Ansible

(and a bit of general IT automation and also a tiny bit of Vagrant, but mostly Ansible)

2014-11-26

Lars Tiede

Ansible?

- A fictitious machine capable of instantaneous or superluminal communication (Wikipedia)

- An open-source software platform for configuring and managing computers (also Wikipedia)

Program

- Introduction to some basic Ansible and general IT automation ideas
  - run examples on a demo infrastructure with local VMs provisioned by Vagrant
- Ansible abstractions (“task”, “role”, etc.)
- Automated deployment of a simple web app on the demo infrastructure
  - Demonstrate usefulness of *idempotent* actions in IT automation
- Ansible specialties (vs competitors)
- My experience: what I like and don’t like
- Challenge: Devs vs Ops vs DevOps in the clouds

Demo infrastructure

- 3 VMs (“control”, “target1”, “target2”) on my laptop, all set up with Vagrant
  - all VMs start off as vanilla Ubuntu 14.04, just a tiny bit of config
    - static IP addresses, entries in /etc/hosts (we want to use hostnames, but we don’t want to set up DNS)
    - “vagrant” user can SSH without password from “control” machine to the others
- “Vagrantfile” on source.uit.no, together with this talk
  - You can replicate the demo setup on your Linux/Mac/Windows machine in *seconds* (almost)

Some ideas (Ansible-specific and general CM)

- Require only SSH and Python to do anything anywhere
  - No agents / daemons (as in Chef or Puppet)
- Describe first and foremost the state you want your target machines to be in, not so much how to get there
  - Not “infrastructure as code”, but “infrastructure as data”
  - This is not enforced. You may break the rules.
- Make re-usable descriptions of smaller aspects of your systems that you can then combine in endless ways

Ansible abstractions (1)

- **Module**: lets you describe a state for a specific thing. Behind the scenes, a module implements the "how to get there" bits. Ansible ships hundreds of modules, you can also write your own (in any language).
  - Example: "apt" module lets you define that a certain package should be present (or absent)
  - Example for potentially breaking the rules: "shell" module lets you run any shell command
- **Task**: "Invocation of a module". Example: "apt package nginx is installed". Ansible runs tasks on target hosts.
- **Inventory**: file describing target hosts, groups of target hosts, and their properties.

Ansible abstractions (2)

- **Playbook**: a file containing sequences of tasks ("plays"), and "role invocations".

- **Role**: a directory in the file system. Contains a sequence of tasks, associated files, templates, variables, and dependencies to other roles.

    - Useful concept for modularizing and re-using bits and pieces.

Demo App Deployment

- We have a simple “hello world” web app written in Python, using the Bottle web framework
- deployment setup: N app servers behind 1 load balancer
- Two Ansible roles: “runs web app” and “runs load balancer”
  - web app role installs dependencies and installs and runs web app on all machines the role is applied to
  - load balancer role installs, configures, and runs a simple nginx load balancer on all machines the role is applied to
    - variables are passed into the role for configuration
    - templates and in general most text you write in Ansible can use these variables (the jinja2 templating engine comes bundled with Ansible)

Idempotence

- “is the property of a certain operation that can be applied multiple times without changing the result beyond the initial application.” (Wikipedia)
- “Turn over a glass of water to empty it” is idempotent but pointless when the glass is already empty. “Pour water into glass” is not idempotent and you might get wet if the glass is already full. “Do what is necessary to have a glass full of water” is idempotent, never pointless, and always safe, but requires intelligence. Ansible ticks like that:
  - “water-in-glass” module code encapsulates the “do what is necessary”
  - module invocation only lets you determine whether you want the glass full or empty
- Idempotence is *good*. You can run an idempotent playbook any number of times and the final state of the system will always be the same (at least with regards to the playbook).
- Idempotence is *not enforced*. You can easily write tasks that are not idempotent.

Ansible specialties (1)

- Most of the below is motivated by “lessons learned” from experiences with Puppet and Chef (author worked at PuppetLabs before writing Ansible)

- Ansible is agent-less, all is done with SSH

- PKI comes for free, can get started right away: no bootstrapping needed, no extra security audits because SSH is most likely already there

- no agents -> push-only based model: sounds limited, but actually the more complicated things you do such as rolling updates and multi-node orchestration are arguably best done with “push”. Why use a more complex pull setup if push can do everything you need?

Ansible specialties (2)

- The language you describe things in should be dead simple and not be a full-fledged programming language. It should also be human readable: YAML playbooks
  - ordered, but only few programming features so that playbooks are easy to understand (“ordered like Chef but declarative like Puppet”)
  - no actual programming language like in Chef (Ruby), no DSL like in Puppet
- Easy extensibility: custom modules can be written in any language (anything that can produce JSON on stdout is fine)
- Versatility: deployment and orchestration of your app, configuration management on your nodes, one-off commands - all doable with Ansible

“I wrote Ansible because none of the existing tools fit my brain. I wanted a tool that I could not use for 6 months, come back later, and still remember how it worked.”

Michael DeHaan, Ansible project founder

From my experience: what I like

- Describe infrastructure, deployment, even things like rolling updates and continuous delivery right alongside your code - “the Makefile for the cloud”?
    - Even better: this Makefile can be read and understood by humans!
- All you write is simple plain text: can put playbooks and roles in source control -> versioned, shareable descriptions of infrastructure.
    - Ansible Galaxy: online repository containing thousands of user-created roles
- “Infrastructure as data” in YAML makes me yearn for just writing a monster shell script instead sometimes, but it *does* help to produce re-usable, modularized, often even readable descriptions.
    - The module and role abstractions are immensely useful.
- I connect my laptop to the network and I can use Ansible to automate stuff right away if I only have SSH access. You can do the same. It doesn’t get easier than that. Simple, decentralized and agent-less architecture <3

From my experience: what I don't like

- "Infrastructure as data" seems impossible to attain in practice and might therefore be a flawed paradigm (at least in Ansible)
  - A playbook contains sequences of tasks: that's already code
  - Not having control structures (loops, if/else) in playbooks but instead being able to flag tasks with "when: <boolean expression>" is halfway re-inventing control structures, just poorly
  - The next major version of Ansible will introduce "blocks" in playbooks…
  - *There's an upside, though: the lack of sophisticated control structures forces you to write simple playbooks. With that, plus having a bunch of good and powerful modules, you can get very far, maybe far enough.*
- YAML can get ugly (but at least it's simple)
- Occasional confusion: "src" parameter refers to local path in module X, but path on target host in module Y

Challenge: Devs vs Ops vs DevOps

- The use of clouds as deployment platforms and the use “agile” methods for software development/release increase interdependencies and blur the lines between Devs and Ops
    - (continuously) deploying an app, possibly consisting of many orchestrated services in the cloud is hard to burden on SysOps (alone). As a consequence, developers are now often supposed to deploy and run their stuff themselves - current buzzwords: “full stack developer”, “DevOps”
    - developers and sysops must work tightly together on all matters automation or else they might disturb each other when running automated tasks on their shared infrastructure
        - Example: that nginx config for our load balancer… Dev might overwrite file when app server landscape changes, Op might overwrite file when SSL config changes.
- How can we facilitate devs and ops not getting in each other’s ways? Can (or should) we get around making everybody “jacks of all trades, masters of none”?
- Approaches? Can clever software and deployment design that facilitates separation of devs and ops roles solve the problem? Is that easy to achieve? Do we have to force everybody to use the same automation tool(s)? Will the next generation of clouds solve all the problems?

Links

- http://www.coloandcloud.com/editorial/an-interview-with-ansible-author-michael-dehaan/ (early interview with Ansible's founder)
- https://speakerdeck.com/mpdehaan/ansible and https://speakerdeck.com/mpdehaan/ansible-pycarolinas (old presentations, but by the project founder himself - esp. the second one shows key concepts and motivation behind them nicely)
- http://www.slideshare.net/MarkPhillips16/dev-ops-cardiff?related=1 (Ansible vs Puppet vs Chef)
- http://www.bubblewrapp.com/why-we-chose-ansible-over-puppet/ (goes into differences btw Puppet and Ansible)
- http://www.slideshare.net/carlo.bonamico/infrastructure-as-data-with-ansible-for-easier-continuous-delivery (ignore the many buzzwords, a few dozen slides in it gets interesting)
- http://docs.ansible.com/guide_vagrant.html (how to use Ansible to configure Vagrant VMs - recommended for Vagrant users in order to not repeat oneself)
    - http://www.ansible.com/docker (same for Docker containers)
- http://stackoverflow.com/questions/16647069/should-i-use-vagrant-or-docker-io-for-creating-an-isolated-environment (slightly off-topic: Vagrant vs Docker - with answers from both Vagrant and Docker authors)