Recap of Terraform Basics

What is Infra-as-code

- Core idea is that you write & execute code to define, deploy and update your infrastructure.
- An important shift in mindset
- Four broad categories:
  - *Ad hoc scripts*
    - bash/ruby/Python
    - using apt-get or similar to install s/w
  - *Configuration management tools*
    - Chef, Puppet, Ansible and SaltStack
    - These tools provide idempotency. To achieve this in adhoc script will take a lot more code using Ifs/Else blocks
  - *Server templating tools*
    - Docker, Packer or Vagrant
    - Create an image and use any other provisioning tool to spawn machines/containers using the template.
  - *Server Provisioning tools*
    - Both the above defines what runs on the server
    - However, Server Provisioning tools are responsible for creating the servers itself.
    - Terraform, Cloudformation, OpenStack Heat

Why bother?

- Speed & Safety
  - Automation will be faster than manual
  - No manual errors
  - Consistent
- Documented because of code
- Code Reviews
  - All changes go through more than two eyes
  - Run automated tests
  - Static analysis tools
- Version Control
  - View entire history of your infrastructure
  - Easier to debug, since you would look at what changed recently
- As per DevOps Report Survey, organisations are deploying 200 times more frequently with lower number of rollbacks.

Configuration Mgmt vs Provisioning

- Examples of configuration management tools are Chef, Puppet and Ansible. Their primary objective of invention was to install and manage software on existing servers.
- Example of Provisioning tools are CloudFormation and Terraform. Their primary objective of invention is to provision the servers (as well as the rest of your infrastructure, like load balancers, databases, networking configuration, etc). And, then these provisioned servers can be configured using the configuration management tools.
- These two categories are not mutually exclusive, as most configuration management tools can do some degree of provisioning and most provisioning tools can do some degree of configuration management. But the focus on configuration management or provisioning means that some of the tools are going to be a better fit for certain types of tasks.

Mutable vs Immutable Infrastructure

- Configuration management tools such as Chef, Puppet, Ansible, and SaltStack typically default to a mutable infrastructure paradigm. For example, if you tell Chef to install a new version of OpenSSL, it'll run the software update on your existing servers and the changes will happen in-place.
- Over time, as you apply more and more updates, each server builds up a unique history of changes. This often leads to a phenomenon known as *configuration drift*, where each server becomes slightly different than all the others, leading to subtle configuration bugs that are difficult to diagnose and nearly impossible to reproduce.

Procedural Language vs Declarative Language

- Chef and Ansible encourage a procedural style where you write code that specifies, step-by-step, how to to achieve some desired end state.
- Terraform, CloudFormation, SaltStack, and Puppet all encourage a more declarative style where you write code that specifies your desired end state, and the IAC tool itself is responsible for figuring out how to achieve that state.
- For example, let's say you wanted to deploy 10 EC2 Instances to run v1 of an app

Ansible template that does this with a procedural approach:

```
- ec2:
    count: 10
    image: ami-v1
    instance_type: t2.micro
```

Terraform template that does the same thing using a declarative approach:

```
resource "aws_instance" "example" {
  count         = 10
  ami           = "ami-v1"
  instance_type = "t2.micro"
}
```

These two approaches may look similar, and when you initially execute them with Ansible or Terraform, they will produce similar results. The interesting thing is what happens when you want to make a change.

- For example, imagine traffic has gone up and you want to increase the number of servers to 15

With Ansible, the procedural code you wrote earlier is no longer useful; if you just updated the number of servers to 15 and reran that code, it would deploy 15 new servers, giving you 25 total! So instead, you have to be aware of what is already deployed and write a totally new procedural script to add the 5 new servers:

```
- ec2:
    count: 5
    image: ami-v1
    instance_type: t2.micro
```

With declarative code, since all you do is declare the end state you want, and Terraform figures out how to get to that end state, Terraform will also be aware of any state it created in the past. Therefore, to deploy 5 more servers, all you have to do is go back to the same Terraform template and update the count from 10 to 15:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-v1"
  instance_type = "t2.micro"
}
```

- Now what happens when you want to deploy v2 of the service?

With the procedural approach, both of your previous Ansible templates are again not useful, so you have to write yet another template to track down the 10 servers you deployed previous (or was it 15 now?) and carefully update each one to the new version.

With the declarative approach of Terraform, you go back to the exact same template once again and simply change the ami version number to v2:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-v2"
  instance_type = "t2.micro"
}
```