Test Driving Your Infrastructure

...

Who am I?

- Systems Engineer at DHI
- Developer with ~10 years of experience using a lot of different languages/technologies

Feel free to contact me on twitter: @jessiepuls

What will we talk about?

- We'll talk about the problems we've had and how we're working on solving them
- We'll go over the workflow we follow to standing up and managing environments using "infrastructure as code"
- We'll talk about how we verify changes we make in our environments are implemented successfully
- And then how we guarantee they continue to be configured as expected
- Finally we'll talk about what's been successful (and probably a little about what hasn't)

Goals

- Improve application/server deployments
- Modernize infrastructure
- Continuous Integration
- Test infrastructure
- Remove Knowledge Silos

What Problems Exist?

...

The Problem: Silos of Knowledge

- Too many things are known by only one person
- If it breaks you have to call them
- People like to take vacations sometimes

The Solution: Teamwork

- All work can be done by anyone on the team
- Pair to solve problems together and have more knowledge of solutions
- Pair to keep focus on work and have a built in code review
- Create an environment where people feel ok not knowing something
- Create an environment where people feel ok not knowing something
- Seriously, create an environment where people feel ok not knowing something...

The Problem: Lack of Scalability

- You have to juggle physical servers to accommodate for new features and load
- Features in applications can only grow within the limits of aging infrastructure
- Aging hardware, very expensive to do simple upgrades

The Solution: Cloud First Strategy

- DHI as a whole went with AWS
- Infrastructure can scale with applications
- Need temporary horsepower? Cool we can solve that problem.
- Development teams are able to try new things more easily

The Problem: Infrastructure is Difficult to Manage

- Every server is just a modified clone of another one
- It's too hard to set up what we need
- Manually created resources are out-of-date

The Solution: Terraform

- Manage infrastructure as code
- Development of infrastructure can be done in modules and shared with others
- Provides a “fresh start” for environments that have been around for a long time (good or bad?)

The Problem: Everything is always broken

- Environments are unreliable and fragile
- Deployments are unreliable and fragile....
- People have been trained that nothing can be trusted
- Quick and dirty fixes are done a lot just to get things working (and then they stick)

The Solution: Tests & Metrics

- Testing allows us to make sure after we modify things they are in the expected state
- Testing our modules allow us to ensure they continue to work for all use cases past and present
- Tests act as documentation for expected behaviors
- Monitoring allows us to make sure this is always true (and have data to track when it isn't)
- Continuous integration (A much bigger discussion)

The Problem: Changes are made manually

- Too much human error
- Too much work
- Things don’t always work the same on all servers in an environment
- Manual changes lead to increased time debugging different issues found in multiple places

The Solution: Puppet

- Changes are made through configuration, committed to git, and merged through environments
- Results are reproducible and consistent (can be used to rebuild quickly)
- The only opportunity for human error is when configuration is created (it's either right or wrong consistently across all servers in an environment)
- You only have to make a change in one place

The Problem: Everything Takes Too Long

- No automated testing of anything
- Puppet (where it is used) fails frequently because module updates are only verified for the problem at hand, and end up causing issues in other areas
- Too many people have to be online to do a release

The Solution: Automation

- If there is a need for more servers in environments we can have them in a matter of minutes rather than days or weeks
- ..... And devs can even open a pull request to create them themselves
- Creating environments through automation allows us to guarantee they are setup identically.
- Less time doing hotfix releases due to mismatched environments

The hope: Over time the the number of people who feel they need to be online on release night shrink

So, what is it that you do here?

...

How We Create Environments

- Terraform to define the infrastructure, and manage state
- Puppet to manage server configuration
- Abstract things that are different between environments
- Write tests to prove it worked

How We Migrate Changes Between Environments

- Environments are defined by branches
- Tests and config travel with the branches so a person never has to remember what has and has not been done

What is a Puppet Module?

Example: An apache module that will allow us to install and configure apache on a number of servers in our infrastructure

How We Configure Environments

- Puppet is used to configure servers based on what “type” of server it is
- Hiera with multiple back ends is used to control configuration differences per environment
- Mcollective is used to manage “collections” of servers
- As a part of application build and deploy processes we update these values and apply them across the applicable nodes

Now Let's Test Implementation

Example: A set of serverspec tests to verify that an apache server was configured and the correct vhosts were created

Let’s Talk About Continuous Integration and Automation

...

Now that all of our infrastructure and configuration are managed like code shouldn’t we start treating it that way?

Continuous Integration

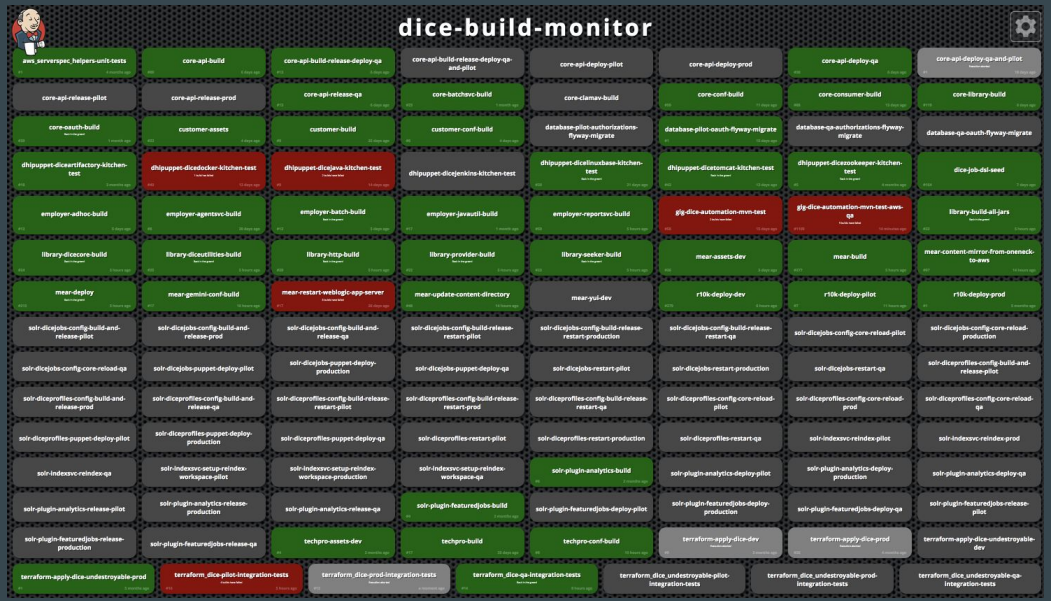
- When code is committed to a puppet module we run Test Kitchen tests using an EC2 driver to quickly discover things that only “work on my machine”
- When terraform changes are applied we automatically run integration tests in applicable environments to ensure systems are configured properly, and all expected services are running.
- Teams who work closely with key applications and components have been developing acceptance tests to verify components are operating as expected.

Automation

- We use Jenkins for the bulk of our automation
- Applications are built and deployed to artifactory using Jenkins
- Application versions “to be deployed” to an environment are managed through jenkins jobs that update hiera values
- Puppet runs are applied for groups of servers by a jenkins job that triggers Choria playbooks

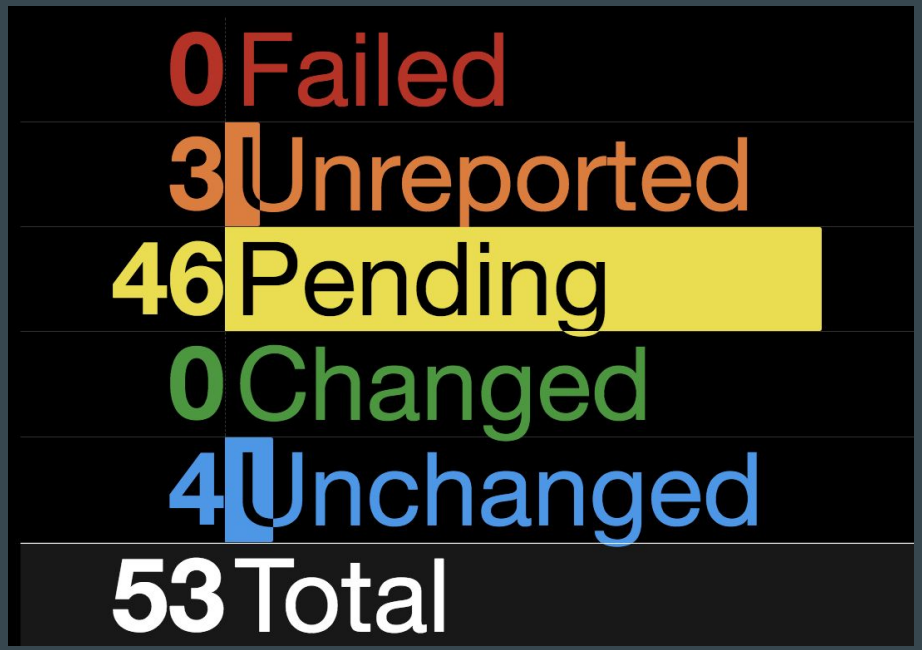
Make It Visible

All of this is reflected on our build monitor which is visible to anyone in our team space



Make It Visible

There's also something that tells us what would happen if we ran puppet on servers "right now"



So, What Lessons Have We Learned?

...

Get Everyone Involved Early

- Anyone who has a stake in the project should be involved at the level that makes sense for their role
- If we want to live in this perfect world where development and operations teams work hand in hand on projects we need to value the opinions of all of the people involved

The Team is Everything

- Make sure you have the right balance of knowledge on your team
- Try as hard as you can to create an environment where everyone feels like they can ask questions, and fail.

Be Flexible

- At nearly every step along the way we've come across things that weren't exactly as expected. You have to be able to adapt, and move forward.
- And the entire organization needs to be a part of this.

Discover Problems Quickly

- This stuff is new to us, and we experience many failures. It's best when we have them early.
- When you do find a problem talk about it

Automation is Reproducible

- Automating processes allows us to be confident that a change will be made in the same way every time it is run.
- Since everything is automated we're able to reproduce failures and resolve them more easily.
- Confidence will grow over time with consistency

Questions?

Examples and slides: https://github.com/jessiepuls/2017-09-prairiecode

Feel free to contact me: @jessiepuls