WHAT OPS CAN LEARN FROM DEV

WILL THAMES

17 JULY 2015

OVERVIEW

ABOUT THIS TALK

- Complement to last year's talk Designing and Developing Software for Operations
- Practices and principles for Operations to write, share and rely on code.
- Configuration/Deployment/Orchestration focus.
- So Ansible/Chef/Puppet etc. but should apply to any code used in Ops.
- http://willthames.github.io/devopsdays2015/

WHY?

- Developers typically have sophisticated practices for writing and maintaining large codebases.
- Operations typically aren't as well versed in these practices.
- More people can use and contribute code when it's easy to access and easy to improve.

ABOUT ME

- Systems Engineer at Red Hat, Brisbane.
- Previously at Suncorp, Brisbane and Betfair, London.
- Contributor to Ansible.
- But this talk is intended to be product agnostic.

WRITING SOLID CODE

HIGHER LEVEL LANGUAGES

- Why use configuration management? Surely bash scripts in an for loop over ssh will suffice?
- Why use python or ruby? Surely assembly or C will suffice?

HIGHER LEVEL AUTOMATION

- Abstraction of patterns to higher layers
- Repeatability
- Error handling
- Reduction of boilerplate code
- Templating
- API calls

ABSTRACTION

- As with functions, modules, libraries and packages, wrap up common operations into reusable code. This might be a module for installing and configuring java, or deploying a particular application type.
- Chef has cookbooks, Ansible has roles and puppet has modules for grouping a bunch of operations.
- Ansible has modules and chef and puppet have providers for creating new operations.

REPEATABILITY

- What happens if you run your code twice?
- What happens if the second time is six months from now?

VERSIONS

- Give your dependencies version identifiers.
- Specify the version of dependency in a suitable place.
- Furthermore specify versions when pulling things from yum, apt-get, git, mercurial etc.

SHARING CODE

VERSION CONTROL

- Have some. Which one is relatively unimportant.
- Find out when something was changed, and by whom.
- See what changed, and hopefully why (needs good commit messages!)
- Go back in time — revert changes, compare differences.

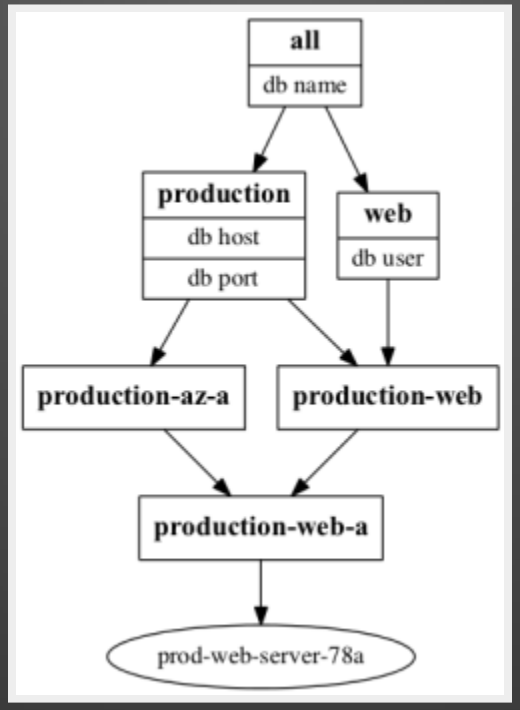
CODE SEPARATE FROM DATA

- Hardcode as little as possible in your templates and task files (beware premature templating though!)
- Should make it easier to maintain, and allows you to source configuration from alternative data sources.
- Using the same tools across all environments reduces likelihood of error.
- Try and make it so that your code could be shared with the world without giving anything away.

DATA INHERITANCE

- Only write as much configuration as you need.
- Some variables will be common to all applications across a particular environment.
- Some variables will be common to all environments for a particular application.
- Use Ansible groups, Chef roles and Puppet's profiles to manage the inheritance hierarchy.

DATA INHERITANCE



SECRETS

- You will need a solution to what to do with secrets. There are many.
- ansible-vault, chef encrypted databags, eyaml.
- Hashicorp's Vault, Keywhiz by Square.

COMMUNITY

- Separation of code and data (particularly secret data) allows you to share your work with others outside of your organisation.
- If you are able to share your code, you can include contributions of others, or set your code free so that others can manage improvements, that you can then benefit from. Opening the source is the start of the journey.

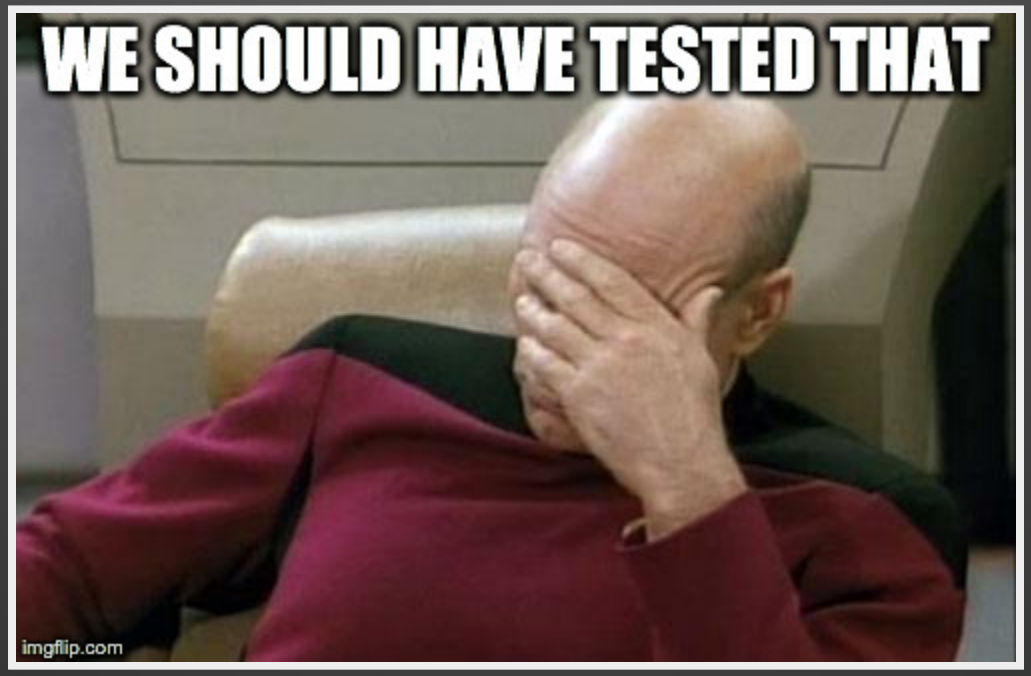
COMMUNITY

- You can also benefit from work others have done — look for modules that others have written before writing your own. They may not be perfect, but they are a start.
- See Ansible Galaxy, Puppet Forge, Chef Supermarket.

CODE QUALITY

OPS HATE TO HEAR

OPS HATE TO HEAR



AND IF OPS HATE TO HEAR THEM

They really hate to be the ones saying them.

QUALITY CONTROL

- Use the tools.
- ansible-lint, puppet-lint, Chef foodcritic.
- pep8, go fmt etc.
- dry run mode, diff mode

STANDARDS DOCUMENTATION

- Best practices are an advisory of things to consider. Call them guidelines if you prefer.
- Standards should be testable, preferably automated.
- We manage our standards and best practices as a git repo using pull requests to achieve consensus.
- Any changes/additions to best practices and standards must achieve a body of support.

CODE REVIEWS

- All code reviews should be objective. If you're objecting to a style issue, you should be able to point to documentation (internally or using an existing style guide for a language/framework)
- Have a policy on what level of consensus is required to accept code into the mainline codebase.
- This will typically be a risk management tradeoff.

TESTING

- Practices such as unit testing and integration testing are currently difficult to achieve.
- Which leaves end-to-end testing in production like environments.
- Virtual machines — RHEV, VMWare, Virtualbox etc.
- PaaS — Heroku, Openshift etc.
- Containers.
- Public cloud (AWS, Azure, GCE etc) and private cloud (Openstack)
- Anything that isn't "your machine"

CONTINUOUS INTEGRATION

- Commit
- Checkout
- Static analysis
- Automated provisioning
- Apply configuration
- Run test suite (e.g. serverspec)
- Deploy to production

DISCLOSURE

- I've yet to see the full implementation of the previous slide in practice.
- Focus on the things that are most likely to eliminate unnecessary errors or effort.

THANKS FOR LISTENING!

QUESTIONS?