AWS Terraform Workshop

Build AWS Resources with Infrastructure as Code

Introductions

- Your Name
- Job Title
- Automation Experience
- Favorite Text Editor

The Slide Deck

Follow along on your own computer at this link:

https://git.io/JerH6

Table of Contents

Chapter 1

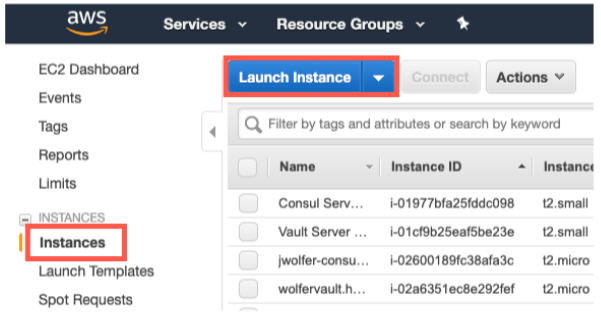
Introduction to Terraform

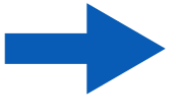
How to Provision an AWS Instance

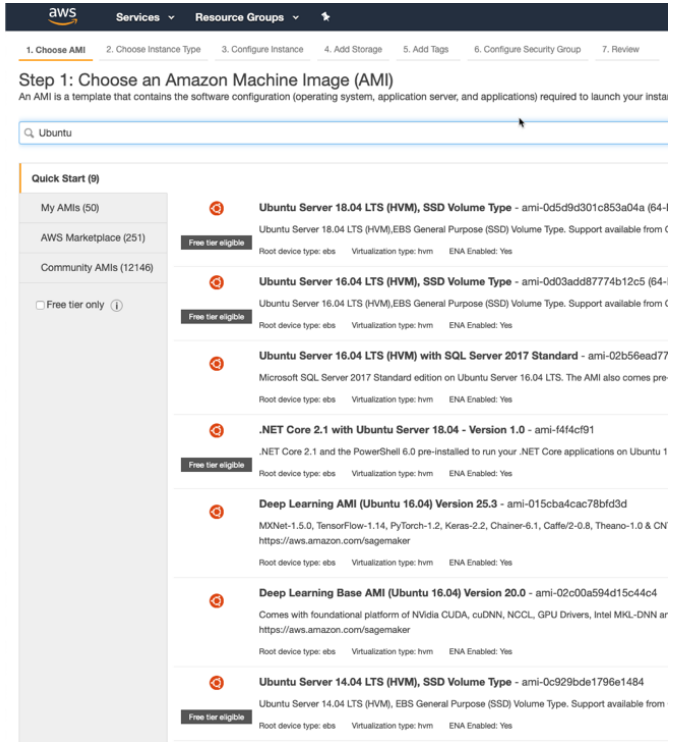
Let's look at a few different ways you could provision a new AWS Instance. Before we start we'll need to gather some basic information including (but not limited to):

- Instance Name
- Operating System (Image)
- VM Size
- Geographical Location (Region)
- Security Groups

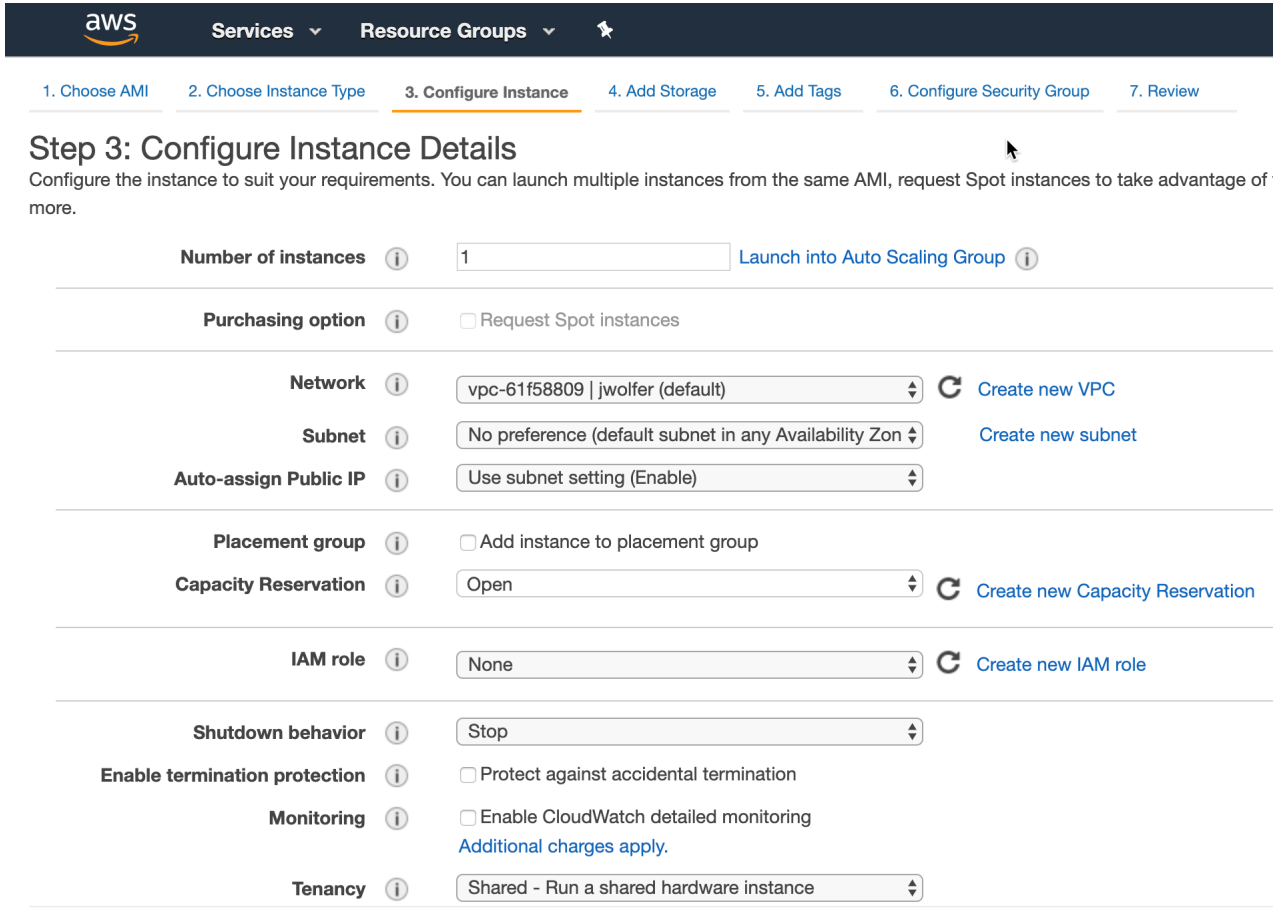
Method 1: AWS Console (GUI)







Method 1: AWS Portal (GUI)



Method 2: CloudFormation Templates

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",

  "Description" : "AWS CloudFormation Sample Template EC2InstanceWithSecurityGroupSample: Crea

  "Parameters" : {
    "KeyName": {
      "Description" : "Name of an existing EC2 KeyPair to enable SSH access to the instance",
      "Type": "AWS::EC2::KeyPair::KeyName",
      "ConstraintDescription" : "must be the name of an existing EC2 KeyPair."
    },
  }
}
```

CloudFormation Templates provide a consistent and reliable way to provision AWS resources. JSON is easy for computers to read, but can be challenging for humans to edit and troubleshoot.

Method 3: Provision with Terraform

```
resource aws_instance "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    Name = "HelloWorld"
  }
}
```

Example Terraform code for building an AWS instance.

What is Terraform?

```
resource aws_instance "catapp" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type
  tags = {
    Name = "${var.prefix}-meow"
  }
```

- Executable Documentation
- Human and machine readable
- Easy to learn
- Test, share, re-use, automate
- Works on all major cloud providers

What is Infrastructure as Code?

Infrastructure as Code (IaC) is the process of managing and provisioning cloud infrastructure with machine-readable definition files.

Think of it as executable documentation.

Infrastructure as Code Allows Us To...

Infrastructure as Code Allows Us To...

- Provide a codified workflow to create infrastructure

Infrastructure as Code Allows Us To...

- Provide a codified workflow to create infrastructure
- Change and update existing infrastructure

Infrastructure as Code Allows Us To...

- Provide a codified workflow to create infrastructure
- Change and update existing infrastructure
- Safely test changes using **`terraform plan`** in dry run mode

Infrastructure as Code Allows Us To...

- Provide a codified workflow to create infrastructure
- Change and update existing infrastructure
- Safely test changes using **`terraform plan`** in dry run mode
- Integrate with application code workflows (Git, CI/CD tools)

Infrastructure as Code Allows Us To...

- Provide a codified workflow to create infrastructure
- Change and update existing infrastructure
- Safely test changes using **`terraform plan`** in dry run mode
- Integrate with application code workflows (Git, CI/CD tools)
- Provide reusable modules for easy sharing and collaboration

Infrastructure as Code Allows Us To...

- Provide a codified workflow to create infrastructure
- Change and update existing infrastructure
- Safely test changes using **`terraform plan`** in dry run mode
- Integrate with application code workflows (Git, CI/CD tools)
- Provide reusable modules for easy sharing and collaboration
- Enforce security policy and organizational standards

Infrastructure as Code Allows Us To...

- Provide a codified workflow to create infrastructure
- Change and update existing infrastructure
- Safely test changes using **`terraform plan`** in dry run mode
- Integrate with application code workflows (Git, CI/CD tools)
- Provide reusable modules for easy sharing and collaboration
- Enforce security policy and organizational standards
- Enable collaboration between different teams

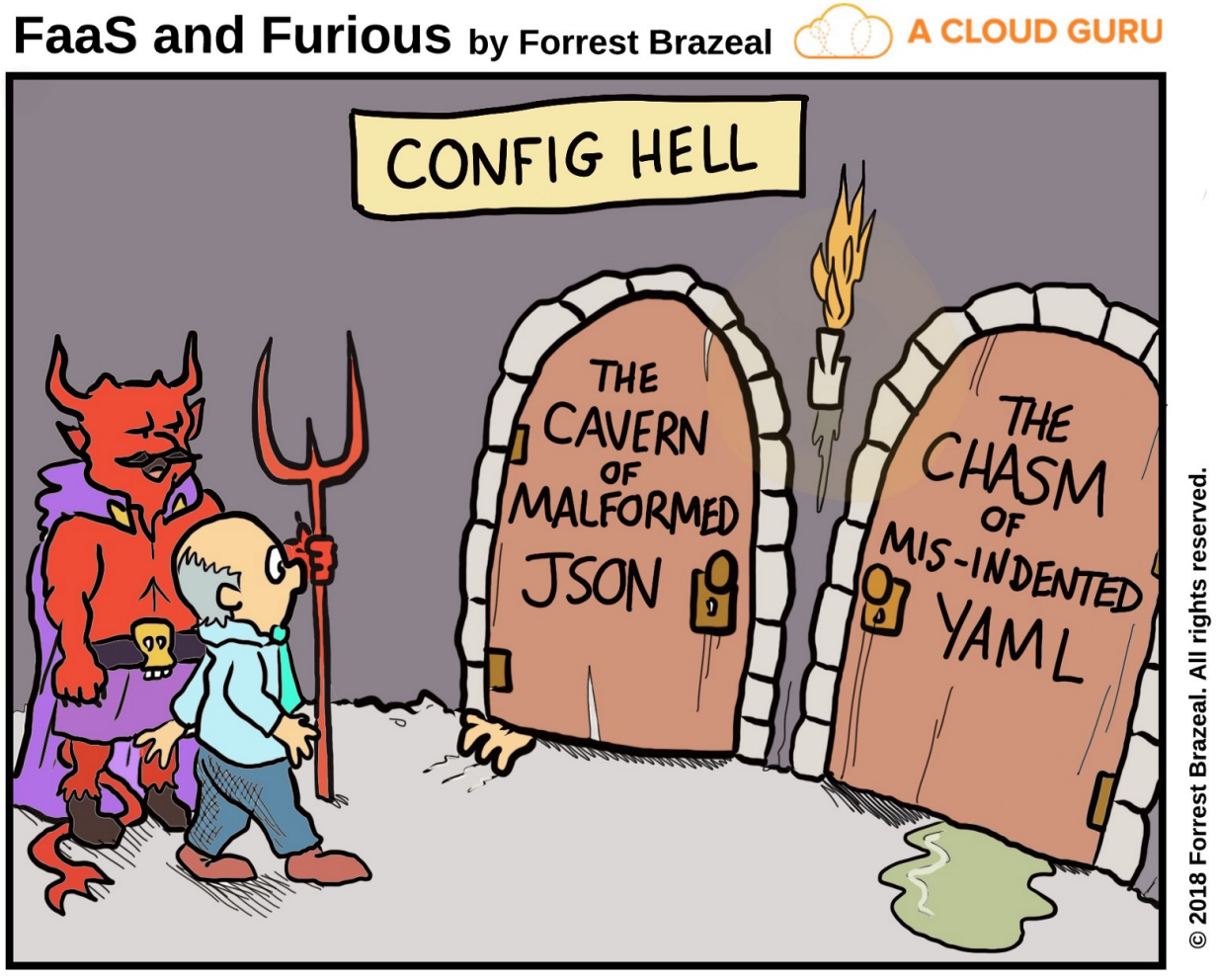
Other Infrastructure as Code Tools

These tools work well for configuring the operating system and application. They are not purpose-built for provisioning cloud infrastructure and platform services.

Native Cloud Provisioning Tools

Each cloud has its own YAML or JSON based provisioning tool.

Terraform can be used across *all* major cloud providers and VM hypervisors.



**"Come on, make up your mind -
or it's back to the Sinkhole of Nested XML."**

Terraform vs. JSON

CFT JSON:

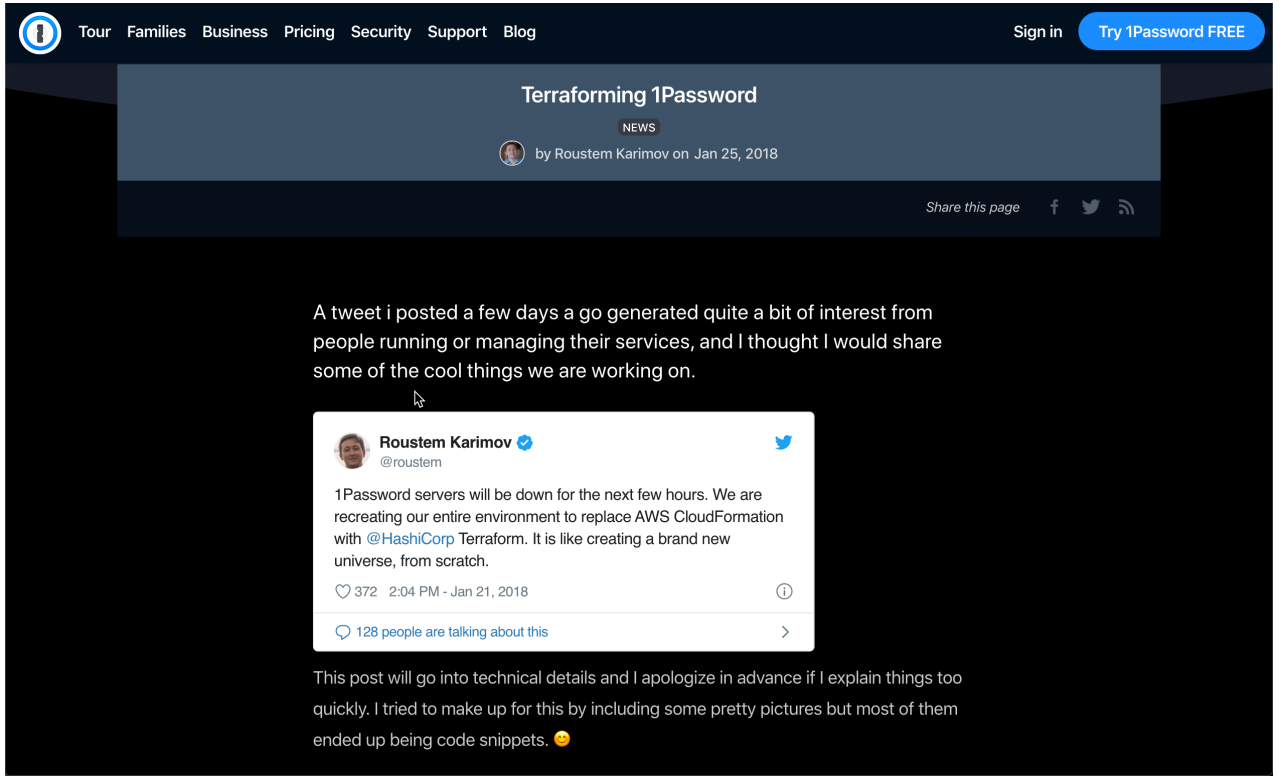
```
"name": "{ "Fn::Join" : [ "-", [ PilotServerName, vm ] ] }",
```

Terraform:

```
name = "${var.PilotServerName}-vm"
```

Terraform code (HCL) is easy to learn and easy to read. It is also 50-70% more compact than an equivalent JSON configuration.

Why Terraform?



https://blog.1password.com/terraforming-1password/

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers
- Increase provisioning speed

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers
- Increase provisioning speed
- Improve efficiency

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers
- Increase provisioning speed
- Improve efficiency
- Reduce risk

Live Demo

Chapter 2

Terraform Basics

What is Terraform?

Terraform is an open source provisioning tool.

It ships as a single binary which is written in Go. Terraform is cross platform and can run on Linux, Windows, or MacOS.

Installing terraform is easy. You simply download a zip file, unzip it, and run it.

Terraform Command Line

Terraform is a command line tool.

Terraform commands are either typed in manually or run automatically from a script.

The commands are the same whether you are on Linux or Windows or MacOS.

Terraform has subcommands that perform different actions.

```
# Basic Terraform Commands
terraform version
terraform help
terraform init
terraform plan
terraform apply
terraform destroy
```

Terraform Help

```
$ terraform help
Usage: terraform [-version] [-help] <command> [args]
...
Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    graph              Create a visual graph of Terraform resources
```

Type `terraform subcommand help` to view help on a particular subcommand.

Terraform Code

```
resource aws_vpc "main" {
  cidr_block       = "10.0.0.0/16"
  instance_tenancy = "dedicated"
}
```

Terraform code is based on the HCL2 toolkit. HCL stands for HashiCorp Configuration Language.

Terraform code, or simply *terraform* is a declarative language that is specifically designed for provisioning infrastructure on any cloud or platform.

Terraform Comments

Line Comments begin with an octothorpe*, or pound symbol: #

```
# This is a line comment.
```

Block comments are contained between /* and */ symbols.

```
/* This is a block comment.
Block comments can span multiple lines.
The comment ends with this symbol: */
```

* Yes, it really is called an octothorpe.

Terraform Workspaces

A terraform workspace is simply a folder or directory that contains terraform code.

Terraform files always end in either a `*.tf` or `*.tfvars` extension.

Most terraform workspaces contain a minimum of three files:

main.tf - Most of your functional code will go here.
variables.tf - This file is for storing variables.
outputs.tf - Define what is shown at the end of a terraform run.

Terraform Init

```
$ terraform init

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.35.0...
...
 provider.aws: version = "~> 2.35"

Terraform has been successfully initialized!
```

Terraform fetches any required providers and modules and stores them in the .terraform directory. If you add, change or update your modules or providers you will need to run init again.

Terraform Plan

```
$ terraform plan
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # aws_vpc.main will be created
  + resource "aws_vpc" "main" {
      + arn                              = (known after apply)
      + cidr_block                       = "10.0.0.0/16"
      ...
      + instance_tenancy                 = "dedicated"
    }
```

Preview your changes with `terraform plan` before you apply them.

Where are Variables Defined?

Terraform variables are placed in a file called *variables.tf*. Variables can have default settings. If you omit the default, the user will be prompted to enter a value. Here we are *declaring* the variables that we intend to use.

```
variable "prefix" {
  description = "This prefix will be included in the name of most resour
}

variable "instance_tenancy" {
  description = "A tenancy option for instances launched into the VPC."
  default     = "dedicated"
}
```

How are Variables Set?

Once you have some variables defined, you can set and override them in different ways. Here is the level of precedence for each method.

This list goes from highest precedence (1) to lowest (5).

1. Command line flag - run as a command line switch
2. Configuration file - set in your terraform.tfvars file
3. Environment variable - part of your shell environment
4. Default Config - default value in variables.tf
5. User manual entry - if not specified, prompt the user for entry

Getting Started with Instruqt

Instruqt is the HashiCorp training platform. Visit the link below for a short tutorial, or if you're already familiar with Instruqt you can skip to the next slide.

https://instruqt.com/instruqt/tracks/getting-started-with-instruqt

Lab Exercise: Terraform Basics

In this lab you'll learn how to set up your editor, use the Terraform command line tool, integrate with AWS, and do a few dry runs with different settings.

Your instructor will provide the URL for the lab environment.

STOP after you complete the second quiz.

Chapter 2 Review

In this chapter we:

- Used the **`terraform init`** command
- Ran the **`terraform plan`** command
- Learned about variables
- Set our tenancy and prefix

Chapter 3

Terraform in Action

Anatomy of a Resource

Every terraform resource is structured exactly the same way.

```
resource type "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
}
```

resource = Top level keyword
type = Type of resource. Example: `aws_instance`.
name = Arbitrary name to refer to this resource. Used internally by terraform. This field *cannot* be a variable.

Terraform Provider Configuration

The terraform core program requires at least one provider to build anything.

You can manually configure which version(s) of a provider you would like to use. If you leave this option out, Terraform will default to the latest available version of the provider.

```
provider "aws" {
  version = "=2.35.0"
}
```

Versioning Operators

```
- = (or no operator): exact version equality
- !=: version not equal
- \>, >=, <, <=: version comparison
- ~>: pessimistic constraint, constraining both the oldest and newest
version allowed. ~> 0.9 is equivalent to >= 0.9, < 1.0, and ~> 0.8.4
is equivalent to >= 0.8.4, < 0.9
```

Re-usable modules should constrain only the minimum allowed version, such as >= 2.35.0.

Terraform Apply

```
$ terraform apply
An execution plan has been generated and is shown below.

Terraform will perform the following actions:
  # aws_vpc.main will be created
  + resource "aws_vpc" "main" {
      + cidr_block                     = "10.0.0.0/16"
      + instance_tenancy               = "dedicated"
        ...
      + tags                           = {
          + "Name" = "main"
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

`terraform apply` runs a plan and then if you approve, it applies the changes.

Terraform Destroy

```
$ terraform destroy
An execution plan has been generated and is shown below.

Terraform will perform the following actions:
  # aws_vpc.main will be destroyed
  - resource "aws_vpc" "main" {
      - cidr_block                       = "10.0.0.0/16" -> null
      - instance_tenancy                 = "dedicated" -> null
        ...
      - tags                             = {
          - "Name" = "main"
        } -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.
```

`terraform destroy` does the opposite. If you approve, your infrastructure is destroyed.

Terraform Format

Terraform comes with a built in code formatter/cleaner. It can make all your margins and list indentation neat and tidy. Beauty works better.

```
terraform fmt
```

Simply run it in a directory containing *.tf files and it will tidy up your code for you.

Terraform 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
}
```

Data sources are a way of querying a provider to return an existing resource, so that we can access its parameters for our own use.

Terraform Dependency Mapping

Terraform can automatically keep track of dependencies for you. Look at the two resources below. Note the highlighted lines in the aws_instance resource. This is how we tell one resource to refer to another in terraform.

```
resource aws_key_pair "my-keypair" {
  key_name   = "my-keypair"
  public_key = file(var.public_key)
}

resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  key_name      = aws_key_pair.my-keypair.name
```

Organize Your Terraform Code

Terraform will read any file in your workspace that ends in a `.tf` extension, but the convention is to have a main.tf, variables.tf, and outputs.tf. You may add more tf files if you wish.

```
main.tf
variables.tf
outputs.tf
```

Let's take a closer look at each of these files.

The Main File

The first file is called main.tf. This is where you normally store your terraform code. With larger, more complex infrastructure you might break this up across several files.

```
# This is the main.tf file.
resource aws_vpc "main" {
  cidr_block       = var.cidr_block
  instance_tenancy = var.instance_tenancy
}

resource aws_subnet "main" {
  vpc_id     = aws_vpc.main.id
  cidr_block = var.cidr_block
  }
}
...
```

The Variables File

The second file is called variables.tf. This is where you define your variables and optionally set some defaults.

```
variable "cidr_block" {
  description = "The address space that is used within the VPC. Changing this forces a new res
}

variable "instance_tenancy" {
  description = "A tenancy option for instances launched into the VPC. Acceptable values are '
  default     = "dedicated"
}
```

The Outputs File

The outputs file is where you configure any messages or data you want to show at the end of a terraform apply.

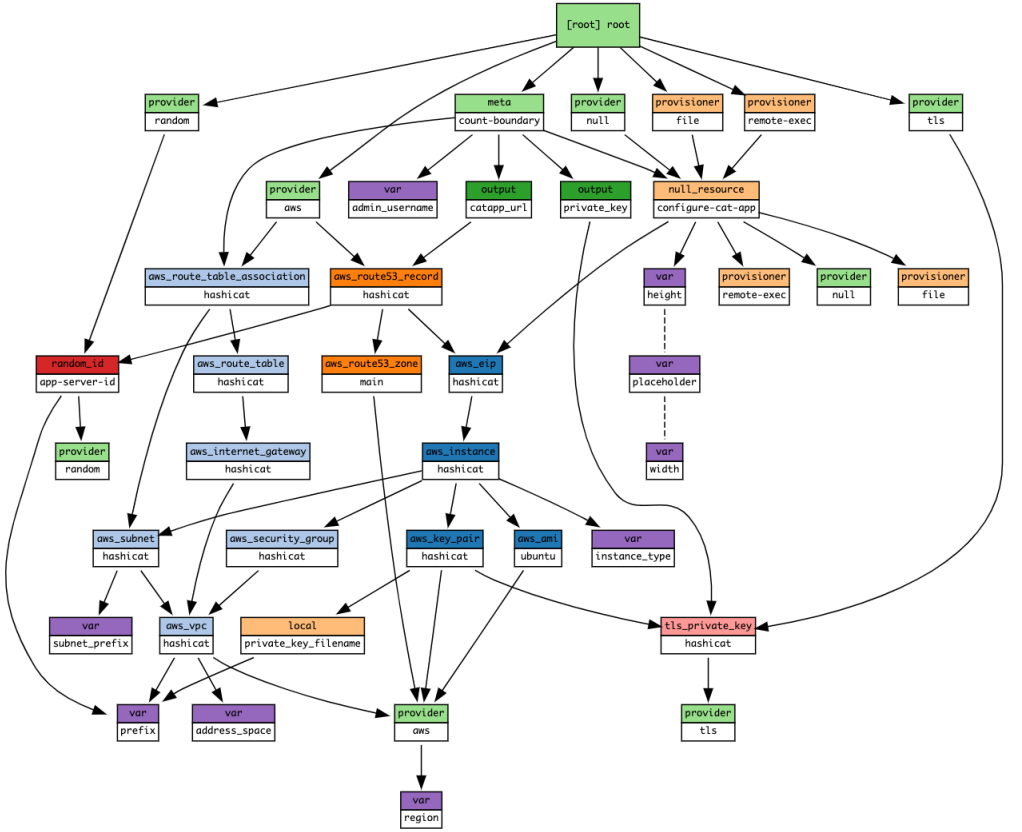
```
output "catapp_url" {
  value = "http://${aws_route53_record.hashicat.fqdn}"
}


output "private_key" {
  value = "${tls_private_key.hashicat.private_key_pem}"
}
```

Terraform Dependency Graph

The terraform resource graph visually depicts dependencies between resources.

The region and prefix variables are required to create the resource group, which is in turn required to build the virtual network.



Lab Exercise: Terraform in Action

Let's use Terraform to build, manage, and destroy AWS resources. In this lab exercise you'll build the HashiCat application stack by running the `terraform apply` command.

🛑 **STOP** after you complete the third quiz.

Chapter 3 Review

In this chapter we:

- Learned about Terraform resources
- Ran terraform plan, graph, apply and destroy
- Learned about dependencies
- Viewed a graph of the lab
- Looked at main.tf, variables.tf and outputs.tf
- Built the Meow World application

Chapter 4

Provision and Configure AWS Instances

Using Terraform Provisioners

Once you've used Terraform to stand up a virtual machine or container, you may wish to configure your operating system and applications. This is where provisioners come in. Terraform supports several different types of provisioners including: Bash, Powershell, Chef, Puppet, Ansible, and more.

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

The File Provisioner

The Terraform file provisioner copies files onto the remote machine.

```
provisioner "file" {
  source      = "files/"
  destination = "/home/${var.admin_username}/"

  connection {
    type        = "ssh"
    user        = var.username
    private_key = file(var.ssh_key)
    host        = ${self.ip}
  }
}
```

Note the *connection* block of code inside the provisioner block. The file provisioner supports both SSH and WinRM connections.

The Remote Exec Provisioner

The remote exec provisioner allows you to execute scripts or other programs on the target host. If it's something you can run unattended (for example, a software installer), then you can run it with remote exec.

```
provisioner "remote-exec" {
  inline = [
    "sudo chown -R ${var.admin_username}:${var.admin_username} /var/www/html",
    "chmod +x *.sh",
    "PLACEHOLDER=${var.placeholder} WIDTH=${var.width} HEIGHT=${var.height} PREFIX=${var.pref
  ]
...
}
```

In this example we're running a few commands to change some permissions and ownership, and to run a script with some enviroment variables.

Terraform & Config Management Tools

Terraform works well with common config management tools like Chef, Puppet or Ansible.

Run Puppet with 'local-exec':
https://www.terraform.io/docs/provisioners/local-exec.html

Terraform and Ansible - Better Together:
https://github.com/scarolan/ansible-terraform

Terraform Provisioner Tips

Terraform provisioners like remote-exec are great when you need to run a few simple commands or scripts. For more complex configuration management you'll want a tool like Chef or Ansible.

Provisioners only run the first time a Terraform run is executed. In this sense, they are not idempotent. If you need ongoing state management of VMs or servers that are long-lived, we recommend using a config management tool.

On the other hand, if you want immutable infrastructure you should consider using our Packer tool.

 Lab Exercise: Provisioners, Variables and Outputs

In part two of the lab we'll use a provisioner to install a new software package. We will also explore variables and outputs.

Return to the training lab and continue where you left off.

🛑 **STOP** after you complete the fourth quiz.

Chapter 4 Review

In this chapter we:

- Learned about Terraform Provisioners
- Explored the **file** and **remote-exec** provisioners
- Rebuilt our web server with a new provisioning step

Chapter 5

Terraform State

Terraform State

Terraform is a *stateful* application. This means that it keeps track of everything you build inside of a **state file**. You may have noticed the terraform.tfstate and terraform.tfstate.backup files that appeared inside your working directory. The state file is Terraform's source of record for everything it knows about.


```
{
  "terraform_version": "0.12.7",
  "serial": 14,
  "lineage": "452b4191-89f6-db17-a3b1-4470dcb00607",
  "outputs": {
    "catapp_url": {
      "value": "http://go-hashicat-5c0265179ccda553.workshop.aws.hashidemos.io",
      "type": "string"
    },
  }
}
```

Terraform Refresh

Sometimes infrastructure may be changed outside of Terraform's control.

The state file represents the *last known* state of the infrastructure. If you'd like to check and see if the state file still matches what you built, you can use the **terraform refresh** command.

Note that this does *not* update your infrastructure, it simply updates the state file.

```
terraform refresh
```

Changing Existing Infrastructure

Whenever you run a plan or apply, Terraform reconciles three different data sources:

1. What you wrote in your code
2. The state file
3. What actually exists

Terraform does its best to add, delete, change, or replace existing resources based on what is in your *.tf files. Here are the four different things that can happen to each resource during a plan/apply:

```
+   create
-   destroy
-/+ replace
~   update in-place
```

Terraform State Quiz

Configuration	State	Reality	Operation
aws_instance			???
aws_instance	aws_instance		???
aws_instance	aws_instance	aws_instance	???
	aws_instance	aws_instance	???
		aws_instance	???
	aws_instance		???

What happens in each scenario? Discuss.

Terraform State Quiz

Configuration	State	Reality	Operation
aws_instance			create
aws_instance	aws_instance		create
aws_instance	aws_instance	aws_instance	no-op
	aws_instance	aws_instance	delete
		aws_instance	no-op
	aws_instance		update state

What happens in each scenario? Discuss.

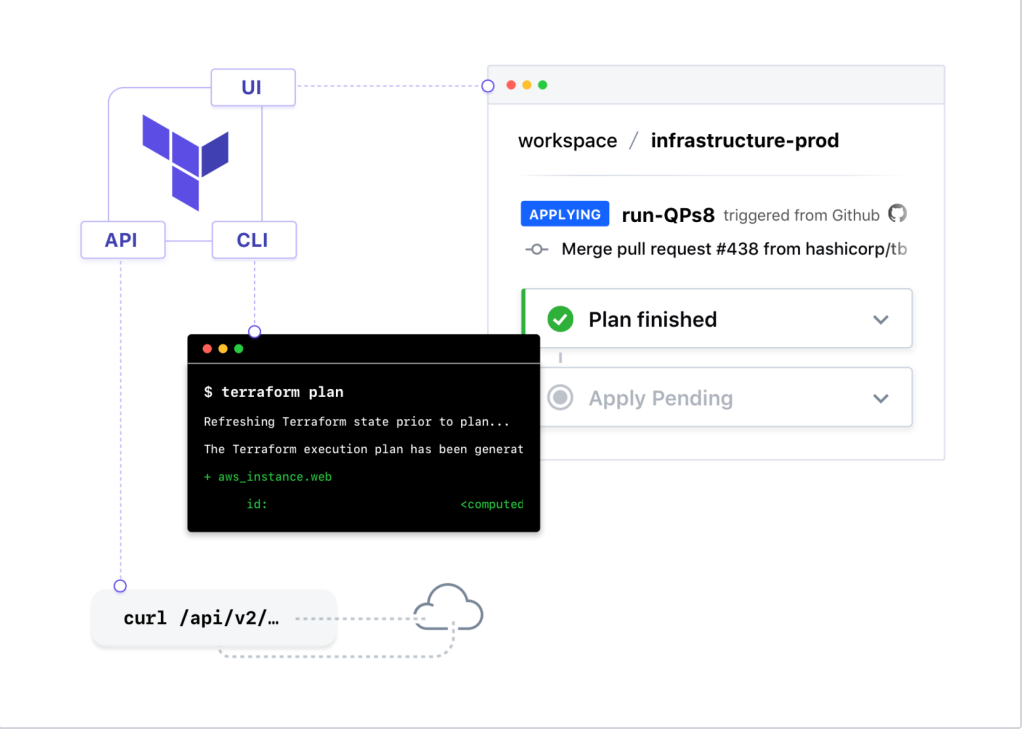
Chapter 6

Terraform Cloud

Terraform Cloud

Terraform Cloud is a free to use SaaS application that provides the best workflow for writing and building infrastructure as code with Terraform.

- State storage and management
- Web UI for viewing and approving Terraform runs
- Private module registry
- Version Control System (VCS) integration
- CLI, API or GUI driven actions
- Notifications for run events
- Full HTTP API for automation



Terraform Cloud or Terraform Enterprise?

Terraform Cloud is a hosted application that provides features like remote state management, API driven runs, policy management and more. Many users prefer a cloud-based SaaS solution because they don't want to maintain the infrastructure to run it.

Terraform Cloud for Business utilizes the same hosted environment as Terraform Cloud, but you get the features more applicable to larger teams. Single Sign-on, Audit Logging, and the ability to Terraform on-prem resources from the cloud.

Terraform Enterprise is the same application, but it runs in your own cloud environment or data center. Some users require more control over the Terraform Cloud application, or wish to run it in restricted networks behind corporate firewalls.

The feature list for these offerings is nearly identical. We will be using Terraform Cloud accounts for our lab exercises today.

Terraform Remote State

By default Terraform stores its state file in the workspace directory on your laptop or workstation. This is ok for development and experimentation, but in a production environment you need to protect and store the state file safely.

Terraform has an option to store and secure your state files remotely. Terraform Cloud accounts now offer unlimited state file storage even for open source users.

All state files are encrypted (using HashiCorp Vault) and stored securely in your Terraform Cloud account. You'll never have to worry about losing or deleting your state file again.

Terraform Cloud Execution Modes

Local Execution - Terraform commands run on your laptop or workstation and all variables are configured locally. Only the terraform state is stored remotely.

Remote Execution - Terraform commands are run in a Terraform Cloud container environment. All variables are stored in the remote workspace. Code can be stored in a Version Control System repository. Limited to 1 concurrent run for free tier users.

Lab Exercise: Terraform Cloud

In the final part of the second lab we'll create a free Terraform Cloud account and enable remote storage of our state file.

Return to the training lab and continue where you left off.

Congratulations, you completed the workshop!

Additional Resources

If you'd like to learn more about Terraform on AWS try the links below:

HashiCorp Learning Portal
https://learn.hashicorp.com/terraform/

Terraform - Beyond the Basics with AWS
https://aws.amazon.com/blogs/apn/terraform-beyond-the-basics-with-aws/

Terraform AWS Provider Documentation
https://www.terraform.io/docs/providers/aws/index.html

Link to this Slide Deck
https://git.io/JerH6

Workshop Feedback Survey

Your feedback is important to us!

The survey is short, we promise:

https://bit.ly/hashiworkshopfeedback