# match against download urls for specific packages - their content never changes for␣
↪the same url, so we cache aggressively
refresh_pattern -i (quarterly|latest)\/All\/.*(\.txz) 1440 100% 1051200 ignore-
↪private ignore-must-revalidate override-expire ignore-no-cache
# match against meta-information - this shouldn't be cached quite so aggressively
refresh_pattern -i (quarterly|latest)\/.*(\.txz) 1440 100% 10080 ignore-private␣
↪ignore-must-revalidate override-expire ignore-no-cache
```

Also you will probably want to adjust the following:

```
maximum_object_size_in_memory 32 KB
maximum_object_size 2000 MB
```

---

### 9.2.3 Bootstrap rc.conf

A crucial component of bootstrapping is configuring `/etc/rc.conf`.

One option is to provide a custom rc.conf (verbatim or as a template) for your host via *Bootstrap files*.

But often times, the default template with a few additional custom lines will suffice.

Here's what the default `rc.conf` template looks like:

```
hostname="{{hostname}}"
sshd_enable="YES"
syslogd_flags="-ss"
zfs_enable="YES"
pf_enable="YES"
{% for interface in interfaces %}
ifconfig_{{interface}}="DHCP"
{% endfor %}
```

This is achieved by providing `boostrap-rc-xxxx` key/values in the instance definition in `ploy.conf`.

### 9.2.4 Bootstrap files

During bootstrapping a certain number of files are copied onto the target host.

Some of these files...

- need to be provided by the user (i.e. `authorized_keys`)
- others have some (sensible) defaults (i.e. `rc.conf`)

- some can be downloaded via URL (i.e.) `http://pkg.freebsd.org/freebsd:10:x86:64/latest/Latest/pkg.txz`

The list of files, their possible sources and their destination is encoded in a `.yml` file, the default of which is this

```
---
'rc.conf':
    remote: '/mnt/etc/rc.conf'
    use_jinja: True
'make.conf':
    remote: '/mnt/etc/make.conf'
'pkg.conf':
    remote: '/mnt/usr/local/etc/pkg.conf'
    use_jinja: True
'pf.conf':
    remote: '/mnt/etc/pf.conf'
'FreeBSD.conf':
    directory: '/mnt/usr/local/etc/pkg/repos'
    remote: '/mnt/usr/local/etc/pkg/repos/FreeBSD.conf'
    use_jinja: True
'sshd_config':
    remote: '/mnt/etc/ssh/sshd_config'
```

**Warning:** Overriding the list of default files is an advanced feature and in most cases it is not needed. Also keep in mind that bootstrapping is only about getting the host ready for running BSDploy. Any additional files beyond that should be uploaded lateron via fabric and/or playbooks.

It is however, quite common and useful to customize files that are part of the above list with custom versions *per host*.

For example, to create a custom `rc.conf` for a particular instance, create a `bootstrap-files` entry for it and point it to a directory in your project, usually `../bootstrap-files/INSTANCENAME/` and place your version of `rc.conf` inside there. Note, that by default this file is rendered as a template, your custom version will be, too.

Any file listed in the YAML file found inside that directory will take precedence during bootstrapping, but any file *not* found in there will be uploaded from the default location instead.

Files encrypted using `ploy vault encrypt` are recognized and decrypted during upload.

SSH host keys are generated locally via `ssh-keygen` and stored in `bootstrap-files`. If your `ssh-keygen` doesn't support a key type (like ecdsa on OS X), then the key won't be created and FreeBSD will create it during first boot. The generated keys are used to verify the ssh connection, so it is best to add them into version control. Since the private ssh keys are sensitive data, you should encrypt them using `ploy vault encrypt ...` before adding them into version control.

### 9.2.5 Bootstrap execution

With (all) those pre-requisites out of the way, the entire process boils down to issuing the following command:

```
% ploy bootstrap
```

Or, if your configuration has more than one instance defined you need to provide its name, i.e.:

```
% ploy bootstrap jailhost
```

Once this has run successfully, you can move on to the final setup step *Configuration*.

## 9.3 Configuring a jailhost

Once the host has been successfully bootstrapped, we are left with a vanilla FreeBSD installation with the following exceptions:

- we have key based SSH access as root
- Python has been installed

But before we can create and manage jails, a few tasks still remain, in particular

- installation and configuration of `ezjail`
- ZFS setup and layout, including optional encryption
- jail specific network setup

Unlike bootstrapping, this final step is implemented using ansible playbooks and has been divided into multiple roles, so all that is left for us is to apply the `configure` to the `ez-master` instance, i.e. like so:

```
ploy configure ploy-demo
```

Among other things, this will create an additional zpool named `tank` (by default) which will be used to contain the jails.

### 9.3.1 Configuring as non-root

While bootstrapping currently *must* be performed as `root` (due to the fact that mfsBSD itself requires root login) some users may not want to enable root login for their systems.

If you want to manage a jailhost with a non-root user, you must perform the following steps manually:

- install `sudo` on the jailhost
- create a user account and enable SSH access for it
- enable passwordless `sudo` access for it
- disable SSH login for root (currently, automatically enabled during bootstrapping)

Additionally, you *must* configure the system using a playbook (i.e. simply assigning one or more roles won't work in this case) and in that playbook you must set the username and enable `sudo`, i.e. like so:

```
---
- hosts: jailhost
  user: tomster
  sudo: yes
  roles:
    # apply the built-in bsdploy role jails_host
    - jails_host
```

And, of course, once bootstrapped, you need to set the same username in `ploy.conf`:

```
[ez-master:jailhost]
user = tomster
```

### 9.3.2 Full-Disk encryption with GELI

One of the many nice features of FreeBSD is its modular, layered handling of disks (GEOM). This allows to inject a crypto layer into your disk setup without that higher up levels (such as ZFS) need to be aware of it, which is exactly what BSDploy supports.

If you add `bootstrap-geli = yes` to an `ez-master` entry, BSDploy will generate a passphrase, encrypt the GEOM provider for the `tank` zpool and write the passphrasse to `'/root/geli-passphrase` and configures the appropriate `geli_*_flag` entries in `rc.conf` so that it is used automatically during booting.

The upshot is that when enabling GELI you still will have the same convenience as without encryption but can easily up the ante by removing the passphrase file (remember to keep it safe, though!). You will, however, need to attach the device manually after the system has booted and enter the passphrase.

### 9.3.3 Additional host roles

The main bulk of work has been factored into the role `jails_host` which also is the default role.

If the network of your host is configured via DHCP you can apply an additional role named `dhcp_host` which takes care of the hosts sshd configuration when the DHCP lease changes. To have it applied when calling `configure` add an explicit `roles` parameter to your `ez-instance`:

```
[ez-master:ploy-demo]
[...]
roles =
        dhcp_host
        jails_host
[...]
```

Technically, BSDploy injects its own roles to ansibles playbook path, so to apply your own custom additions, add additional roles to a top-level directory named `roles` and include them in your configuration and they will be applied as well.

Common tasks for such additional setup could be setting up a custom ZFS layout, configuring snapshots and backups, custom firewall rules etc, basically anything that you would not want to lock inside a jail.

---

**Note:** Curently, the `jails_host` playbook is rather monolithic, but given the way ansible works, there is the possibility of making it more granular, i.e. by tagging and/or parametrisizing specific sub-tasks and then to allow applying tags and parameter values in `ploy.conf`.

---

# CHAPTER 10

# General usage

How to create and manage jails once the host is set up:

## 10.1 Managing jails

The life cycle of a jail managed by BSDploy begins with an `instance` entry in `ploy.conf`, i.e. like so:

```
[instance:webserver]
ip = 10.0.0.1
master = ploy-demo
```

The minimally required parameters are the IP address of the jail (`ip`) and a reference to the jailhost (`master`) – the name of the jail is taken from the section name (in the example `webserver`).

---

**Note:** Unlike `ez-master` or other instances, names of jails are restricted by the constraints that FreeBSD imposes, namely they cannot contain dashes (`-`)

---

BSDploy creates its own loopback device (`lo1`) during configuration and assigns a network of `10.0.0.0/8` by default (see `bsdploy/roles/jails_host/defaults/main.yml` for other values and their defaults), so you can use any `10.x.x.x` IP address out-of-the-box for your jails.

Once defined, you can start the jail straight away. There is no explicit `create` command, if the jail does not exist during startup, it will be created on-demand:

```
# ploy start webserver
INFO: Creating instance 'webserver'
INFO: Starting instance 'webserver' with startup script, this can take a while.
```

You can find out about the state of a jail by running `ploy status JAILNAME`.

A jail can be stopped with `ploy stop JAILNAME`.

A jail can be completely removed with `ploy terminate JAILNAME`. This will destroy the ZFS filesystem specific to that jail.

### 10.1.1 SSH Access

BSDploy encourages jails to have a private IP address but compensates for that by providing convenient SSH access to them anyway, by automatically configuring an SSH ProxyCommand.

Essentially, this means that you can SSH into any jail (or other instance) by providing it as a target for ploy's `ssh` command, i.e.:

```
# ploy ssh webserver
FreeBSD 9.2-RELEASE (GENERIC) #6 r255896M: Wed Oct  9 01:45:07 CEST 2013

Gehe nicht über Los.
root@webserver:~ #
```

Strictly speaking, you would need to address the jail instance together with the name of the host (to disambiguate multi-host scenarios) but since in this example there is only one jail host defined, `webserver` is enough, otherwise you would use `jailhost-webserver`.

#### rsync and scp

To access a jail with `rsync` (don't forget to install the `rsync` package into it!) or `scp` you can pass the `ploy-ssh` script into them like so:

```
scp -S ploy-ssh some.file webserver:/some/path/
rsync -e ploy-ssh some/path webserver:/some/path
```

## 10.2 Ansible integration

Not only does BSDploy use ansible playbooks internally to configure jail hosts (via the ploy_ansible plugin), but you can use it to configure jails, too, of course.

---

**Note:** An important difference between using ansible *as is* and via BSDploy is that unlike ansible, BSDploy does not require (or indeed support) what ansible refers to as an inventory file – all hosts are defined within ploy's configuration file.

---

You can either assign roles or a playbook file to an instance and then apply them via the `configure` command.

### 10.2.1 Playbook assignment

There are two ways you can assign a playbook to an instance – either via naming convention or by using the `playbook` parameter.

Assigning by convention works by creating a top-level `*.yml` file with the same name as the instance you want to assign it to.

Keep in mind, however, that unlike when targetting instances from the command line, there is no aliassing in effect, meaning you *must* use the *full* name of the jail instance in the form of *hostname-jailname*, i.e. in our example `jailhost-webserver.yml`.

The *by-convention* method is great for quick one-off custom tasks for an instance – often in the playbook you include one or more roles that perform the bulk of the work and add a few top-level tasks in the playbook that didn't warrant the overhead of a a role of their own. However, the target of the playbook is effectively hard-coded in the name of the file, so re-use is not possible.

If you want to use the same playbook for multiple instances, you can assign it by using `playbook = PATH`, where `PATH` usually is a top level `*.yml` file.

Note that any paths are always relative to the location of the `ploy.conf` file, so usually you would have to refer to a top-level file like so:

```
[instance:webserver1]
...
playbook = ../nginx.yml

[instance:webserver2]
...
playbook = ../nginx.yml
```

### 10.2.2 Role assignment

Playbook assignment can be convenient when dealing with ad-hoc or one-off tasks but both ansible and BSDploy strongly encourage the use of roles.

Role assignment works just as you've probably guessed it by now: by using a `roles` parameter. Unlike with `playbook` you can assign more than one role. You do this by listing one role per line, indented for legibility, i.e.:

```
[instance:webserver]
roles =
        nginx
        haproxy
        zfs-snapshots
```

**Note:** Assignment of roles and assignment of playbooks are mutually exclusive. If you try to do both, BSDploy will raise an error.

### 10.2.3 Tags

When applying a playbook or roles via the `configure` command you can select only certain tags of them to be executed by adding the `-t` parameter, i.e. like so:

```
ploy configure webserver -t config
```

To select multiple tags, pass them in comma-separated. Note that in this case you must make sure you don't add any whitespace, i.e.:

```
ploy configure webserver -t config,cert
```

### 10.2.4 Directory structure

The directory of a BSDploy environment is also an ansible project structure meaning, you can create and use top-level directories such as `roles`, `group_vars` or even `library`, etc. (see the link about the ansible directory structure).

## 10.3 Fabric integration

> Fabric is a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.

BSDploy supports applying Fabric scripts to jails and jail hosts via the ploy_fabric plugin.

There are two ways you can assign a fabric file to an instance:

- *by convention* – if the project directory contains a directory with the same name as the instance, i.e. `jailhost` or `jailhost-jailname` containing a Python file named `fabfile.py` and no explicit file has been given, that file is used
- by *explicit assigning one* in `ploy.conf`, i.e.:

```
[instance:webserver]
...
fabfile = ../fabfile.py
```

### 10.3.1 Fabfile anatomy

Let's take a look at an example `fabfile.py`:

```
1 from fabric import api as fab
2
3 fab.env.shell = '/bin/sh -c'
4
5 def info():
6     fab.run('uname -a')
7
8 def service(action='status'):
9     fab.run('service nginx %s' %  action)
```

- (1) Fabric conveniently exposes all of its features in its `api` module, so we just import that for convenience
- (3) Fabric assumes `bash`, currently we must explicitly adapt it FreeBSD's default (this may eventually be handled by a future version of BSDploy)
- (8) You can pass in parameters from the command line to a fabric task, see Fabric's documentation on this for more details.

### 10.3.2 Fabfile execution

You can execute a task defined in that file by calling `do`, i.e.:

```
# ploy do webserver service
[root@jailhost-webserver] run: service nginx status
[root@jailhost-webserver] out: nginx is running as pid 1563.
```

Fabric has a relatively obtuse syntax for passing in arguments because it supports passing to multiple hosts in a single call.

To alleviate this, `ploy_fabric` adds a simpler method in its `do` command, since that only always targets a single host.

So, in our example, to restart the webserver you could do this:

```
# ploy do webserver service action=restart
[root@jailhost-webserver] run: service nginx restart
[root@jailhost-webserver] out: Performing sanity check on nginx configuration:
[root@jailhost-webserver] out: nginx: the configuration file /usr/local/etc/nginx/
↪nginx.conf syntax is ok
[root@jailhost-webserver] out: nginx: configuration file /usr/local/etc/nginx/nginx.
↪conf test is successful
[root@jailhost-webserver] out: Stopping nginx.
[root@jailhost-webserver] out: Waiting for PIDS: 1563.
[root@jailhost-webserver] out: Performing sanity check on nginx configuration:
[root@jailhost-webserver] out: nginx: the configuration file /usr/local/etc/nginx/
↪nginx.conf syntax is ok
[root@jailhost-webserver] out: nginx: configuration file /usr/local/etc/nginx/nginx.
↪conf test is successful
[root@jailhost-webserver] out: Starting nginx.
```

You can also list all available tasks with the `-l` parameter:

```
ploy do webserver -l
Available commands:

    info
    service
```

## 10.4 Combining Ansible and Fabric

Both Ansible and Fabric are great tools, but they truly shine when used together.

A common pattern for this is that a playbook is used to set up a state against which then a fabric script is (repeatedly) executed.

For example an initial setup of an application, where a playbook takes care that all required directories are created with the proper permissions and into which then a fabric script performs an upload of the application code.

### 10.4.1 Sharing variables between playbooks and fabric scripts

For such a collaboration both fabric and ansible need to know *where* all of this should take place, for instance. I.e. fabric has to know the location of the directory that the playbook has created.

You can either define variables directly in `ploy.conf` or in group or host variables such as `group_vars/all.yml` or `group_vars/webserver.yml`.

To create key/value pairs in `ploy.conf` that are visible to ansible, you must prefix them with `ansible-`.

For example, you could create an entry in ploy.conf like so:

```
[instance:webserver]
...
ansible-frontend_path = /opt/foo
```

And then use the following snippet in a playbook:

```
- name: ensure the www data directory exists
  file: path={{frontend_path}} state=directory mode=775
```

Applying the playbook will then create the application directory as expected:

```
ploy configure webserver
PLAY [jailhost-webserver] *****************************************************

GATHERING FACTS ***************************************************************
ok: [jailhost-webserver]

TASK: [ensure the www data directory exists] ***********************************
changed: [jailhost-webserver]
```

Now let's create a fabric task that uploads the contents of that website:

```
def upload_website():
    ansible_vars = fab.env.instance.get_ansible_variables()
    fab.put('dist/*', ansible_vars['frontend_path'] + '/')
```

Notice, how we're accessing the ansible variables via Fabrics' `env` where `ploy_fabrics` has conveniently placed a ploy instance of our host.

Let's run that:

```
# ploy do webserver upload_website
[root@jailhost-webserver] put: dist/index.html -> /opt/foo/index.html
```

Putting variables that you want to share between fabric and ansible into your `ploy.conf` is the recommended way, as it upholds the configuration file as the canonical place for all of your configuration.

However, until ploy learns how to deal with multi-line variable definitions, dealing with such requires setting them in `group_vars/all.yml`.

CHAPTER 11

# Special use cases

## 11.1 Staging

One use case that BSDploy was developed for is to manage virtual copies of production servers.

The idea is to have a safe environment that mirrors the production environment as closely as possible into which new versions of applications or system packages can be installed or database migrations be tested etc.

But safe testing and experimenting is just one benefit, the real benefit comes into play, once you've tweaked your staging environment to your liking. Since BSDploy makes it easy for you to capture those changes and tweaks into playbooks and fabric scripts, applying them to production is now just a matter changing the target of a `ploy` command.

### 11.1.1 Extended configuration files

To help you keep the staging and production environment as similar as possible we will use the ability of `ploy` to inherit configuration files.

We define the general environment (one or more jail hosts and jails) in a `base.conf` and create two 'top-level' configuration files `staging.conf` and `production.conf` which each extend `base.conf`.

During deployment we specify the top-level configuration file via' `ploy -c xxx.conf`. A variation of this is to name the staging configuration file `ploy.conf` which then acts as default. This has the advantage that during development of the environment you needn't bother with explicitly providing a configuration file and that when moving to production you need to make the extra, explicit step of (now) providing a configuration file, thus minimizing the danger of accidentally deploying something onto production.

Here is an example `base.conf`:

```
[ez-master:jailhost]
instance = provisioner
roles = jails_host

[instance:webserver]
master = jailhost
```

```
ip = 10.0.0.1


[instance:appserver]
master = jailhost
ip = 10.0.0.2
```

and `staging.conf`:

```
[global]
extends = base.conf

[vb-instance:provisioner]
vm-ostype = FreeBSD_64
[...]
```

and `production.conf`:

```
[global]
extends = base.conf

[ec2-instance:provisioning]
ip = xxx.xxx.xxx.xxx
instance_type = m1.small
# FreeBSD 9.2-RELEASE instance-store eu-west-1 from daemonology.net/freebsd-on-ec2/
image = ami-3e1ef949
[...]
```

**Note:** In practice, it can often be useful to split this out even further. We have made good experiences with using a virtualbox based setup for testing the deployment during its development, then once that's finished we apply it to a public server (that can be accessed by stakeholders of the project for evaluation) that actually runs on the same platform as the production machine and once that has been approved we finally apply it to the production environment. YMMV.

### 11.1.2 Staging with FQDN

A special consideration when deploying web applications is how to test the entire stack, including the webserver with fully qualified domain names and SSL certificates etc.

BSDploy offers a neat solution using the following three components:

- template based webserver configuration
- xip.io based URLs for testing
- VirtualBox host-only networking

IOW if you're deploying a fancy-pants web application at your production site `fancypants.com` using i.e. a nginx configuration snippet like such:

```
server {
    server_name  fancypants.com;
        [...]
}
```

Change it to this:

```
server {
    server_name  fancypants.com{{fqdn_suffix}};
        [...]
}
```

And in your `staging.conf` you define `fqdn_suffix` to be i.e. `.192.168.56.10.xip.io` and in `production.conf` to an empty string.

Finally, configure the VirtualBox instance in staging to use a second nic (in addition to the default host-only interface) via DHCP so it can access the internet:

```
[vb-instance:provisioner]
vm-nic2 = nat
```

`ploy_virtualbox` will ensure that the virtual network `vboxnet0` exists (if it doesn't already). You can then use the fact that VirtualBox will set up a local network (default is `192.168.56.xxx`) with a DHCP range from `.100 - .200` and assign your nic1 (`em0` in our case) a static IP of, i.e. `192.168.56.10` which you then can use in the abovementioned xip.io domain name.

The net result? Deploy to staging and test your web application's full stack (including https, rewriting etc.) in any browser under `https://fancypants.com.192.168.56.10.xip.io` in the knowledge that the only difference between that setup and your (eventual) production environment is a single suffix string.

## 11.2 Updating

While in theory automated systems such as ploy allow you to create new and up-to-date instances easily and thus in theory you would never have to upgrade existing instances because you would simply replace them with newer versions, in practice both jails and host systems will often have to be upgrade in place.

To support this, bsdploy provides a few helper tasks which are registered by default for jailhosts.

If you want to use them in your own, custom fabfile you must import them their to make them available, i.e. like so:

```
from bsdploy.fabutils import *
```

You can verify this by running the *do* command with *-l*:

```
# ploy do HOST -l Available commands:

    pkg_upgrade rsync update_flavour_pkg
```

You can use the *pkg_upgrade* task to keep the pkg and and the installed packages on the host up-to-date.

The *update_flavour_pkg* is useful after updating the ezjail world, so that newly created jails will have an updated version of pkg from the start. (if the pkg versions become too far apart it can happen, that new jails won't bootstrap at all, because they already fail at installing python).

See the *fabutils.py* file for more details.

## 11.3 Customizing bootstrap

Currently the bootstrap API isn't ready for documentation and general use. In case you want to mess with it anyway, here are some things which will be safe to do.

### 11.3.1 mfsBSD http proxy example

If you are setting up many virtual machines for testing, then a caching http proxy to reduce the downloads comes in handy. You can achieve that by using polipo on the VM host and the following changes.

First you need a custom fabfile:

```
from bsdploy.fabfile_mfsbsd import _bootstrap, _mfsbsd
from fabric.api import env, hide, run, settings
from ploy.config import value_asbool


def bootstrap(**kwargs):
    with _mfsbsd(env, kwargs):
        reboot = value_asbool(env.instance.config.get('bootstrap-reboot', 'true'))
        env.instance.config['bootstrap-reboot'] = False
        run('echo setenv http_proxy http://192.168.56.1:8123 >> /etc/csh.cshrc')
        run('echo http_proxy=http://192.168.56.1:8123 >> /etc/profile')
        run('echo export http_proxy >> /etc/profile')
        _bootstrap()
        run('echo setenv http_proxy http://192.168.56.1:8123 >> /mnt/etc/csh.cshrc')
        run('echo http_proxy=http://192.168.56.1:8123 >> /mnt/etc/profile')
        run('echo export http_proxy >> /mnt/etc/profile')
        if reboot:
            with settings(hide('warnings'), warn_only=True):
                run('reboot')
```

For the ezjail initialization you have to add the following setting with a FreeBSD http mirror of your choice to your jail host config in `ploy.conf`:

```
ansible-ploy_ezjail_install_host = http://ftp4.de.freebsd.org
```

The `_mfsbsd` context manager takes care of setting the `bootstrap-host-keys` etc for mfsBSD. The `_bootstrap` function then runs the regular bootstrapping.

For the jails you can use a startup script like this:

```
#!/bin/sh
exec 1>/var/log/startup.log 2>&1
chmod 0600 /var/log/startup.log
set -e
set -x
echo setenv http_proxy http://192.168.56.1:8123 >> /etc/csh.cshrc
echo http_proxy=http://192.168.56.1:8123 >> /etc/profile
echo export http_proxy >> /etc/profile
http_proxy=http://192.168.56.1:8123
export http_proxy
pkg install python27
```

CHAPTER 12

# Contribute

Code and issues are hosted at github:

- Issue Tracker: http://github.com/ployground/bsdploy/issues
- Source Code: http://github.com/ployground/bsdploy
- IRC: the developers can be found on #bsdploy on freenode.net

CHAPTER 13

# License

The project is licensed under the Beerware license.

CHAPTER 14

# TODO

The following features already exist but still need to be documented:

- **provisioning + bootstrapping**
  - EC2 (daemonology based)
  - pre-configured SSH server keys
- **jail access**
  - port forwarding
  - public IP
- ZFS management
- Creating and restoring ZFS snapshots
- poudriere support
- Upgrading strategies
- 'vagrant mode' (use - virtualized - jails as development environment)

The following features don't exist yet but should eventually :)

- **OS installers**
  - homebrew
- support vmware explicitly (like virtualbox)?