Using Salt and Vagrant for Rapid Development

by Peter Meulbroek | Feb 13, 2019

One of our main jobs at Risk Focus is to work closely with our clients to integrate complex tools into their environments, and we use a plethora of technologies to achieve our clients' goals. We are constantly learning, adopting, and mastering new applications and solutions, and we find ourselves constantly creating demos and proofs of concept (PoCs) to demonstrate new configurations, methods, and tools.

We use a variety of applications in our deliverables but often rely on Salt for the foundation of our solutions. Salt is amazing: it combines a powerful declarative language that can be easily extended in Python, a set of components that supports a diverse array of use cases from configuration and orchestration to automation and healing, and a strong supportive community of practitioners and users.

In my own work I'm often off the grid, traveling to clients, at a client site, or in situations where a lack of internet access precludes me from doing work in the public cloud. In these situations, I rely on technology that allows me to experiment or demonstrate some of the key concepts in DevOps from the laptop. Towards this end, I do a lot of this work using Vagrant by Hashicorp. I love Vagrant. It's a fantastic platform to quickly create experimental environments to test distributed applications. Vagrant's DSL is based on Ruby, which fits in well with my past developer experience. Finally, Vagrant is easily extensible: it delegates much of the work of provisioning to external components and custom plugins and comes with Salt integration out of the box.

After working with Salt and Vagrant on a set of demos, I've decided to share some of the tools I put together to improve and extend this basic integration. The tools are aimed at two goals: faster development of new environments and environmental validation. In pursuit of the first goal, this post is about host provisioning and integrating Salt states. In pursuit of the second, I will post a follow-up post to describe how to generate ServerSpec tests to validate newly created hosts. These tools offer a quick path to creating and validating virtual infrastructure using Salt.

Background

The ironic aspect of using Vagrant is that, although it is implemented in Ruby and the central configuration file (the Vagrantfile) is implemented in Ruby, the actual implementation of Vagrantfiles is un-Ruby-esque. The syntax can be ugly, validation of the file is difficult, and the naïve implementation is not DRY*. However, with a bit of coding, the ugliness can be overcome.

I've put together a set of provisioning configuration helper classes written in Ruby that allow me to more succinctly define the configuration of a test cluster and share code between projects. The idea behind the classes is very simple: extract from the Vagrantfile all of the ugly and repetitive assignments so that creating a simple Salt-ified platform is trivial.

The code for this post is found at https://github.com/riskfocus/vagrant-salt. Readers are encouraged to check it out.

* Don't Repeat Yourself

TL;DR

The post assumes you have Vagrant installed on your local machine, along with a suitable virtualization engine (such as Oracle's VirtualBox).
1.) In your vagrant project directory, check out the code

```
git clone https://github.com/riskfocus/vagrant-salt
```

2.) Copy the configuration file (vagrant-salt/saltconfig.yml.example) to the vagrant project directory as saltconfig.yml

3.) Copy the sample vagrant file (vagrant-salt/Vagrantfile.example) to the vagrant project directory as Vagrantfile (replacing the default), or update your Vagrantfile appropriately

4.) Set up default locations for Pillars and Salt States. From the Vagrant project directory:

```
./vagrant-salt/bin/bootstrap.sh
```

5.) Initialize the test machine(s)

```
vagrant up
```

Congratulations. You now have an example Salt cluster with one minion and one master. The bootstrap script has also created a simple Salt layout, complete with pillar and state directories and top files for both.

Uses

The examples directory contains a few sample configuration files that can be used to explore Salt. These are listed in vagrant-salt/examples/topology and include:

- One master, two minions
- One master, one syndic, two minions

To use either of these topologies, copy the example Vagrantfile and saltconfig.yml to the Vagrant project directory and follow instructions 2-5, above.

Deep Background: The Classes

The development necessary to get Vagrant and Salt to work together seamlessly is key management and configuration – creating and installing a unique set of keys per cluster to avoid reuse of keys or potential vulnerability.

The code consists of a factory class to create the configuration for Salt-controlled hosts and a set of classes that represent each type of Salt host (minion, syndic, master). Each class loosely follows the adapter pattern.

When put into action, the factory class takes a hash that specifies the hosts to be created. Each host created is defined by a value in this hash. It is quite convenient to use a yaml file to initialize this hash, and all examples given below list the configuration in yaml. Example yaml configurations are provided in the code distribution.

There are three classes that can be instantialized to create configuration objects for each of the Salt types. All configuration objects need two specific pieces of information: a hostname and IP. If a host is to be included in Salt topology, the configuration must include the name of its Salt master.

- The base for “salt-ified” hosts is the minion class, which corresponds to a host running a Salt minion.
- The master class corresponds to a host running a Salt minion and also the Salt master process.
- The syndic class corresponds to a Salt master that also runs the syndic process.

The Configuration Structure

The configuration structure is used by the factory class to instantiate a hash of host objects. It has three sections: defaults, roles, and hosts.
The defaults section specifies project-wide defaults, such as number of virtual CPUs per host, or memory per host, as follows:

```
defaults:
memory: 1024
cpus: 1
```

Entries in this section will be overwritten by the role- and host-specific values. This section is particularly useful for bootstrapping strings.

The roles section specifies values per-role (minion, master, syndic). This gives a location to specify role-specific configuration, such as the location of configuration files.

```
roles:
minion:
grains:saltstack/etc/minion_grains
```

The hosts section specifies the hosts to create and host-specific configuration. Each host must, at minimum, contain keys for role, IP, and the name of its master (when it has one):

```
hosts:
minion1:
ip: 1.2.3.4
role: minion
master: master
master:
ip: 1.2.3.4
role: master
master: master
```

Note that per-host values (such as memory or cpu count) can be added here to overwrite defaults.:

```
hosts:
master:
cpus: 2
memory: 1536
ip: 1.2.3.4
role: master
master: master
```

The Vagrantfile

Incorporating the configuration classes into a Vagrantfile greatly simplifies its structure. The factory class creates a hash of configuration objects. Executing the Vagrantfile iterates through this hash, creating each VM in turn. The objects store all necessary cross-references to create the desired topology without having to write it all out. The configuration objects also hide all the messy assignments associated with the default Salt implementation in Vagrant and allow the Vagrantfile to remain clean and DRY.

The file Vagrantfile.example in the distribution shows this looping structure.

Integrating Salt States

The above description shows how hosts can be bootstrapped to use Salt. Of much more interest is integrating Salt with the configuration and maintenance of that host. This integration is fairly trivial. Included in the vagrant-salt/bin directory is a bash script called “bootstrap.sh” that will create a skeleton directory for Salt states and pillars. This directory structure can be used by the Salt master(s) by including the appropriate Salt master configuration. For example, with default setup, the included Salt master configuration will incorporate those directories:

```
hosts:
master:
role: master
ip: 10.0.44.2
memory: 1536
cpus: 2
master: master
master_config:
file_roots:
base:
– /vagrant/saltstack/salt
pillar_roots:
base:
– /vagrant/saltstack/pillar
```

Conclusion

Salt is a very powerful control system for creating and managing the health of an IT ecosystem. This post shares a foundational effort that simplifies the integration of Salt within Vagrant, allowing the user to quickly test deployment and implementation strategies both locally and within the public cloud. For developers, it gives the ability to spin up a new Salt cluster, validating configuration, states, and the more advanced capabilities of Salt such as reactors, orchestrators, mines, and security audits. The classes also provide an easy way to explore Salt Enterprise edition and the visualization capabilities it delivers.

The next post in this series will focus on validation. As a preview, at Risk Focus we strongly believe in automated infrastructure validation. In the cloud or within container management frameworks, addressable APIs for all aspects of the environment mean that unit, regression, integration, and performance testing of the infrastructure is all automatable. The framework described in this post also includes a test generation model, to quickly set up an automated test framework for the infrastructure. Such testing allows for rapid development and can be moved to external environments. In the next post, we’ll go through using automatically generated, automated tests for Salt and Vagrant.