Infrastructure as Code

Managing Servers in the Cloud

Kief Morris

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Infrastructure as Code
by Kief Morris

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson
Production Editor: Kristen Brown
Copyeditor: Amanda Kersey
Proofreader: Jasmine Kwityn
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rong Tang and Rebecca Demarest

June 2016: First Edition

Revision History for the First Edition
2016-06-07: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491924358* for release details.

978-1-491-92435-8

[LSI]

Working with Infrastructure Definition Tools

Now that we've covered general guidelines for infrastructure as code tools, we can dig more deeply into the specifics. Sometimes called infrastructure orchestration tools, these tools are used to define, implement, and update IT infrastructure architecture. The infrastructure is specified in configuration definition files. The tool uses these definitions to provision, modify, or remove elements of the infrastructure so that it matches the specification. It does this by integrating with the dynamic infrastructure platform's APIs (as discussed in the previous chapter).

Examples of infrastructure definition tools that work this way include AWS Cloud Formation (*https://aws.amazon.com/cloudformation/*), HashiCorp Terraform (*https://terraform.io/*), OpenStack Heat (*https://wiki.openstack.org/wiki/Heat*), and Chef Provisioning (*https://github.com/chef/chef-provisioning*).[1]

Many teams use procedural scripts to provision infrastructure, especially those who started working this way before more standard tools emerged. These can be shell or batch scripts that call CLI tools to interact with the infrastructure platform—for example, AWS CLI (*https://aws.amazon.com/cli/*) for AWS. Or they might be written in a general-purpose language with a library to use the infrastructure platform API. These could be Ruby scripts using the Fog library (*http://fog.io*), Python scripts with the Boto (*http://boto3.readthedocs.org*) library, or Golang scripts with the AWS SDK for Go (*http://aws.amazon.com/sdk-for-go*) or Google Cloud library (*https://godoc.org/google.golang.org/cloud*).

1 As with many of the tools described in this book, infrastructure definition tools are evolving quickly, so the list of tools is likely to change by the time you read this.

Definition of "Provisioning"

"Provisioning" is a term that can be used to mean somewhat different things. In this book, provisioning is used to mean making an infrastructure element such as a server or network device ready for use. Depending on what is being provisioned, this can involve:

- Assigning resources to the element
- Instantiating the element
- Installing software onto the element
- Configuring the element
- Registering the element with infrastructure services

At the end of the provisioning process, the element is fully ready for use.

"Provisioning" is sometimes used to refer to a more narrow part of this process. For instance, Terraform and Vagrant both use it to define the callout to a server configuration tool like Chef or Puppet to configure a server after it has been created.

Provisioning Infrastructure with Procedural Scripts

On one team, we wrote our own command-line tool in Ruby to standardize our server provisioning process on Rackspace Cloud. We called the tool "spin," as in, "spin up a server." The script took a few command-line arguments so that it could be used to create a few different types of servers in different environments. An early version of the script shown in Example -1 would be run like this to create an application server in our QA environment:

```
# spin qa app
```

Our spin script included the logic for defining what a server of type "appserver" should be: what size, what starting image to use, and so on. It also used "qa" to select details about the environment the server should be added into. Whenever we needed to change the specification for a particular server type or environment, we would edit the script to make the change, and all new servers would consistently be created according to the new specification.

Script to provision a server based on type and environment

```
#!/usr/bin/env ruby

require 'aws-sdk'
require 'yaml'

usage = 'spin <qa|stage|prod> <web|app|db>'
```

```
abort("Wrong number of arguments. Usage: #{usage}") unless ARGV.size == 2
environment = ARGV[0]
server_type = ARGV[1]

#
# Set subnet_id based on the environment defined on the command line.
# The options are hardcoded in this script.
subnet_id = case environment
when 'qa'
  'subnet-12345678'
when 'stage'
  'subnet-abcdabcd'
when 'prod'
  'subnet-a1b2c3d4'
else
  abort("Unknown environment '#{environment}'. Usage: #{usage}")
end

#
# Set the AMI image ID based on the server type specified on the command line.
# Again, the options are hardcoded here.
image_id = case server_type
when 'web'
  'ami-87654321'
when 'app'
  'ami-dcbadcba'
when 'db'
  'ami-4d3c2b1a'
else
  abort("Unknown server type '#{server_type}'. Usage: #{usage}")
end

#
# Use the AWS Ruby API to create the new server
ec2 = Aws::EC2::Client.new(region: 'eu-west-1')
resp = ec2.run_instances(
  image_id: image_id,
  min_count: 1,
  max_count: 1,
  key_name: 'my-key',
  instance_type: 't2.micro',
  subnet_id: subnet_id
)

#
# Print the server details returned by the API
puts resp.data[:instances].first.to_yaml
```

This met our requirement for a repeatable, transparent process. The decisions of how to create the server—how much RAM to allocate, what OS to install, and what subnet to assign it to—were all captured in a transparent way, rather than needing us to

decide each time we created a new server. The "spin" command was simple enough to use in scripts and tools that trigger actions, such as our CI server, so it worked well as part of automated processes.

Over time, we moved the information about the server types and environments into configuration files, called *servers.yml* and *environments.yml*, respectively. This meant we needed to change the script itself less often. We only needed to make sure it was installed on the workstations or servers that needed to run it. Our focus was then on putting the right things into the configuration file, and treating those as artifacts to track, test, and promote.

Interestingly, by changing our scripts to make use of configuration files like *servers.yml* and *environments.yml*, we were moving toward declarative definitions.

Defining Infrastructure Declaratively

Splitting definitions into their own files encourages moving away from needing to think about provisioning procedurally, such as "first do X, then do Y." Instead, definition files can be declarative, such as "should be Z."

Procedural languages are useful for tasks where it's important to understand how something should be done. Declarative definitions are useful when it's more important to understand what you want. The logic of how it is done becomes the responsibility of the tool that reads the definition and applies it.

Declarative definitions are nicely suited to infrastructure configuration. You can specify how you would like things to be: the packages that should be installed, user accounts that should be defined, and files that should exist. The tool then makes the system match your specification.

You don't need to worry about the state of the system before the tool runs. The file may not exist. Or maybe it exists, but has a different owner or permissions. The tool includes all of the logic to figure out what changes need to be made, and what to leave alone.

So declarative definitions lend themselves to running idempotently. You can safely apply your definitions over and over again, without thinking about it too much. If something is changed to a system outside of the tool, applying the definition will bring it back into line, eliminating sources of configuration drift. When you need to make a change, you simply modify the definition, and then let the tooling work out what to do.

Example -2 shows a declarative Terraform configuration file that could be used instead of my earlier procedural spin script to create web servers. The definition uses a variable named `environment` to set the subnet for the servers.

Terraform configuration file

```
variable "environment" {
  type = "string"
}

variable "subnets" {
  type = "map"

  default = {
    qa = "subnet-12345678"
    stage = "subnet-abcdabcd"
    prod = "subnet-a1b2c3d4"
  }
}

resource "aws_instance" "web" {
  instance_type = "t2.micro"
  ami = "ami-87654321"
  subnet_id = "${lookup(var.subnets, var.environment)}"
}
```

The difference between this definition and the spin script is what happens when you run it multiple times with the same arguments. If you run the spin script five times specifying the same environment and the `web` server role, it will create five identical web servers. If you apply the Terraform definition five times, it will only create one web server.

Using Infrastructure Definition Tools

Most infrastructure definition tools use a command-line tool to apply a configuration definition. For example, if the example Terraform definition was saved to a file called *web_server.tf*, the command-line tool to create or update it would be run using a command like this:

```
# terraform apply -var environment=qa web_server.tf
```

Infrastructure team members can run the tool from their local workstations or laptops, but it's better to have it run unattended. Running the tool interactively is handy for testing changes to the infrastructure, by applying it to a personal sandbox instance of the infrastructure. But the definition file should then be committed to the VCS, and a CI or CD server agent should automatically apply and test the updated configuration to relevant environments. See ??? for more details on this.

Configuring Servers

An infrastructure definition tool will create servers but isn't responsible for what's on the server itself. So the definition tool declares that there are two web servers, but

does not install the web server software or configuration files onto them. The next chapter covers tools and approaches for configuring servers.[2]

But the infrastructure definition tool often needs to pass configuration information to a server configuration tool when creating a server. For example, it may specify the server's role so the configuration tool installs the relevant software and configuration. It may pass network configuration details such as DNS server addresses.

Example -3 runs Chef on the newly created web server. The `run_list` argument specifies the Chef role to apply, which Chef uses to run a certain set of cookbooks. The `attributes_json` argument passes configuration parameters to Chef in the JSON format that it prefers. In this case, it is passing an array with two IP addresses for the DNS servers. One of the Chef cookbooks that is run will presumably use these parameters to build the server's */etc/resolv.conf* file.

Passing configuration to Chef from Terraform

```
resource "aws_instance" "web" {
  instance_type = "t2.micro"
  ami = "ami-87654321"

  provisioner "chef"  {
    run_list = [ "role::web_server" ]
    attributes_json = {
      "dns_servers": [
        "192.168.100.2",
        "192.168.101.2"
      ]
    }
  }
}
```

Another way to provide configuration information to servers is through a configuration registry.

Configuration Registries

A configuration registry is a directory of information about the elements of an infrastructure. It provides a means for scripts, tools, applications, and services to find the information they need in order to manage and integrate with infrastructure. This is particularly useful with dynamic infrastructure because this information changes continuously as elements are added and removed.

2 In practice, some tools, such as Ansible, can fill the role of defining infrastructure and also define the contents of the server. But in this book, I'm describing these responsibilities separately.

For example, the registry could hold a list of the application servers in a load balanced pool. The infrastructure definition tool would add new servers to the registry when it creates them and remove them when the servers are destroyed. One tool might use this information to ensure the VIP configuration in the load balancer is up to date. Another might keep the monitoring server configuration up to date with the list of these servers.

There are different ways to implement a configuration registry. For simpler infrastructures, the configuration definition files used by the definition tool may be enough. When the tool is run, it has all of the information it needs within the configuration definitions. However, this doesn't scale very well. As the number of things managed by the definition files grows, having to apply them all at once can become a bottleneck for making changes.

There are many configuration registry products. Some examples include Zookeeper (*https://zookeeper.apache.org/*), Consul (*https://www.consul.io/*), and etcd (*https://github.com/coreos/etcd*). Many server configuration tool vendors provide their own configuration registry—for example, Chef Server, PuppetDB, and Ansible Tower. These products are designed to integrate easily with the configuration tool itself, and often with other elements such as a dashboard.

In order to work well with a dynamic infrastructure, the configuration registry service must support programmatically adding, updating, and removing entries from the registry.

Lightweight Configuration Registries

Rather than using a configuration registry server, many teams implement a lightweight configuration registry using files stored in a central, shared location, such as an object store like an AWS S3 bucket or a VCS. The files can then be made available using off-the-shelf static web hosting tools.

A variation of this is packaging configuration settings into system packages, such as a *.deb* or *.rpm* file, and pushing them to an internal APT or YUM repository. The settings can then be pulled to local servers using the normal package management tools.

These lightweight approaches for implementing a configuration registry take advantage of mature, off-the-shelf tooling, such as web servers and package management repositories. These are simple to manage, fault tolerant, and easy to scale. Registry files can be versioned in a VCS and then distributed, cached, and promoted following infrastructure as code practices. There may be some complexity in handling frequent

updates from large infrastructures, but these can often be managed by splitting or sharding registry files.

Pitfall: Tight Coupling with the Configuration Registry

A heavily used registry can become a source of tight coupling and/or brittleness. It can be unclear which scripts and services depend on a given entry in the registry. Changing the format of an entry, or removing ones that no longer seem necessary, can cause unexpected breakages. This in turn leads to hesitation to make changes to the registry, until it grows into a fragile, overly complicated mess.

For example, I once had a provisioning script add entries for web servers to the registry under the key `/environments/${environment}/web-servers/${servername}`. This was used by another script that configured load balancers to add and remove web servers from a VIP so that it matched the entries under this key. Later on, I changed these scripts to use a key under an entry for each server, like: `/servers/${servername}/environment=${environment}` and `/servers/${servername}/pool=web-servers`. I changed the load balancer configuration script to use this structure, and I changed the provisioning script to put an entry there, no longer creating the original key.

What I hadn't known was that a colleague had written a script to automatically update monitoring checks for web servers, using the original key structure. After I made my change, because the old registry keys no longer existed, the web server monitoring checks were automatically removed. None of us noticed the problem at first, as the monitoring checks simply disappeared. It was over a week later that someone noticed the missing checks, and it took nearly a day to investigate to figure out what had happened.

Good design and communication can help to avoid this kind of problem. Automated testing can help as well. Some variation of consumer-driven contract (CDC) testing (as described in ???) could have helped here. People who write scripts that make use of the registry could write simple tests that raise an alert when registry structures and formats they rely on have changed.

Is a Configuration Registry a CMDB?

The concept of a configuration management database (CMDB) pre-dates the rise of automated, dynamic infrastructure. A CMDB is a database of IT assets, referred to as configuration items (CI), and relationships between those assets. It is many ways similar to a configuration registry: they're both databases listing the stuff in an infrastructure.

But a CMDB and a configuration registry are used to address two different core problems. Although they handle overlapping concerns, they do it in very different ways. For these reasons, it's worth discussing them as separate things.

A configuration registry is designed to allow automation tools to share data about things in the infrastructure so that they can dynamically modify their configuration based on the current state of things. It needs a programmable API, and the infrastructure team should use it in a way that guarantees it is always an accurate representation of the state of the infrastructure.

CMDBs were originally created to track IT assets. What hardware, devices, and software licenses do you own, where are they used, and what are they used for? In my distant youth, I built a CMDB with a spreadsheet and later moved it into a Microsoft Access database. We managed the data by hand, and this was fine because it was the iron age, when everything was built by hand, and things didn't change very often.

So these are the two different directions that configuration registries and CMDBs come from: sharing data to support automation, and recording information about assets.

But in practice, CMDB products, especially those sold by commercial vendors, do more than just track assets. They can also discover and track details about the software and configuration of things in an infrastructure, to make sure that everything is consistent and up to date. They even use automation to do this.

An advanced CMDB can continuously scan your network, discover new and unknown devices, and automatically add them to its database. It can log into servers or have agents installed on them, so it can inventory everything on every server. It will flag issues like software that needs to be patched, user accounts that should not be installed, and out-of-date configuration files.

So a CMDB aims to address the same concerns as infrastructure as code, and they even use automation to do it. However, their approach is fundamentally different.

The CMDB Audit and Fix Antipattern

The CMDB approach to ensuring infrastructure is consistent and compliant is reactive. It emphasizes reporting which elements of the infrastructure have been incorrectly or inconsistently configured so that they can be corrected. This assumes infrastructure will be routinely misconfigured, which is common with manually driven processes.

The problem is that resolving these inconsistencies adds a constant stream of work for the team. Every time a server is created or changed, new work items are added to fix them. This is obviously wasteful and tedious.

The Infrastructure-as-Code Approach to CMDB

The infrastructure-as-code approach is to ensure that all servers are provisioned consistently to start with. Team members should not be making ad hoc decisions when they provision a new server, or when they change an existing server. Everything should be driven through the automation. If a server needs to be built differently, then the automation should be changed to capture the difference.

This doesn't address 100% of the concerns addressed by a CMDB, although it does simplify what a CMDB needs to do. Here are some guidelines for handling CMDB concerns when managing an infrastructure as code:

- Make sure everything is built by automation and so is recorded correctly and accurately.
- If you need to track assets, consider using a separate database for this and have your automation update it. This should be kept very simple.
- Your automation should accurately record and report your use of commercial software licenses in the configuration registry. A process can report on license usage, and alert when you are out of compliance, or when you have too many unused licenses.
- Use scanning to find and report on things that were not built and configured by your automation. Then either remove them (if they don't belong) or else add them to the automation and rebuild them correctly.

For example, you can have scripts that use your infrastructure provider API to list all resources (all servers, storage, network configurations, etc.) and compare this with your configuration registry. This will catch errors in your configuration (e.g., things not being added correctly to the registry), as well as things being done outside the proper channels.