opsguru
https://www.opsguru.io/

DevGeekWeek 2019
INSPIRED BY INNOVATION

Ansible and Terraform-IAC
&
CM Made easy

Lev Epshtein

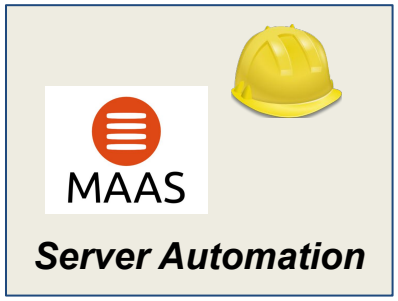
JOHN BRYCE
תלמד הייטק. זה עובד!
a matrix company

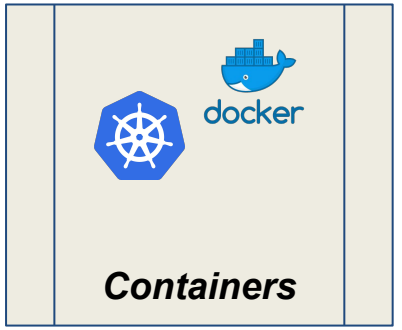
Content

- **Introduction to IaC - Infrastructure as Code**
  - Challenges and Principles
- **Ansible**
  - What is Ansible?
  - Architecture and process flow
  - Ansible and Inventory configuration
  - Ansible modules
  - Plays and playbooks
  - Roles
- **Terraform**
  - What is Terraform?
  - Terraform basics
  - Terraform with AWS
  - Terraform Advanced

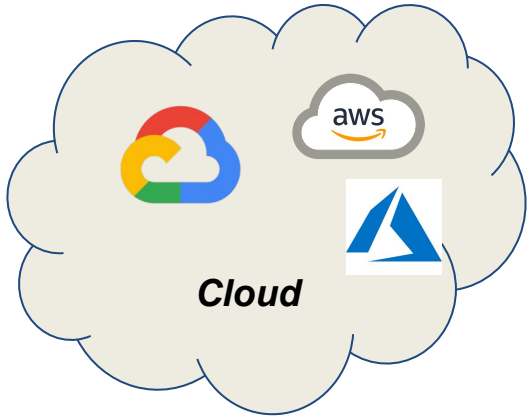
Introduction to IaC

Infrastructure as Code





Transformation of IT Infrastructure



What is Infrastructure as Code?

Infrastructure as code is an approach to using cloud era technologies to build and manage dynamic infrastructure. It treats the infrastructure, and the tools and services that manage the infrastructure itself, as a software system, adapting software engineering practices to manage changes to the system in a structured, safe way. This results in infrastructure with well tested functionality to manage routine operational tasks, and a team that has clear, reliable processes for making changes to the system.

- Virtualization, cloud, and containers have turned **infrastructure into software and data.**
- Develop, Test, and Deploy these systems using the same engineering practices and tooling that **have been been proven effective for application development.**
- Scale the number of systems without needing to scale team size to match.

IT systems are not just business critical, they are the business!

amazon NETFLIX Google facebook

Tools for IaC

Infrastructure provisioning

Terraform

Configuration management

ANSIBLE

puppet labs

SALTSTACK

CHEF

- Test Driven Development (TDD)
- Continuous Integration (CI)
- Continuous Delivery (CD)

Provide rigorous quality controls with a rapid pace of change.

CI/CD tools


circleci


CODESHIP


Spinnaker


XL DEPLOY


GitLab


TeamCity


Travis CI


Bamboo


Jenkins

Goals of infrastructure as code

- IT infrastructure should support and enable change, not be an obstacle or a constraint.
- IT staff should spend their time on valuable things which engage their abilities, not on routine, repetitive tasks.
- Users should be able to provision and manage the resources they need, without needing IT staff to do it for them.
- Teams should know how to recover quickly from failure, rather than depending on avoiding failure.
- Changes to the system should be routine, without drama or stress for users or IT staff.
- Improvements should be made continuously, rather than done through expensive and risky "big bang" projects.
- Solutions to problems should be proven through implementing, testing, and measuring them, rather than by discussing them in meetings and documents.

Principles of infrastructure as Code

- Reproducibility
- Consistency
- Repeatability
- Disposability
- Service continuity
- Self-testing systems
- Self-documenting systems
- Small changes
- Version all the things

Vagrant

vagrant --help
vagrant up --provider=virtualbox
vagrant status
vagrant halt
vagrant destroy
vagrant destroy <host name>
vagrant ssh <host name>

https://www.vagrantup.com/

Ansible

https://docs.ansible.com/ansible/latest/index.html

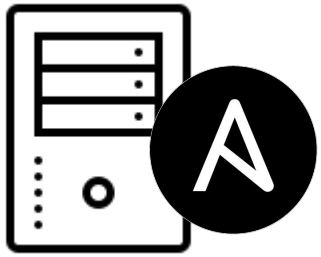
What's So Great About Ansible?

- Easy-to-Read Syntax
- Nothing to Install on the Remote Hosts
- Push Based
- Ansible Scales Down
- Built-in Modules

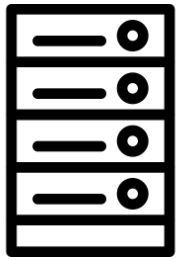
Idempotence is the ability to run an operation which produces the same result whether run once or multiple times.

In fact, almost every aspect of Ansible modules and commands is idempotent, and for those that aren't, Ansible allows you to define when the given command should be run, and what constitutes a changed or failed command, so you can easily maintain an idempotent configuration on all your servers.

ANSIBLE



- Configuration
- Server Role
- User

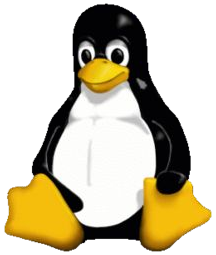


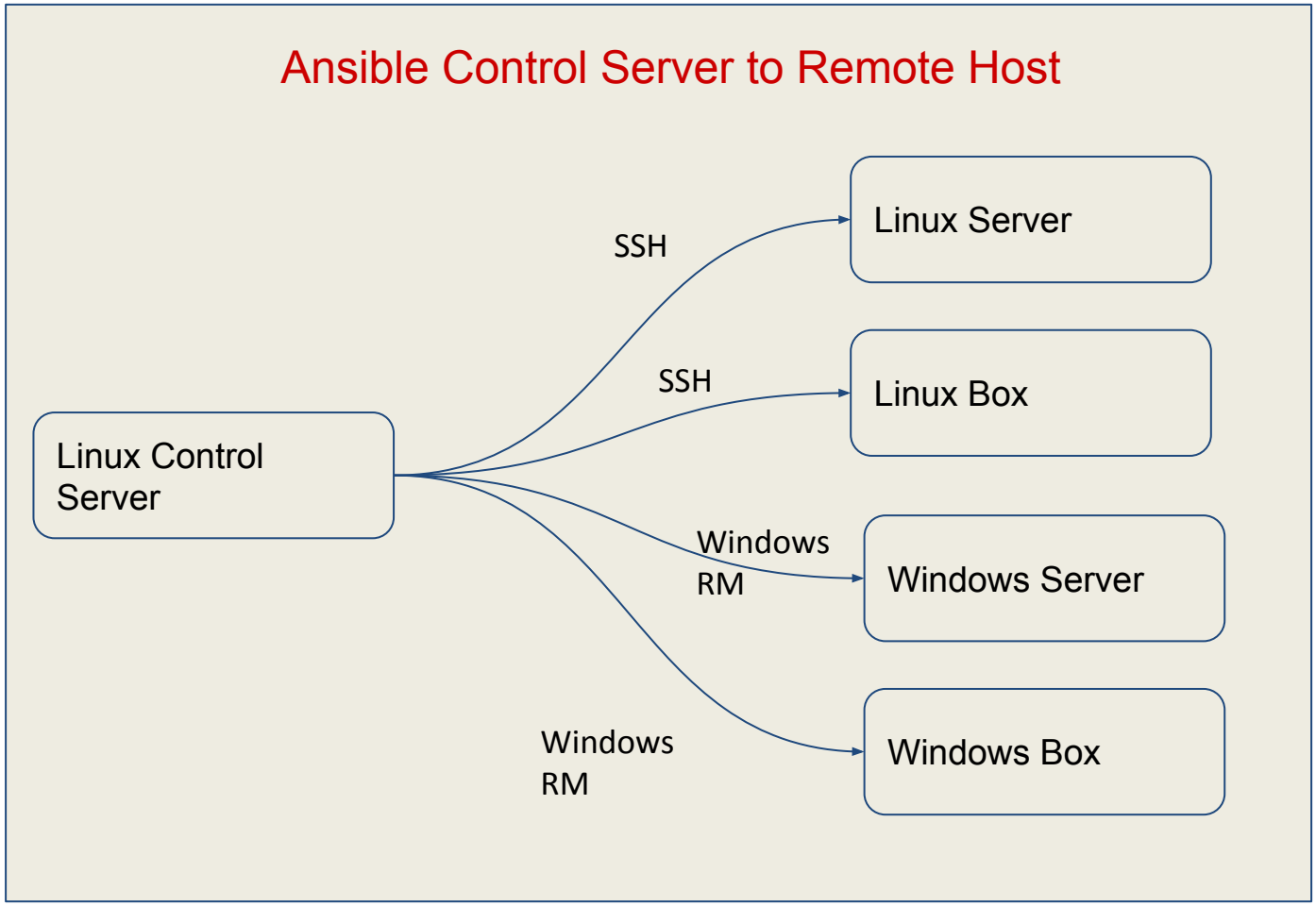
… a fictional machine capable of instantaneous or superluminal communication…

Ansible Runs on *nix

python™

Control Computer Must be *nix
but can manage both





- Ansible has the ability to simultaneously control between 1 and 10,000 machines without a significant degradation in performance.

Inventory: Describing Your Servers

Can be INI or YAML

https://yaml.org/

https://en.wikipedia.org/wiki/INF_file

https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#how-variables-are-merged

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:

 three.example.com:
```

Inventory file

Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systems listed in Ansible's inventory, which defaults to being saved in the location /etc/ansible/hosts. You can specify a different inventory file using the -i <path> option on the command line.

- Ad-Hoc Command: An ad-hoc command is something that you might type in to do something really quick, but don't want to save for later.

- Playbooks: playbooks are the basis for a really simple configuration management and multi-machine deployment system, unlike any that already exist, and one that is very well suited to deploying complex applications.

Ad-Hoc Examples

Usage: *ansible <host-pattern> [options]*

run the command on all servers in a group, in this case, *atlanta*, in 10 parallel forks:

```
$ ansible atlanta -a "/sbin/reboot" -f 10
```

To transfer a file directly to many servers:

```
$ ansible atlanta -m copy -a "src=/etc/hosts dest=/tmp/hosts"
```

Ensure a package is installed, but don't update it:

```
$ ansible webservers -m yum -a "name=acme state=present"
```

Ensure a package is installed to a specific version:

```
$ ansible webservers -m yum -a "name=acme-1.5 state=present"
```

https://docs.ansible.com/ansible/latest/user_guide/intro_adhoc.html

Ansible Playbooks

A playbook consists of a list of plays, each play is a list that contains tasks. Playbooks are written in YAML, which, like Python, cares about whitespace.

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running
    service:
      name: httpd
      state: started
  handlers:
    - name: restart apache
      service:
        name: httpd
          state: restarted
```

https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

Task List

Here is what a basic task looks like. As with most modules, the service module takes `key=value` arguments:

```
tasks:
  - name: make sure apache is running
    service:
      name: httpd
      state: started
```

The **command** and **shell** modules are the only modules that just take a list of arguments and don't use the `key=value` form. This makes them work as simply as you would expect:

```
tasks:

  - name: enable selinux

    command: /sbin/setenforce 1
```

https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#about-playbooks

```
tasks:
  - name: <Some description>
    <API>: PARAM1=foo PARAM2=foo PARAM3=foo
```

Playbooks can contain multiple plays. You may have a playbook that targets first the web servers, and then the database servers. For example:

```
---
- hosts: webservers
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf

- hosts: databases
  remote_user: root

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
      state: latest
  - name: ensure that postgresql is started
    service:
      name: postgresql
        state: started
```

Executing a Playbooks

Let's run a playbook using a parallelism level of 10:

```
ansible-playbook playbook.yml -f 10
```

Ansible-Pull

Should you want to invert the architecture of Ansible, so that nodes check in to a central location, instead of pushing configuration out to them, you can.

The ansible-pull is a small script that will checkout a repo of configuration instructions from git, and then run ansible-playbook against that content.

Run ansible-pull --help for details.

Ansible modules

https://docs.ansible.com/ansible/latest/user_guide/modules_intro.html

Modules (also referred to as "task plugins" or "library plugins") are discrete units of code that can be used from the **command line** or in a **playbook task**.

```
ansible webservers -m service -a "name=httpd state=started"

ansible webservers -m ping

ansible webservers -m command -a "/sbin/reboot -t now"
```

Each module supports taking arguments. Nearly all modules take key=value arguments, space delimited. Some modules take no arguments, and the command/shell modules simply take the string of the command you want to run.

From playbooks, Ansible modules are executed in a very similar way:

```
- name: reboot the servers

   action: command /sbin/reboot -t now
```

Ansible modules

Ansible modules normally return a data structure that can be registered into a variable, or seen directly when output by the ansible program.

https://docs.ansible.com/ansible/latest/reference_appendices/common_return_values.html

Module Index:

- System
- Commands
- Files
- Database
- Cloud
- Windows
- More...

Variables and Facts

Defining Variables in Playbooks

```
vars:
    key_file: /etc/nginx/ssl/nginx.key
    cert_file: /etc/nginx/ssl/nginx.crt
    conf_file: /etc/nginx/sites-available/default
    server_name: localhost
```

Ansible also allows you to put variables into one or more files, using a section called *vars_files*.

```
vars_files:
    - nginx.yml
```

nginx.yml

```
key_file: /etc/nginx/ssl/nginx.key
cert_file: /etc/nginx/ssl/nginx.crt
conf_file: /etc/nginx/sites-available/default
server_name: localhost
```

Variables and Facts

Registering Variables

Set the value of a variable based on the result of a task.

Capture the output of the whoami command to a variable named login:

```
- name: capture output of whoami command
  command: whoami
  register: login
```

Once you've defined variables, you can use them in your playbooks using the Jinja2 templating system. Here's a simple Jinja2 template:

```
My amp goes to {{ max_amp_value }}
```

In playbook:

```
template: src=foo.cfg.j2 dest={{ remote_install_path }}/foo.cfg
```

Variables and Facts

Facts

Before the first task runs, this happens:

```
GATHERING FACTS ***************************************************
ok: [servername]
```

Queries host for all kinds of details about the: CPU architecture, operating system, IP addresses, memory info, disk info, and more. This information is stored in variables that are called **facts**, and they behave just like any other variable.

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variables-discovered-from-systems-facts

Conditionals and Loops

https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html

https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html

Ansible Roles

In Ansible, the **role** is the primary mechanism for breaking a playbook into multiple files. This simplifies writing complex playbooks, and it makes them easier to reuse.

Basic Role Structure

`roles/database/tasks/main.yml` - Tasks

`roles/database/files/` - Holds files to be uploaded to hosts

`roles/database/templates/` - Holds Jinja2 template files

`roles/database/handlers/main.yml` - Handlers

`roles/database/vars/main.yml` - Variables that shouldn't be overridden

`roles/database/defaults/main.yml` - Default variables that can be overridden

`roles/database/meta/main.yml` - Dependency information about a role

Web Server structure example

Demo3

ansible-galaxy init gw2019.mysql
ansible-galaxy init gw2019.nginx
ansible-galaxy init gw2019.php
ansible-galaxy init gw2019.wordpress

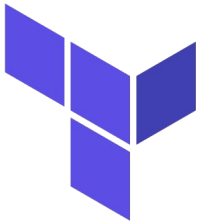
Using Roles

`roles:` option for a given play:

```
---
- hosts: webservers
  roles:
    - common
    - webservers
```

Terraform

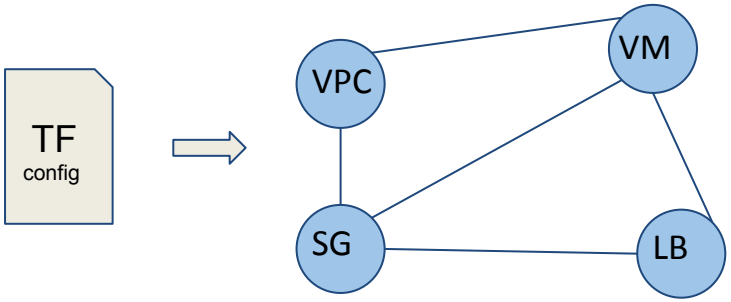
https://www.terraform.io/



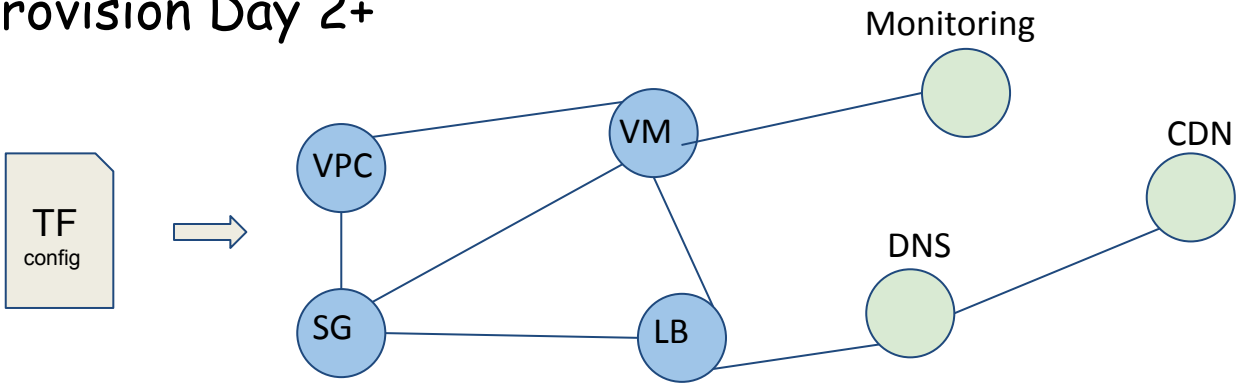
HashiCorp
Terraform

What is Terraform?

- Provision Day 1



- Provision Day 2+



Main command

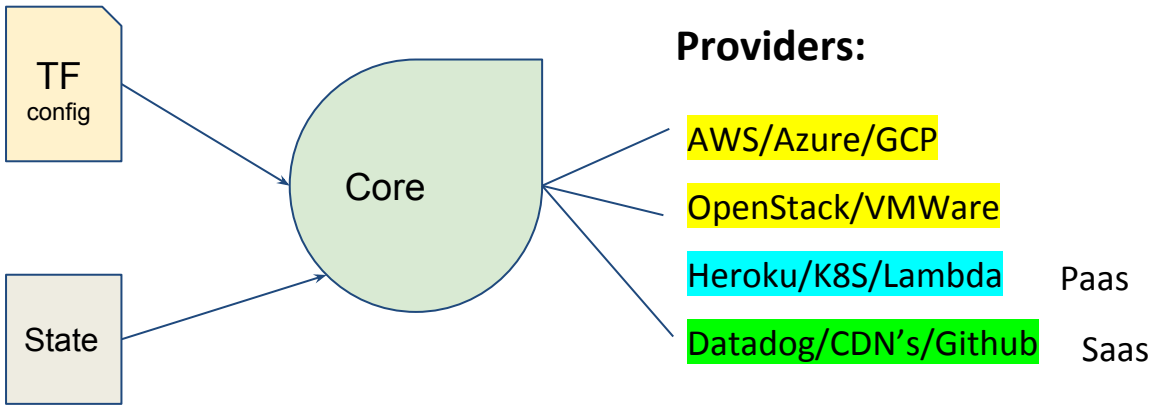
terraform plan (Real World vs Desired Config)

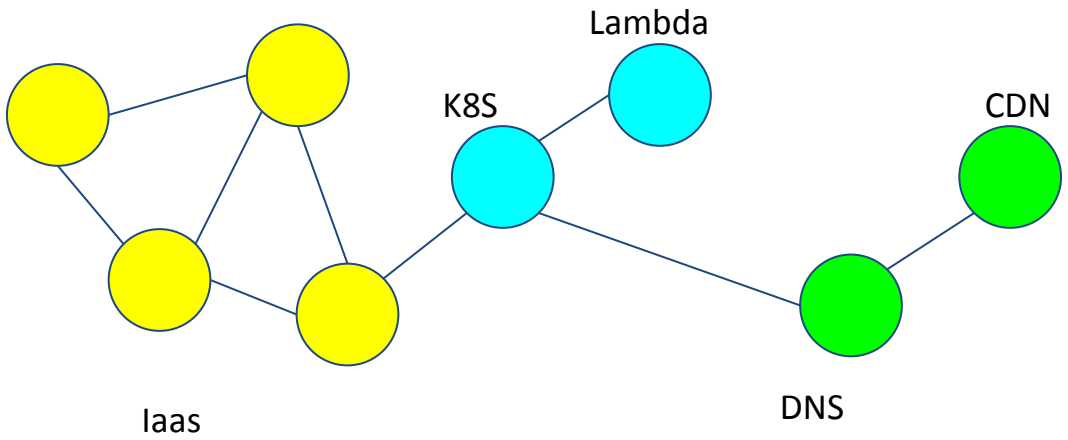
terraform apply (Plan vs Real World)

terraform destroy (Plan vs Real World)

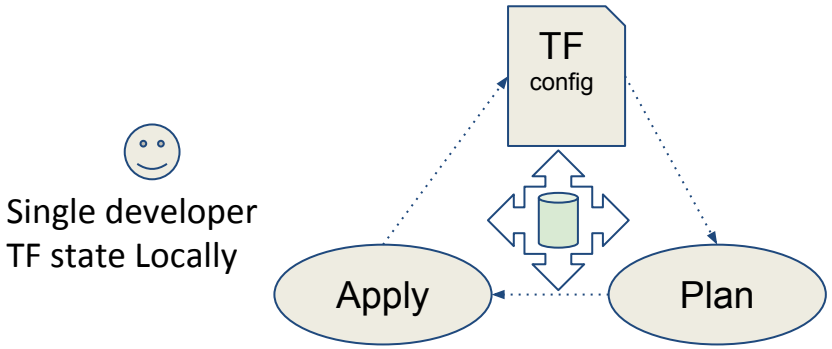
terraform help (for all)

How Terraform Work

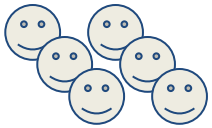




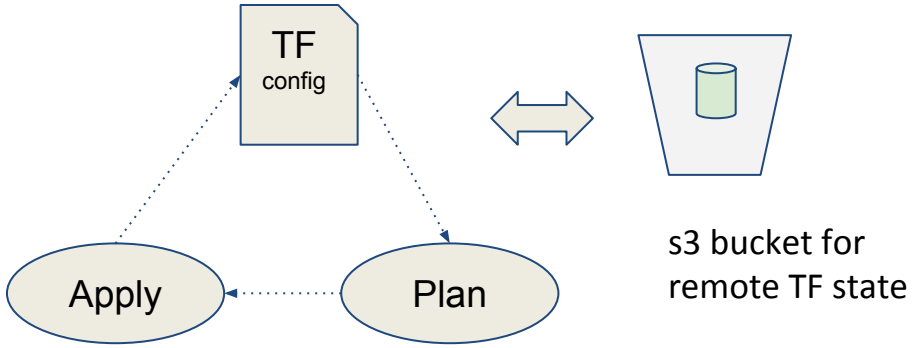
How we manage Terraform state



Single developer
TF state Locally



Multiple developers
Remote TF state



s3 bucket for
remote TF state

What is Terraform? - Summary

- Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
- Terraform can manage existing and popular service providers as well as custom in-house solutions.
- Configuration files describe to Terraform the components needed to run a single application or your entire datacenter.
- Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure.
- As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.
- The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

Deploy Single Server

Demo 1

Configuration Language

- Resources
- Providers
- Data Sources
- Input Variables
- Output Values
- Modules
- Configuration Syntax
- Expressions
- Functions

Resources

The main purpose of the Terraform language is declaring resources. All other language features exist only to make the definition of resources more flexible and convenient.

```
resource "aws_instance" "web" {
 ami           =
"ami-0f2176987ee50226e"
 instance_type = "t2.micro"

 tags = {
   Name = "GeekWeek2019"
 }
}
```

Providers

```
provider "aws" {
 region = "us-west-2"
}
```

https://www.terraform.io/docs/providers/index.html

Deploy Single Server

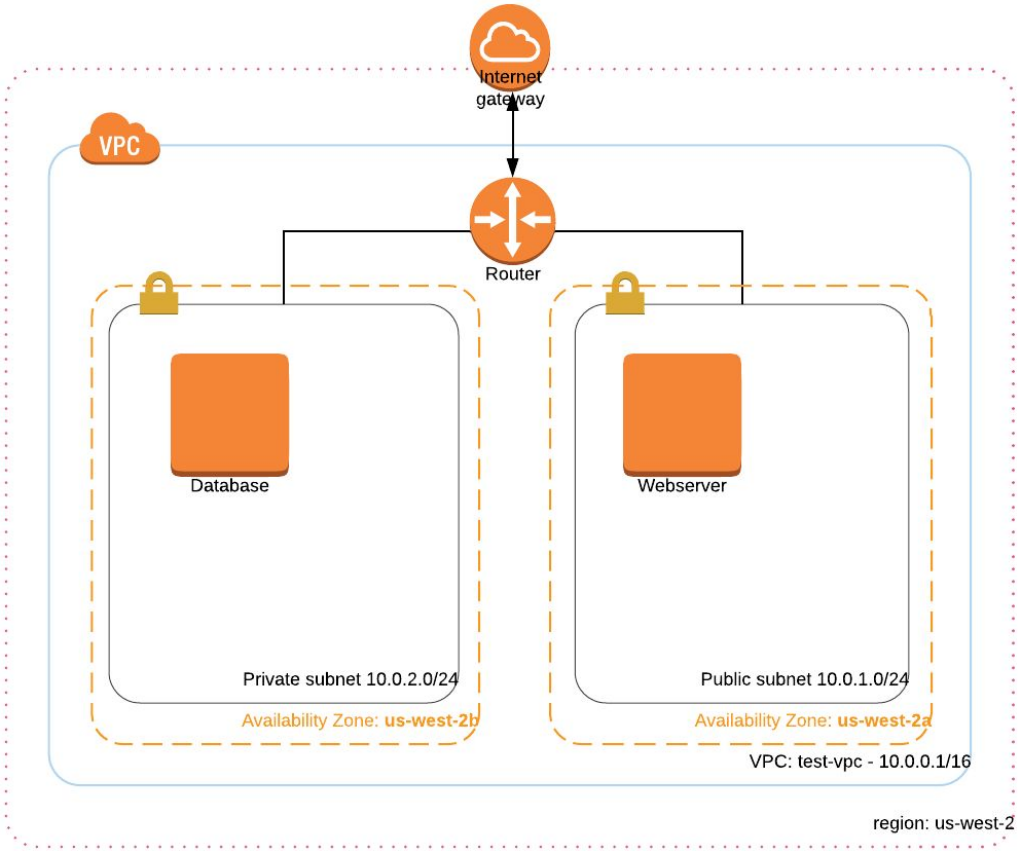
Demo 2

Data Sources

```
data "aws_ami" "ubuntu" {
 most_recent = true

 filter {
   name   = "name"
   values = ["ubuntu/images/hvm-ssd/ubuntu-trusty-14.04-amd64-server-*"]
 }

 filter {
   name   = "virtualization-type"
   values = ["hvm"]
 }

 owners = ["099720109477"] # Canonical
}
```

https://www.terraform.io/docs/configuration/data-sources.html

Build VPC Deploy Web+Db Servers



Demo 3

Variables Input

```
variable "aws_region" {
 description = "Region for the VPC"
 default = "us-west-2"
}

variable "vpc_cidr" {
 description = "CIDR for the VPC"
 default = "10.0.0.0/16"
}
```

https://www.terraform.io/docs/configuration/variables.html

Output Values

```
output "web_ip_addr" {
 value =
aws_instance.wb.public_ip
}
```

https://www.terraform.io/docs/configuration/outputs.html

Modules

```
# Module VPC
module "vpc" {
 source = "./vpc"
 vpc_cidr = "${var.vpc_cidr}"
 public_subnet_cidr = "${var.public_subnet_cidr}"
 private_subnet_cidr = "${var.private_subnet_cidr}"
}
```

https://www.terraform.io/docs/configuration/modules.html

Configuration syntax

https://www.terraform.io/docs/configuration/syntax.html

Expressions

https://www.terraform.io/docs/configuration/expressions.html

Built-in Function

https://www.terraform.io/docs/configuration/functions.html

DevGeekWeek 2019
INSPIRED BY INNOVATION

Thanks :)

See you next year

JOHN BRYCE
תלמד הייטק. זה עובד!
a matrix company