Rambo Documentation

Release 0.4.5.dev0

Terminal Labs

Mar 14, 2023

Usage Documentation

Currently viewing version v0.4.5.dev0

CHAPTER 1

Quickstart

To get started fast, see Quickstart.

1.1 Quickstart

1.1.1 Hardware Recommendations

For running VMs locally in VirtualBox (default), we suggest a minimum of:

- Reasonably fast cpu with 2 cores and virtualization hardware support (e.g. an Intel i7-3612QM 2.1GHz, 4 core chip with VT-x)
- 8GB RAM
- 16GB free drive space

For running containers locally (e.g. LXC) or spawning cloud based VMs (e.g. AWS EC2) you can get away with comparatively slow computer and you don't need VT-x, you don't even need VirtualBox. In fact, these providers can be managed from just a Raspberry Pi.

1.1.2 Supported Host Operating Systems

- Ubuntu 16.04 or newer
- OSX

We expect it's likely you can get Rambo to work on any Unix-like system, but your milleage may vary. So far we have made no effort to get this working with Windows. Contributions are very welcome.

1.1.3 Installation

1. Install / Use Python 3.6+ and pip (for example with a Virtual Environment).

2. Download and install VirtualBox 5.1 or newer.
3. Download and install Vagrant.
4. Install Rambo with pip,
    - latest release with pypi: `pip install rambo-vagrant`, or
    - from source: go into the repository and `pip install -e .`
5. Install plugins with `rambo install-plugins`

Note: Vagrant and VirtualBox update frequently, and sometimes with breaking changes. Additionally there are may be provider specific dependencies.

1.1.4 Create Project

Now that Rambo is installed, you must initialize a project. This will create a directory that will be tied to your VM. Outside of this directory, Rambo won't be able to find the VM to control it. This also means that if you want to create or control multiple VMs with Rambo, you can, by simply creating more projects and running Rambo commands from the directories where they reside. Create and go to your project:

```
rambo createproject yourprojectname
cd yourprojectname
```

In this project directory Rambo gave you a few things to help you get started, a `rambo.conf`, `auth` dir, and `saltstack` dir. These are basic configs to start you out. You don't need to modify them for basic use.

1.1.5 Providers

Rambo supports various providers, and aims to let you switch between them as easily as possible. Nevertheless, some providers do have particular considerations, such as setting up keys and payment for cloud services, or specific dependencies for the host OS. This is a list of Rambo's supported providers, with links to specific documentation pages for each.

- *AWS EC2*
- *DigitalOcean*
- *Docker*
- *LXC*
- VirtualBox (see below)

Default Provider - VirtualBox:

If you never specify any provider, Rambo will use the VirtualBox as its default choice, and is simply

`rambo up`

1.1.6 Provisioning

Rambo does very little provisioning on its own. It can set a hostname, set up some synced directories, and allow a command to be run. That command is your entry point to doing anything else.

See provisioning for some examples.

1.2 Python API

Rambo's CLI and Python API are compatible. In other words, what you can do in the CLI, you can do in the Python API. To accomplish this the CLI is largely dependant on the Python API. You can access the Python API by importing the various functions in app.py, such as with `from rambo.app import up`

Through the Python API you can call `up` and `destroy` to create and destroy VMs. `ssh` is also available and presents an interactive shell to the VM, as if you ran `rambo ssh` with the CLI.

CLI options are available to be set as either functions in app.py, or as parameters to those functions, depending on whether the CLI option was on `rambo` or a command (e.g. `up`). For instance, the following are equivalent:

```
rambo --vagrant-cwd /sample/path up -p virtualbox
```

```
from rambo.app import up
up(vagrant_cwd,"/sample/path", provider="virtualbox")
```

1.3 CLI

Rambo's CLI is the expected normal way people interact with Rambo. At it's core, Rambo is an interface to Vagrant. Rambo duplicates several commands from Vagrant, that are either commonly used, or Rambo needs to do some preemptive work for before passing the reigns to Vagrant. For most other Vagrant commands, you can call Vagrant through Rambo. Many commands have various options that have defaults that are used when the option is not specified, e.g. `rambo up` defaults using VirtualBox as the `provider`.

This is a short list of Rambo's commands, followed by a more detailed explanation of each:

1.3.1 Commands

- *createproject*: Create a Rambo project dir with basic setup.
- *destroy*: Destroy a VM / container and all its metadata. Default leaves logs.
- *export-vagrant-conf*: Get Vagrant configuration.
- *halt*: Halt VM.
- *install-plugins*: Install Vagrant plugins.
- *scp*: Transfer files with scp.
- *ssh*: Connect with `vagrant ssh`
- *up*: Start a VM / container with `vagrant up`.
- *vagrant*: Run a Vagrant command through Rambo.

createproject

Create project takes an arguement for the name to give to the project it creates. It will create a directory in the CWD for this project. Upon creation, this project directory will contain a `rambo.conf` file, an `auth` directory, and a `saltstack` directory.

- `rambo.conf` is the config file that is required to be present in your project to run `rambo up`, and is described later in conf.md.

- `auth` contains some sample scripts that will aid in setting up keys / tokens for the cloud providers. It is not required. How to use that is described in the cloud provider specific documentation.
- `saltstack` is a basic set of SaltStack configuration code that Rambo offers. It can be modified for custom configuration.

destroy

Destroy a VM / container. This will tell vagrant to forcibly destroy a VM, and to also destroy its Rambo metadata (provider and random_tag), and Vagrant metadata (`.vagrant` dir).

export-vagrant-conf

Places the default `Vagrantfile` and its resources (`vagrant` dir, `settings.json`) in the CWD for customizing.

halt

Tells Vagrant to 'halt' the VM. Useful to free the Host's resources without destroying the VM.

install-plugins

Install passed args as Vagrant plugins. `all` or no args installs all default Vagrant plugins from host platform specific list.

scp

Transfer files or directories with scp. Accepts two args in one of the following forms:

```
<local_path> <remote_path>

<local_path> :<remote_path>

:<remote_path> <local_path>

<local_path> [vm_name]:<remote_path>

[vm_name]:<remote_path> <local_path>
```

For example: `rambo scp localfile.txt remotefile.txt`

ssh

Connect to the VM / container over SSH. With `-c` / `--command`, will executed an SSH command directly.

up

Start a VM or container. Will create one and begin provisioning it if it did not already exist. Accepts many options to set aspects of your VM. Precedence is CLI > Config > Env Var > defaults.

vagrant

Accepts any args and forwards them to Vagrant directly, allowing you to run any Vagrant command. Rambo has first-class duplicates or wrappers for the most common Vagrant commands, but for less common commands or commands that are not customized, they don't need to be duplicated, so we call them directly.

1.3.2 rambo.conf

The `rambo.conf` file is used to add options to various Rambo commands without having to pass them to the CLI. This is encouraged and has a few benefits. See the following quick example:

```
[up]
provider = digitalocean
guest_os = centos-7
```

is equivalent to:

```
rambo up --provider digitalocean --guest-os centos-7
```

An optional `my_rambo.conf` is also used, so you can have personallized and untracked configuration.

For a more detailed description, see the separate rambo.conf docs.

1.3.3 Environment Variables

RAMBO_ env vars

Like the config file, options can also be specified as environment variables. All CLI options that you can pass to Rambo are available to set by environment variables. They take the form of the CLI option, but are all upper-cased, use underscores instead of dashes, and are prefixed with `RAMBO_`. E.g. `RAMBO_GUEST_OS` is the environment variable for the CLI option `--guest-os`.

VAGRANT_ env vars

This is strongly discouraged.

Rambo uses Vagrant, so Vagrant specific environment variables can be used. Rambo itself sets some of these after its CLI invocation, so these may be overridden by Rambo. We do not support using these env vars manually with Rambo.

1.4 rambo.conf

The `rambo.conf` file is used to add options to various Rambo commands without having to pass them to the CLI. This is encouraged and has a few benefits. See the following quick example:

```
[up]
provider = digitalocean
guest_os = centos-7
```

is equivalent to:

```
rambo up --provider digitalocean --guest-os centos-7
```

The `rambo.conf` file is required at the top level in your project directory. It is an INI config file that can specify options. Options passed to the CLI will take precedence over options set via this config file. If you're repeating the same CLI options, setting those options in this config might make your life a little easier. Further, if you intend on tracking your Rambo project in version control, it can be very handy to set some options in this config that match the purpose of your project.

Options can be set in `rambo.conf`. For example, a useful `rambo.conf` could look like this:

```
[up]
provider = digitalocean
guest_os = centos-7
```

which is equivalent to:

```
rambo up --provider digitalocean --guest-os centos-7
```

Setting the config file to this allows you to type simply `rambo up` to run `up` with the `provider` and `guest-os` options set in the rambo.conf, and not specified in the CLI.

1.4.1 Option Names

The options in the conf file are the same as the full option names in the CLI, with preceeding dashes removed and other dashes replaced with underscores. As examples:

- `vagrant_dotfile_path` in the conf, corresponds to `--vagrant-dotfile-path` in the CLI
- `provider` in the conf, corresponds to `--provider` or `-p` in the CLI
- `guest_os` in the conf, corresponds to `--guest-os` or `-o` in the CLI
- `ram_size` in the conf, corresponds to `--ram-size` or `-r` in the CLI

The full list is available with `rambo up --help`.

1.4.2 my_rambo.conf

Rambo will also load configuration from a `my_rambo.conf` file. This file is optional, and configuration found here takes precedence over the main `rambo.conf` file.

The intention is that a `rambo.conf` file is tracked (e.g. with git), but so that a shared project can have its configuration easily altered by individual users, values may be overridden by an untracked `my_rambo.conf`. For example, a project may use `provider = ec2`, but individual contributors may want to develop locally in Docker or VirtualBox instead.

1.4.3 Option Precedence

The precedence for options is:

CLI > Environment Variable > `my_rambo.conf` > `rambo.conf` > defaults

When an option is set in more than one place, the CLI takes precedence. Defaults are overridable by everything.

Example 1

```
# rambo.conf

[up]
provider = digitalocean
```

```
rambo up -p virtualbox
```

yields the provider `virtualbox`.

Example 2

If instead, the config still read

```
# rambo.conf

[up]
provider = digitalocean
```

```
rambo up
```

yields the provider `digitalocean`.

Example 3

```
RAMBO_PROVIDER=digitalocean rambo up -p ec2
```

yields the provider `ec2`.

Example 4

```
# rambo.conf

[up]
provider = ec2
```

```
RAMBO_PROVIDER=digitalocean rambo up
```

yields the provider `digitalocean`.

Example 4

```
# my_rambo.conf

[up]
provider = virtualbox
```

```
# rambo.conf

[up]
provider = ec2
```

yields the provider `virtualbox`.

1.5 Customizing Rambo

Rambo aims to make it easy for you to switch providers and customize provisioning. Below is documentation about how to go about cusomizing your provisioning with Salt Stack, switching provisioners, adding providers, and customizing provider-specific code.

Rambo is young, and we'd love to improve Rambo and make this all easier still. Please consider opening a pull request if you add another provisioner or provider, or make any customization that would be a good contribution. :)

1.5.1 Custom Provisioning

Rambo provides a basic default provisioning with Vagrant and SaltStack. To build out what you need for your project you will need your own customized provisioning. You can do this provisioning with any tool you like through Vagrant, such as with shell scripts, SaltStack, or any other provisioning tool.

All Rambo code that is used to provision the VM is kept where Rambo is installed. This directory is copied into the VM at an early stage in the spawn process so that it can be invoked to provision the VM.

SaltStack

Rambo has a few basic Salt States available that are placed in your project dir by `rambo createproject`. These run unless removed, and work out of the box. The `saltstack` dir can also be modified however you like for any SaltStack provisioning. You can add your custom Salt States right into the Salt code and they should be automatically picked up and used.

Other Provisioners

If you want to add provisioning with any other tool, you will need to modify the Vagrantfiles to add that provisioning. To export the Vagrantfiles, run `rambo export-vagrant-conf` inside your project dir. This will drop the Vagrantfile and several of its dependencies. You can likely add custom provisioning straight to the main Vagrantfile without worrying about the other files.

For example, if you'd like to provision with Ansible, you will need to add custom Vagrant code to make this work. There are many useful introductions to various provisioners on Vagrant's website, such as the page on Ansible Provisioning.

1.5.2 Custom Providers / Provider configuration

First grab the Vagrantfiles with `rambo export-vagrant-conf`.

The main Vagrantfile is extended by other provider-specific vagrantfiles located in `vagrant/vagrantfiles` such as `vagrant/vagrantfiles/virtualbox` for VirtualBox. If you need to customize how Rambo works with a provider manually, these are the files you'll need to modify. For instance, you may want to customize many aspects of your VirtualBox VM's networking.

1.6 Basic Provisioning

By default Rambo will do a small amount of basic provisioning. It will:

- Set the hostname
- Sync your project directory
- Sync custom directories
- Run a custom command

1.6.1 Hostname

The hostname can be set by specifying the `--hostname` option.

1.6.2 Syncing

Sync Types

The default syncing method is whatever Vagrant uses as its default (link).

> Vagrant will automatically choose the best synced folder option for your environment.

For basic usage, that means the default for Rambo, using the VirtualBox provider, is shared folders.

This can be changed to any of the syncing methods that Vagrant supports; see their docs for details. Syncing can also be entirely disabled. Common options are:

- `disabled`
- `rsync`
- `shared`

These can be specified with `--sync-type`.

`--sync-type disabled` turns off syncing entirely.

Synced directories

Rambo will sync your project directory and any custom directories. The following mappings are synced in the order listed.

Host (source) VM (target) ============== ===========

[project dir] /vagrant custom dirs on host custom dirs on VM

Custom dirs can be passed to `--sync-dirs` as a list of lists of the form

```
--sync-dirs "[['path on host', 'absolute path on VM'], ['second path on host',
↪'second absolute path on VM']]"
```

This list of lists must

- be able to be evaluated by Python and Ruby as a list of lists of strings,
- specify target paths with absolute paths

Since this is rather cumbersome to pass in the CLI, remember that it can also be set in the *configuration file*, like

```
[up]
sync_dirs = [['path on host', 'absolute path on VM'], ['second path on host', 'second␣
↪absolute path on VM']]
```

1.6.3 Command Provisioning

Rambo is able to provision with a command. This command can be passed to `rambo up` in the cli, or set in the `rambo.conf`. For example:

```
rambo up -c "hostname"
```

will provision and run the command `hostname`, displaying the hostname of the instance as part of the provisioning process.

This command is intended to be the entry point to any user-controlled way of provisioning, and easily used with other synced code. For instance, this command could run a script, or invoke configuration management tools like Salt or Puppet.

For example, this setup will run a command, that calls a custom script that installs Salt and runs a highstate. This example works as-is with the basic Salt setup that Rambo provides in a new project.

```
# rambo.conf

[up]
provider = virtualbox
box = ubuntu/bionic64
sync_dirs = [["saltstack/etc", "/etc/salt"], ["saltstack/srv", "/srv"]]
command = bash /vagrant/provision.sh
```

```
# provision.sh
if  [ ! -f "bootstrap.sh" ]; then
    echo "Updating system and installing curl"
    apt update
    apt install curl -y

    echo "Downloading Salt Bootstrap"
    curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
fi

echo "Installing Salt with master and Python 3"
bash bootstrap-salt.sh -M -x python3

echo "Accepting the local minion's key"
salt-key -A -y


# Is Salt ready yet? Proceed once it is.
salt \* test.ping --force-color
while [ $? -ne 0 ]
do
    echo "Waiting for Salt to be up. Testing again."
    salt \* test.ping --force-color
done
```

(continues on next page)

(continued from previous page)

```
echo "Running highstate. Waiting..."
salt \* state.highstate --force-color
```

That script can then be used with `rambo up`.

Alternatively, the command can be passed in the CLI, like

```
rambo up -c 'bash /vagrant/provision.sh'
```

Note that when this is done, keep in mind that double quotes may use shell expansion. So if

```
rambo up -c "echo $PWD"
```

is used with bash, the working directory *of the host* will be echoed.

1.7 AWS EC2

1.7.1 Create Account

After you installed the dependencies on your host computer you now need to create an account at AWS. This repo will create real resources on AWS so you need to provide AWS with valid payment and remember you might rack up a bill if you run a whole bunch of machines. You have been warned.

1.7.2 Create SSH Keys

Next you need to create a SSH key pair for AWS.

Run:

```
mkdir -p auth/keys
cd auth/keys
ssh-keygen -t rsa -N '' -f "aws.pem"
```

*If you want multiple users or computers to access the same AWS profile or team, you must have unique key names. For example, you will need to change the base name of your `.pem` and `.pem.pub` files to something else like `aws-myname`.

Create a new key and name the key the same as the base name of your SSH key. If AWS's key name and the one one your host don't match, you won't communicate to your VM.

Now go to AWS's "EC2 Dashboard", on the left hand side go to "Key Pairs" and click the "Import Key Pair" button.

Here are instructions on how to setup SSH keys with aws:

http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html

You will need to copy the contents of `aws.pem.pub`.

NOTE: You need the aws.pem file and the aws.pem.pub file. The aws.pem file needs permissions set to 600 and The aws.pem.pub file needs permissions set to 644

Be careful not to commit this file to the repo. We setup this repo to ignore all files ending in `.pem`. But, you could theoretically still commit the pem file (by forcing a commit for example). Store this pem file in a safe place with restricted access. Anyone who has this file can log into your machines on AWS and run arbitrary commands on them.

1.7.3 Create Security Group

Now go to AWS's "EC2 Dashboard", on the left hand side go to "Security Group" and click the "Create Security Group" button.

Name the new security group **salted_server**.

Add these inbound rules to the security group

```
"All ICMP - IPv4", ICMP, 0 - 65535, anywhere

"SSH", TCP, 22, anywhere

"HTTP", TCP, 80, anywhere

"HTTPS", TCP, 443, anywhere

"Custom TCP Rule", TCP, 4505, anywhere

"Custom TCP Rule", TCP, 4506, anywhere

"Custom TCP Rule", TCP, 5000, anywhere

"Custom TCP Rule", TCP, 8080, anywhere

"Custom TCP Rule", TCP, 8888, anywhere
```

1.7.4 Create API Token

Next you need to manually create an API access token on AWS.

Go to the "IAM Dashboard", then go to "users", now click on the user who will be creating the AWS EC2 instances. Click on the "Security Credectials" tab, click the "create access key" button.

You MUST get both the **Access key ID** and the **Secret access key**.

NOTE: AWS will only show you this key ONCE.

1.7.5 Create SSH Key

Now you need to create or upload an ssh key. This key is not your user's general key in IAM, but the one in the EC2 -> Key Pairs section.

1.7.6 Edit Script to Load Environment Variables

Here is the contents of the aws.env.sh file. Edit it by replacing the placeholder tags with your keys and tokens.

```
#!/bin/bash

# for aws
# associated with your aws user
export AWS_ACCESS_KEY_ID=<YOUR AWS KEY ID>
export AWS_SECRET_ACCESS_KEY=<YOUR AWS ACCESS KEY>
# associated with ec2 specifically (not your user's general ssh key)
export AWS_KEYPAIR_NAME="name"
export AWS_SSH_PRIVKEY="auth/keys/name.pem"
```

Put your aws access key token in the line. `export AWS_ACCESS_KEY_ID=<YOUR AWS KEY ID>`

Put your aws secret acces key token in the line: `export AWS_SECRET_ACCESS_KEY=<YOUR AWS ACCESS KEY>`

Put the **name** of your aws ssh private key in the line: `export AWS_KEYPAIR_NAME="name"`

Put the **path** to your aws ssh private key in the line: `export AWS_SSH_PRIVKEY="auth/keys/name.pem"`

After editing, your aws.env.sh file will look similar to this:

```
#!/bin/bash

# for aws
# associated with your aws user
export AWS_ACCESS_KEY_ID="AKIAITT673DAF4YNV7MA"
export AWS_SECRET_ACCESS_KEY="m25AyjXtiYB2cCWMv1vQeyZtWqiWg0nqxi2Wm2QX"
# associated with ec2 specifically (not your user's general ssh key)
export AWS_KEYPAIR_NAME="name"
export AWS_SSH_PRIVKEY="auth/keys/name.pem"
```

Note: the public key must be in the same dir as the private key and the public key must share the same base name as the private key (just append ".pub" on the public key file's name).

Now you need to source the aws.env.sh file. cd into the repo and run:

```
source aws.env.sh
```

1.7.7 Launching Your AWS EC2 Instance

Finally, run:

```
rambo up -p ec2
rambo ssh
```

1.8 DigitalOcean

1.8.1 Create Account

After you installed the dependencies on your host computer you now need to create an account at DigitalOcean.

This repo will create real resources on DigitalOcean so you need to provide DigitalOcean with valid payment and remember you might rack up a bill if you run a whole bunch of machines. You have been warned.

1.8.2 Create SSH Keys

Next you need to create a SSH key pair for DigitalOcean.

Run*:

```
mkdir -p auth/keys
cd auth/keys
ssh-keygen -t rsa -N '' -f "digitalocean.pem"
```

*If you want multiple users or computers to access the same DigitalOcean profile or team, you must have unique key names. For example, you will need to change the base name of your `.pem` and `.pem.pub` files to something else like `digitalocean-myname`.

Now go to https://cloud.digitalocean.com/settings/security

Create a new key and name the key the same as the base name of your SSH key. If DigitalOcean's key name and the one one your host don't match, you won't communicate to your VM.

Copy the contents of `digitalocean.pem.pub` into the new key field.

Here are instructions on how to setup SSH keys with DigitalOcean:

https://www.digitalocean.com/community/tutorials/how-to-use-ssh-keys-with-digitalocean-droplets

You will need to copy the contents of `digitalocean.pem.pub`.

NOTE: You need the `digitalocean.pem` file and the `digitalocean.pem.pub` file. The `digitalocean.pem` file needs permissions set to 600 and The `digitalocean.pem.pub` file needs permissions set to 644

Be careful not to commit this file to the repo. We setup this repo to ignore all files ending in `.pem`. But, you could theoretically still commit the pem file (by forcing a commit for example). Store this pem file in a safe place with restricted access. Anyone who has this file can log into your machines on DigitalOcean and run arbitrary commands on them.

1.8.3 Create API Token

Next you need to manually create an API access token on digitalocean.com

Go to:

https://cloud.digitalocean.com/settings/api/

NOTE: DigitalOcean will only show you this key ONCE.

Store this token in a safe place with restricted access. Anyone who has this token can create, edit, or destroy resources on digital ocean, they could rack up a huge bill for you or shut down all your vms.

1.8.4 Edit Script to Load Environment Variables

Here is the contents of the `digitalocean.env.sh` file. Edit it by replacing the placeholder tags with your key and token.

```
#!/bin/bash

# for digitalocean
export DIGITALOCEAN_TOKEN=<YOUR DIGITALOCEAN API TOKEN>
export DIGITALOCEAN_PRIVATE_KEY_PATH="auth/keys/digitalocean.pem"
```

Put your DigitalOcean API token in the line: `export DIGITALOCEAN_TOKEN=<YOUR DIGITALOCEAN API TOKEN>`

Put the **path** to your DigitalOcean ssh private key in the line: `export DIGITALOCEAN_PRIVATE_KEY_PATH=<PATH TO YOUR PRIVATE KEY>`

After editing, your `digitalocean.env.sh` file will look similar to this:

```
#!/bin/bash

# for digitalocean
export DIGITALOCEAN_TOKEN=
↪"0bf1d884e737417e2ea6f7a29c6035752bf8c31b366489c5366745dad62a8132"
export DIGITALOCEAN_PRIVATE_KEY_PATH="auth/keys/digitalocean.pem"
```

Note: the public key must be in the same dir as the private key and the public key must share the same base name as the private key (just append ".pub" on the public key file's name)

Now you need to source the `digitalocean.env.sh` file. cd into the repo and run:

```
source digitalocean.env.sh
```

1.8.5 Launching Your DigitalOcean Instance

Finally, run:

```
rambo up -p digitalocean
rambo ssh
```

1.9 Docker

At this time Docker is supported by using an intermediate host. The intermediate host is created with VirtualBox, and is the same OS as is used for the Docker container inside it. Setting both of those OSes to be the same avoids certain complexities and potential problems that would otherwise present.

Basic usage of Docker:

```
rambo up -p docker
rambo ssh
```

1.10 LXC

NOTE: At this time, this will only work on Ubuntu 16.04+ host OS

At this time LXC is supported natively on Ubuntu 16.04. For this native support, you need a few additional dependencies. They can all be installed with this:

```
sudo apt install -y build-essential linux-headers-$(uname -r) lxc lxc-templates␣
↪cgroup-lite redir
```

After that, starting an LXC container with basic usage is:

```
rambo up -p lxc
rambo ssh
```

Note: At this time using LXC as a provider will require root priveleges / sudo.

1.11 Changelog

1.11.1 TBD

FEATURES:

- Add option to set path to sync into VM.
- Add option to provision from script with path.
- Add option to start provisioning with command.
- Add option to set VM name.
- Add option to set hostname.
- Add option to set cpus in virtualbox.
- Add option to set sync_type and defauting to Vagrant's default.
- Add option to provision with Salt.
- Add option to set salt-bootstrap args.
- Add option to sync all listed dirs.
- Add optional setting overrides with my_rambo.conf.
- Add option to resize base VirtualBox drive size with vagrant plugin.
- Add option to set ports that are forwarded.

IMPROVEMENTS:

- `rambo destroy` will now use vboxmanage to fully poweroff and delete VirtualBox VMs.
- No longer using any custom sources.list.
- Renamed sync_dir to project_dir.
- Now you can pass fancy pathing like `..` and symlinks to the CLI.
- More comprehensive logging.
- Splitting expected saltstack dir into saltstack/etc and saltstack/srv to more easily work with the common pattern of having Salt code in /etc/salt and /srv.
- Left legacy style salt provisioning, but added a flag to use it instead of the newer style.

BUGFIXES:

- Fix bug incorrectly setting cwd, leading to nested temp dirs.
- Passes provider explicitly as cmd arg to Vagrant.
- Fixes guest hostname generation when given underscores in the path, casting it to "95", it's ascii code.
- Fixes guest hostname generation when too long, truncating the part preceding the hash so total length stays below 64 chars.
- Fix bug when setting machine-type.
- Fix ability to set cwd.

1.11.2 0.4.4 (March 9, 2018)

BUGFIX:

- Now custom fork of click_configfile is added as a submodule so it's always present, and included this in the MANIFEST.

1.11.3 0.4.0 (March 9, 2018)

FEATURES:

- Added Ubuntu Dockerfile.
- Added machine-type option for various cloud providers.
- Added ramsize and drivesize options.
- Added ability to load options via rambo.conf.
- Added `createproject` cmd to create project dir, including default saltstack code, auth dir, and mandatory rambo.conf.
- Added `install-plugins` cmd for installing vagrant plugins.
- Refactored shell invocation of Vagrant to ouput near real-time stdout and stderr, keeps ANSI formatting, stderr and exit status passthrough.
- Better logging of shell invocation of Vagrant.
- Added the ability to custom saltstack code dir that is automatically used.
- Added the ability to custom Vagrantfile that is automatically used.
- Added export-vagrant-conf cmd for dropping vagrant files for customization.
- Added option for guest os.

IMPROVEMENTS:

- Rounded out OS whitelist: Debian 8/9, CentOS 7, Ubuntu 14.04/16.04
- Fix Docker bugs
- Added readthedocs.
- Defining a project by the existence of a conf file.
- Remove support for using env vars and api simultaneously.
- If no saltstack dir is in a the project dir, no salt is run.
- Saltstack files moved to terminal-labs/sample-states repo. The 'basic' branch is pulled and used.
- Added in vagrantfile setting kind of syncing.
- Added toggle in vagrantfile for grabbing canonical apt sources or not since some images come with different, unreliable sources.
- Renamed `vagrant_resources` to `vagrant`, and `salt_resources` to `saltstack`.
- Changed default guest os from Debian 8 to Ubuntu 16.04.

1.11.4 0.3.3 (November 28, 2017)

FEATURES:

- Added ubuntu 14.04 to hosts list.

IMPROVEMENTS:

- Add version cli option.

1.11.5 0.3.2 (November 27, 2017)

FEATURES:

- Added additional Salt states for Hadoop edgenode and worker.
- Allowing setting custom tmpdir path.

IMPROVEMENTS:

- Using standard get/set_env_var_rb functions.
- Change VM_SIZE to RAMBO_RAM and created RAMBO_DRIVESIZE.
- Changed VM_Host to RAMBO_GUEST_OS.
- Cleaned up some Vagrant code.

BUG FIXES:

- Stop setting a default apt source on CentOS.
- Stop setting hostname on CentOS. Another ticket was made for that.
- Passing ctx to ssh and destroy commands.
- Changed name of base box according to the box name change on app.vagrantup.com for the default Debian box.

1.11.6 0.3.1 (November 8, 2017)

FEATURES:

- Now AWS makes use of VM_Size flag to produce t2.nano, t2.micro, and t2.small VMs.

IMPROVEMENTS:

- Updated docs for CLI, Python API, Environment Variables
- Renamed tmp dir to rambo-tmp.

BUG FIXES:

- `rambo destroy` now finds and deletes metadata in tmp dir.
- Fix Docker failing on editing non-existant bashrc. Now ensuring existence first.
- Fixing vagrant up exit trigger when VM not named 'default'.
- Fixed bug preventing provisioning without Salt.

1.11.7 0.3.0 (October 26, 2017)

FEATURES:

- Added Salt states to apply Anaconda licenses.
- Adding a Python API.
- Added Nano to base Salt provisioning.
- Able to set Vagrant environment variables via the CLI
- Refactored packaging for PyPI.
- Added in support for Ubuntu 14.04 and Centos 7 guest OSs.
- Added in 4GB and 8GB RAM for all supported OSs.
- Added Salt states for setting up licensed Anaconda.
- Made Rambo a pip installable package.
- Created a Python based CLI for Rambo.
- Added support for multiple users on DigitalOcean.
- Added a Salt state for Hadoop Ambari.
- Added basic network modifications for clustering.
- Setting the hostname to the VM_NAME.

IMPROVEMENTS:

- Now downloading base vagrant boxes from vagrantup.com.
- Now enforcing Vagrant >=1.9.7.
- VM_NAME now contains host's hostname, and rambo's working dir, and a unique hex id.
- Now deletes broken symlinks found that would otherwise break Rambo during the rsync process.

BUG FIXES:

- Fix ability to set repository branch and then execute highstate.

1.11.8 0.2.1 (August 9, 2017)

FEATURES:

- Now activating conda environment upon `vagrant ssh`.
- Added Salt State for Anaconda.
- Added Salt State for loading a database dump from a local store, and allowing using this or the artifacts state.

IMPROVEMENTS:

- Added default fingerprints for BitBucket and GitHub.
- Renamed miniconda state to conda.
- Added documentation for Docker provider.

BUG FIXES:

- Fixed misnamed reference to Miniconda state.

- Now requiring artifacts grains before trying to load a database.
- Deduping state IDs for adding fingerprints for git and hg.
- Specifying fingerprint hash type since that's now required by a Salt update.
- Deduping state IDs for installing pip requirements with conda and venv.
- Removing unused salt state directory.
- Bumped required vagrant version.

1.11.9 0.2.0 (August 3, 2017)

FEATURES:

- Added a Salt State for Miniconda.
- Added Docker as a provider.

IMPROVEMENTS:

- Using Packer made base boxes for VirtualBox.
- Now using paravirtualization with VirtualBox for increased speed.
- Enhanced documentation and helper markdown files.
- Renamed 'AWS' provider to 'EC2' to avoid future confusion.
- Updated documentation.
- Some code cleaning.

BUG FIXES:

- Changed the standard AWS EC2 size to t2.micron.

1.11.10 0.1.0 (May 22, 2017)

FEATURES:

- Initial commit.

The changelog began with open sourcing Rambo at version 0.1.0.

CHAPTER 2

What's Rambo For?

This project is for provisioning and configuring virtual machines (and containers) in a simple, predictable, and highly reproducible way. Just run one command and your VM is up, code is deployed, and your app is running, on any supported platform.

At this time this repo allows you to create a Linux VM on multiple providers (AWS EC2, DigitalOcean, VirtualBox, LXC). Several Operating Systems are available on select providers. The base machine configuration is a Ubuntu 16.04 64bit OS with 1024MB RAM, and 30GB drive.

One of the goals of this project is be able to run a simple command and have a new VM be created on your provider of choice. Once the VM is initialized a provisioner is used to deploy code to and provision your machine. The provisioning configuration code will run the same regardless of which provider is actually running the machine. You can easily cycle your VMs by destroying and rebuilding them.

Another goal of this repo is to have the spawned VMs be maximally similar across providers. Usually, your configuration will not need to change at all and will simply run on all providers.

By default Rambo offers a basic VM configuration with [SaltStack](https://github.com/saltstack/salt/), but you can customize this. See Customizing Rambo for that.

CHAPTER 3

Basic Usage

Once installed, you can run one of these commands to get your VM:

for VirtualBox run

```
$ rambo up
$ rambo ssh
```

for AWS EC2 run

```
$ rambo up -p ec2
$ rambo ssh
```

for DigitalOcean run

```
$ rambo up -p digitalocean
$ rambo ssh
```

for Docker run

```
$ rambo up -p docker
$ rambo ssh
```

for LXC run

```
$ rambo up -p lxc
$ rambo ssh
```

CHAPTER 4

Contributing

We heartily welcome any contirubtions to this project, whether in the form of commenting on or posting an issue, or development. If you would like to submit a pull request, you might first want to look at our development guidelines for this project.

CHAPTER 5

Special Thanks

Thanks go out to the Vagrant community and HashiCorp. Vagrant is a great tool it has helped us a lot over the years.

Rambo is supported by Terminal Labs.