# Infra as Code

## Why do it?

- Getting started with Infrastructure as Code (**IAC**)
- Getting to know Terraform
- Build a sample Web Application using Terraform

## What We will be covering

- Session 2 - Infrastructure as Code
    - Concepts and benefits
    - Terraform
    - Ansible
    - CloudFormation
    - Code Modularity and Re-Use
    - States and Secret Management
    - Provisioning Infrastructure using IAC

## What is the agenda?

- Why do we need yet another framework (IAC)?
- What  is Terraform and its benefits
- Terraform Basics
  - Installation
  - Commands, Workflows, Resource creation, File structure
- Some (not all) Best Practices
- Additional Resources for Learning and Practice

## IAC – Why today?

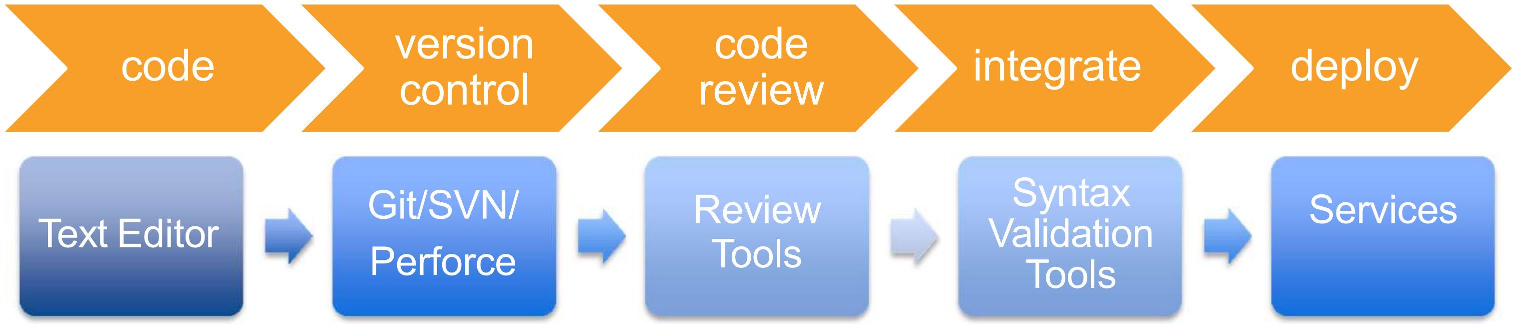
• Moving to cloud based infrastructure opens doors to building and managing infrastructure in completely new ways:

- Infrastructure can be provisioned in seconds
- Scale can be achieved without complicated capacity planning
- Being API driven means we can interact with our infrastructure via languages typically used in our applications

## Infrastructure as Code

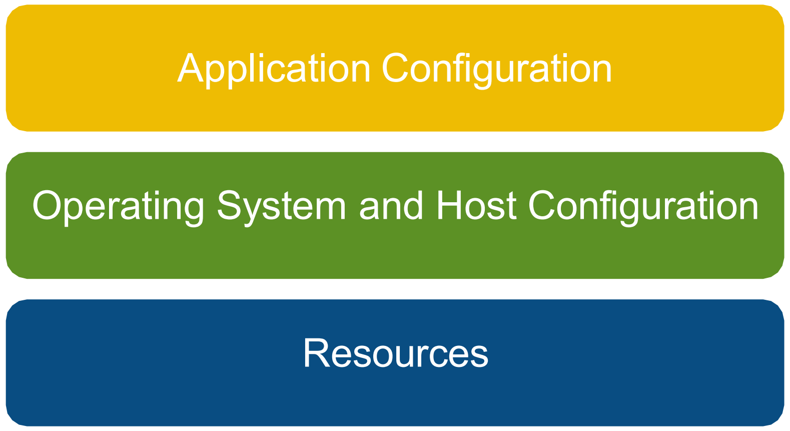
**Infrastructure as Code** is a practice by where traditional infrastructure management techniques are supplemented and often replaced by using code based tools and software development techniques.

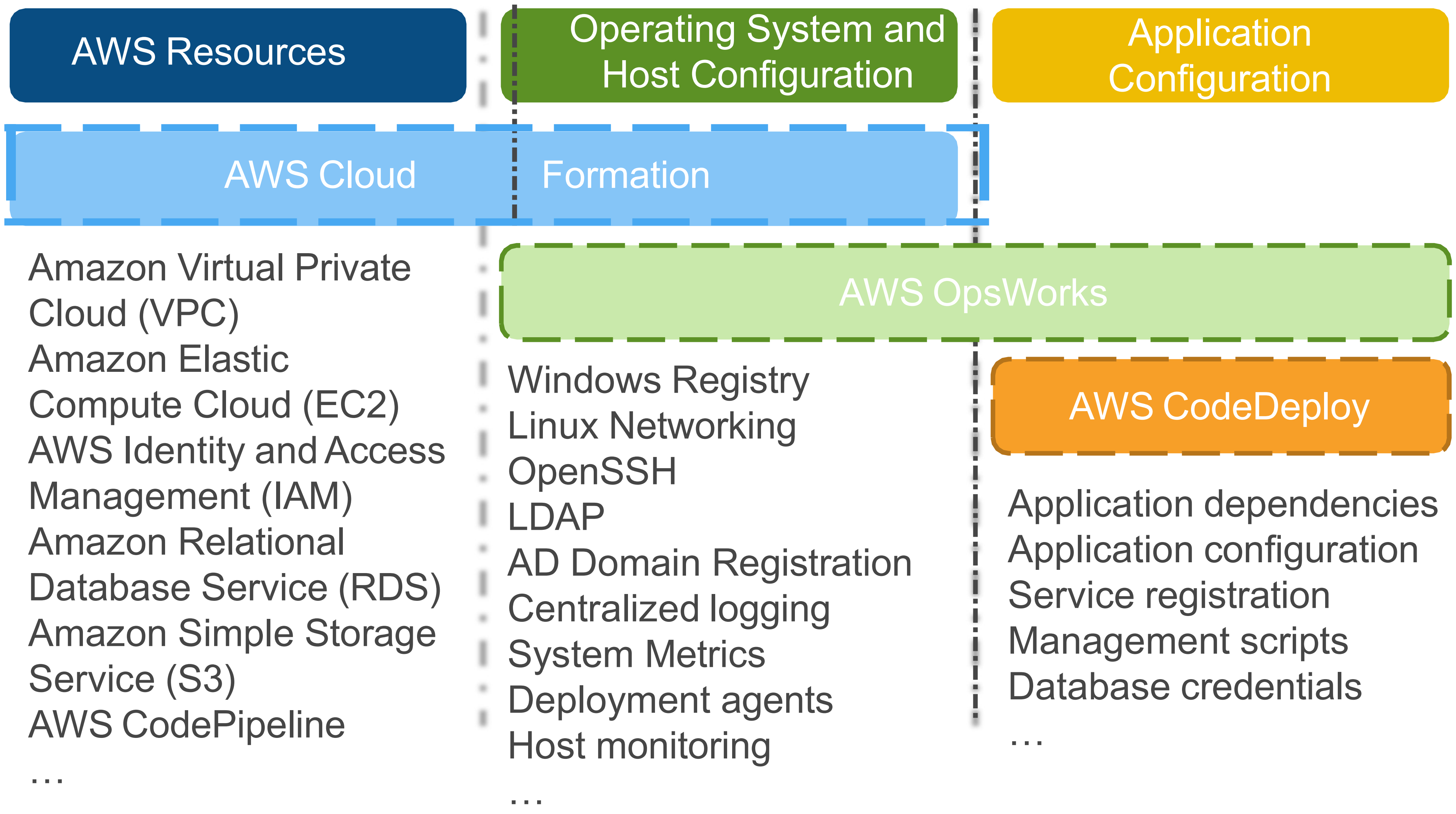
# Infrastructure as Code Workflow



**"It's all software"**

## IT IS ALL CODE


Application Configuration
Operating System and Host Configuration
Resources


AWS Resources
Operating System and Host Configuration
Application Configuration
AWS Cloud Formation
AWS OpsWorks
AWS CodeDeploy
Amazon Virtual Private Cloud (VPC)
Amazon Elastic Compute Cloud (EC2)
AWS Identity and Access Management (IAM)
Amazon Relational Database Service (RDS)
Amazon Simple Storage Service (S3)
AWS CodePipeline
…
Windows Registry
Linux Networking
OpenSSH
LDAP
AD Domain Registration
Centralized logging
System Metrics
Deployment agents
Host monitoring
…
Application dependencies
Application configuration
Service registration
Management scripts
Database credentials
…

## AWS CloudFormation

- AWS CloudFormation – more official quickstarts available
- Supports both JSON and YAML
- Need to use nested Templates to achieve re-usable (gone away since 2020 ☺ in form of modules)

- So why not AWS CloudFormation directly?
    - Terraform is cloud Agnostic. The earlier slide could be done for Azure also
    - Dynamically Create Resources (for_each, count, dynamic blocks)
    - Built-in functions

## AWS CloudFormation

Create templates of your infrastructure

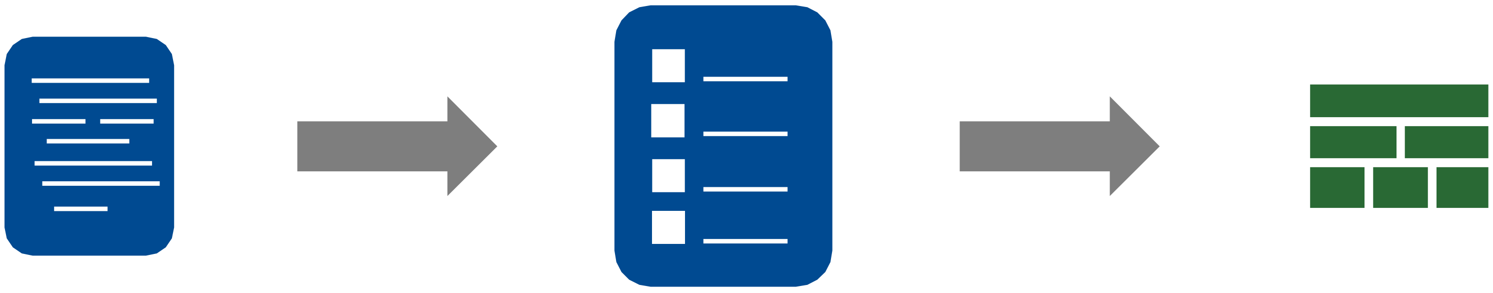
CloudFormation provisions AWS resources based on dependency needs

Version control/replicate/update templates like code

Integrates with development, CI/CD, management tools

Launched in 2010

# CloudFormation – Components & Technology



**JSON formatted file**

*Parameter definition*

*Resource creation*

*Configuration actions*

**Framework**

*Stack creation*

*Stack updates*

*Error detection and rollback*

**Configured AWS resources**

*Comprehensive service support*

*Service event aware*

*Customizable*

*Plain text*

*Perfect for version control*

# JSON

*Validatable*

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

  "Description" : "AWS CloudFormation Sample Template EC2InstanceSample: **WARNING** This template an Amazon EC2 instances. You will be billed for the AWS resources used if
you create a stack from this template.",

  "Parameters" : {
    "KeyName" : {
      "Description" : "Name of an existing EC2 KeyPair to enable SSH access to the instance",
      "Type" : "String"
    },

     "Environment": {
       "Type" : "String",
       "Default" : "Dev",
       "AllowedValues" : ["Mgmt", "Dev", "Staging", "Prod"],
        "Description" : "Environment that the instances will run in."
      }
  },

  "Mappings" : {
    "RegionMap" : {
      "us-east-1"      : { "AMI" : "ami-7f418316" },
      "us-west-2"      : { "AMI" : "ami-16fd7026" }
    }
  },

 "Conditions" : {
     "EnableEBSOptimized" : {"Fn::Equals" : [{"Ref" : " Environment "}, "Prod"]},
 },

  "Resources" : {
    "Ec2Instance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "KeyName" : { "Ref" : "KeyName" },
        "EbsOptimized " : {"Fn::If": [ " EnableEBSOptimized ", {"true"}, {"false"}]},
        "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]},
        "UserData" : { "Fn::Base64" : "80" }
      }
    }
  },

  "Outputs" : {
    "InstanceId" : {
      "Description" : "InstanceId of the newly created EC2 instance",
      "Value" : { "Ref" : "Ec2Instance" }
    },
     "PublicDNS" : {
      "Description" : "Public DNSName of the newly created EC2 instance",
      "Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PublicDnsName" ] }
    }
  }
}
```

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

  "Description" : "AWS CloudFormation Sample Template EC2InstanceSample: **WARNING** This template an Amazon EC2 instances. You will be billed for the AWS resources used if
you create a stack from this template.",
  "Parameters" : {
    "KeyName" : {
      "Description" : "Name of an existing EC2 KeyPair to enable SSH access to the instance",
      "Type" : "String"
    },

     "Environment": {
       "Type" : "String",
       "Default" : "Dev",
       "AllowedValues" : ["Mgmt", "Dev", "Staging", "Prod"],
        "Description" : "Environment that the instances will run in."
     }
  },
  "Mappings" : {
    "RegionMap" : {
      "us-east-1"      : { "AMI" : "ami-7f418316" },
      "us-west-2"      : { "AMI" : "ami-16fd7026" }
    }
  },
 "Conditions" : {
     "EnableEBSOptimized" : {"Fn::Equals" : [{"Ref" : " Environment "}, "Prod"]},
 },
  "Resources" : {
    "Ec2Instance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "KeyName" : { "Ref" : "KeyName" },
        "EbsOptimized " : {"Fn::If": [ " EnableEBSOptimized ", {"true"}, {"false"}]},
        "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]},
        "UserData" : { "Fn::Base64" : "80" }
      }
    }
  },
  "Outputs" : {
    "InstanceId" : {
      "Description" : "InstanceId of the newly created EC2 instance",
      "Value" : { "Ref" : "Ec2Instance" }
    },
     "PublicDNS" : {
      "Description" : "Public DNSName of the newly created EC2 instance",
      "Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PublicDnsName" ] }
    }
  }
}
```

HEADERS

**PARAMETERS**

**MAPPINGS**

**CONDITIONALS**

**RESOURCES**

**OUTPUTS**

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

  "Description" : "AWS CloudFormation Sample Template EC2InstanceSample: **WARNING** This template an Amazon EC2 instances. You will be billed for the AWS resources used if you create a stack from this template.",
```

Description of what your stack does, contains, etc

HEADERS

```
  "Parameters" : {
    "KeyName" : {
      "Description" : "Name of an existing EC2 KeyPair to enable SSH access to the instance",
      "Type" : "String"
    },

    "Environment": {
      "Type" : "String",
      "Default" : "Dev",
      "AllowedValues" : ["Mgmt", "Dev", "Staging", "Prod"],
       "Description" : "Environment that the instances will run in."
    }
  },
```

Provision time values that add structured flexibility and customization

PARAMETERS

```
  "Mappings" : {
    "RegionMap" : {
      "us-east-1"      : { "AMI" : "am[illegible]" },
      "us-west-2"      : { "AMI" : "ami-16fd7026" }
    }
  },
```

Pre-defined conditional case statements

MAPPINGS

```
  "Conditions" : {
    "EnableEBSOptimized" : {"Fn::Equals" : [{"Ref" : "Environment"}, "Prod"]}
  },
```

Conditional values set via evaluations of passed references

CONDITIONALS

```
  "Resources" : {
    "Ec2Instance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "KeyName" : { "Ref" : "KeyName" },
        "EbsOptimized " : {"Fn::If": [ " EnableEBSOptimized", {[illegible]}, {[illegible]}]},
        "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]},
        "UserData" : { "Fn::Base64" : "80" }
      }
    }
  },
```

AWS resource definitions

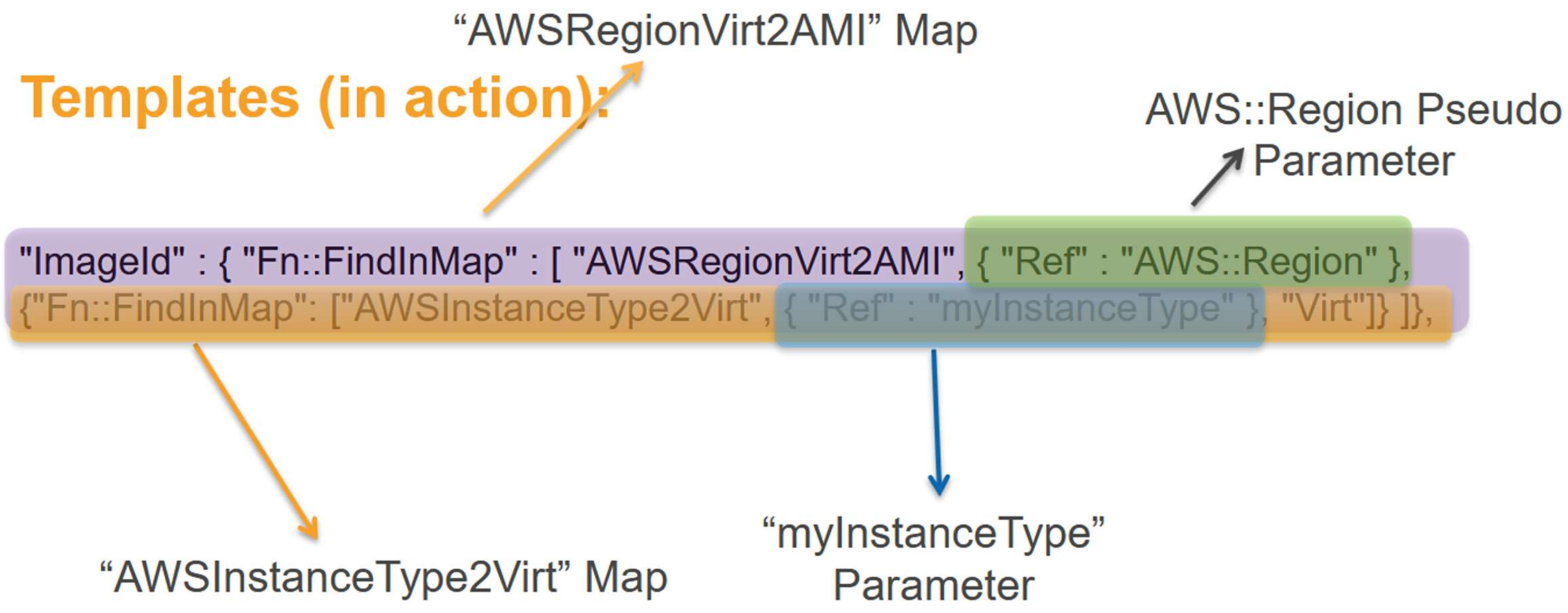
RESOURCES

```
  "Outputs" : {
    "InstanceId" : {
      "Description" : "InstanceId of the newly created EC2 instance",
      "Value" : { "Ref" : "Ec2Instance" }
    },
    "PublicDNS" : {
      "Description" : "Public DNSName of the newly created EC2 instance",
      "Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PublicDnsName" ] }
    }
  }
}
```

Resulting attributes of stack resource creation

OUTPUTS

## AWS CloudFormation



Templates (in action):

```
"ImageId" : { "Fn::FindInMap" : [ "AWSRegionVirt2AMI", { "Ref" : "AWS::Region" },
{"Fn::FindInMap": ["AWSInstanceType2Virt", { "Ref" : "myInstanceType" }, "Virt"]} ]},
```

"AWSRegionVirt2AMI" Map
AWS::Region Pseudo Parameter
"AWSInstanceType2Virt" Map
"myInstanceType" Parameter

## Mappings

### Parameters:

```
 "myInstanceType" : {
    "Type" : "String",
    "Default" : "t2.large",
    "AllowedValues" : ["t2.micro", "t2.small",
"t2.medium", "t2.large"],
    "Description" : "Instance type for instances created,
must be in the t2 family."
}
```

### Mappings:

```
"AWSInstanceType2Virt": {
      "t2.micro": {"Virt": "HVM"},
      "t2.small": {"Virt": "HVM"},
      "t2.medium": {"Virt": "HVM"},
      "t2.large": {"Virt": "HVM"},
 }
```

### Mappings:

```
"AWSRegionVirt2AMI": {
      "us-east-1": {
        "PVM": "ami-50842d38",
        "HVM": "ami-08842d60"
      },
      "us-west-2": {
        "PVM": "ami-af86c69f",
        "HVM": "ami-8786c6b7"
      },
      "us-west-1": {
        "PVM": "ami-c7a8a182",
        "HVM": "ami-cfa8a18a"
      }
    }
```

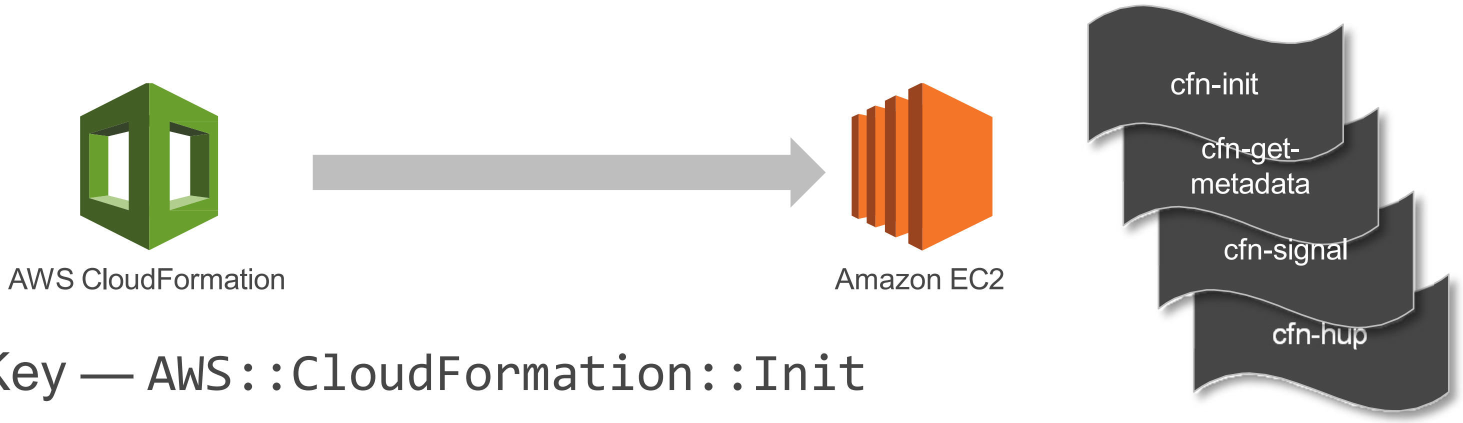
# Bootstrapping Applications

Option 1: Use EC2 UserData, which is available as a property of AWS::EC2::Instance resources

```
"Resources" : { "Ec2Instance"
    : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "KeyName" : { "Ref" : "KeyName" },
        "SecurityGroups" : [ { "Ref" : "InstanceSecurityGroup" } ],
        "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]},
        "UserData" : { "Fn::Base64" : { "Fn::Join" : ["",[
            "#!/bin/bash -ex","\n",
            "yum -y install gcc-c++ make","\n",
            "yum -y install mysql-devel sqlite-devel","\n",
            "yum -y install ruby-rdoc rubygems ruby-mysql ruby-devel","\n", "gem
            install --no-ri --no-rdoc rails","\n",
            "gem install --no-ri --no-rdoc mysql","\n", "gem
            install --no-ri --no-rdoc sqlite3","\n", "rails new
            myapp","\n",
            "cd myapp","\n",
            "rails server -d","\n"]]}}
      }
    }
```

## Bootstrapping Applications & Handling Updates

Option 2: AWS CloudFormation provides helper scripts for deployment within your EC2 instances



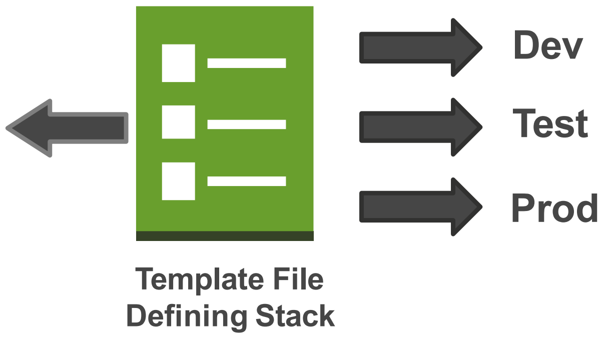
Metadata Key — `AWS::CloudFormation::Init`

Cfn-init reads this metadata key and installs the packages listed in this key (e.g., httpd, mysql, and php). Cfn-init also retrieves and expands files listed as sources.

# Multiple Stacks/Environment

Use the version control system of your choice to store and track changes to this template

**Git**
**Perforce**
**SVN**
**...**



Build out multiple environments, such as for Development, Test, Production and even DR using the same template

The entire infrastructure can be represented in an AWS CloudFormation template.

## AWS CloudFormation

- **Versioning!**
- You track changes within your code Do it within your infrastructure!
    - What is changing?
    - Who made that change?
    - When was it made?
    - Why was it made?(tied to ticket/bug/project systems?)

    - **Self imposed, but you need to be doing this!**

## AWS CloudFormation

- **Testing:**
  - Validate via API/CLI
    - $ aws cloudformation validate-template – confirm CF syntax
    - Use something like Jsonlint (http://jsonlint.com/) to find JSON issues like missing commas, brackets!
  - Throw this into your testing/continuous integration pipelines!

## AWS CloudFormation

- **Deploy & update via console or API/command line:**
  - Just a couple of clicks OR
  - aws cloudformation create-stack --stack-name myteststack --template-body file:////home//local//test//sampletemplate.json -- parameters ParameterKey=string,ParameterValue=string

## What happens once deployed?

## Ongoing Management

- Updates/patches?
- New software?
- New configurations?
- New code deploys?
- Pool specific changes?
- Environment specific changes?
- Run commands across all hosts?
- Be on top of all running resources?

## Select Right Tool

- Can be done with AWS CloudFormation – maintenance will be harder.
- Consider AWS OpsWork and Chef

## Infrastructure-as-Code (IAC) Benefits

- ✓ Reproducible Environments
- ✓ Automation – CI/CD
- ✓ Trackable – GitHub
- ✓ Language – HCL (Hashicorp Configuration Language)
- ✓ Workflow
- ✓ Multiple Providers
- ✓ Can provide similar configuration across multiple cloud vendors
- ✓ Similar infra across environments

## What is Terraform?

- A templating language **H**ashiCorp **C**onfiguration **L**anguage (**HCL**)
- A tool that can be used to orchestrate the provisioning of:
  - Public clouds (Azure, AWS, GCP, Oracle, Alibaba)
  - On-premises (VMware)
  - Other (Cisco, GitHub, GitLab, New Relic, Okta, Rabbit MQ)
- Uses State files (more on this later)
- Is **NOT** used for configuration
  - PowerShell Desired State Configuration (DSC), Chef, Puppet, Ansible

## TERRAFORM <> ANSIBLE

- TERRAFORM – IAC
- ANSIBLE – CONFIGURATION MANAGEMENT
- ANSIBLE
  - Makes configuration management and application deployment easier
- TERRAFORM
  - building, updating, and **versioning** **infrastructure** safely and effectively

## Terraform

- Clean and easy code to write and maintain
- Fully declarative configuration
- Version control on infrastructure
- Implicit dependencies management – explicit can be forced
- Ecosystem of providers and skilled personnel

## Terraform

- Multiple Cloud Providers
- Declarative Configuration
- Execution Plans
- Resource Graphs
- State Management
- Loops, Conditionals, Dynamic Blocks
- ....

# Terraform

Check Point
SOFTWARE TECHNOLOGIES LTD.

Bitbucket

CISCO

Akamai

Azure

aws

paloalto
NETWORKS

HELM

ORACLE
CLOUD

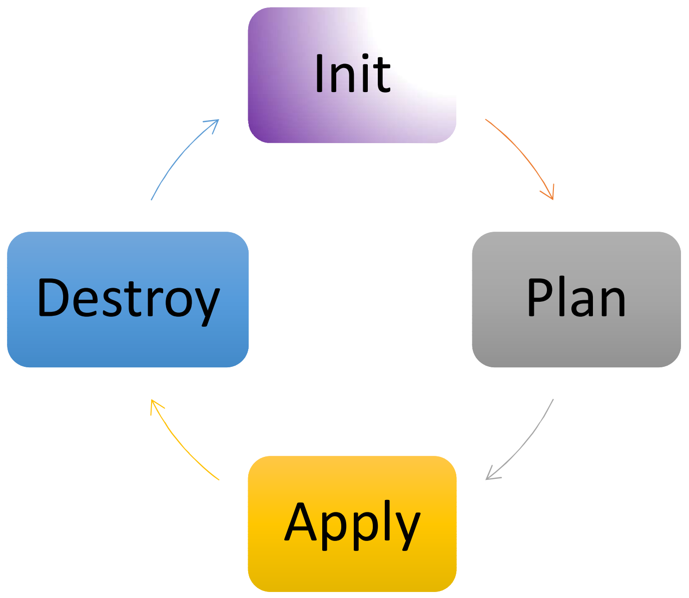
Alibaba Cloud

## Getting Started

- Installation on Windows is pretty straightforward
- Download AMD64 version for Windows from here.
- Unzip the file and copy terraform to a folder that exists in PATH

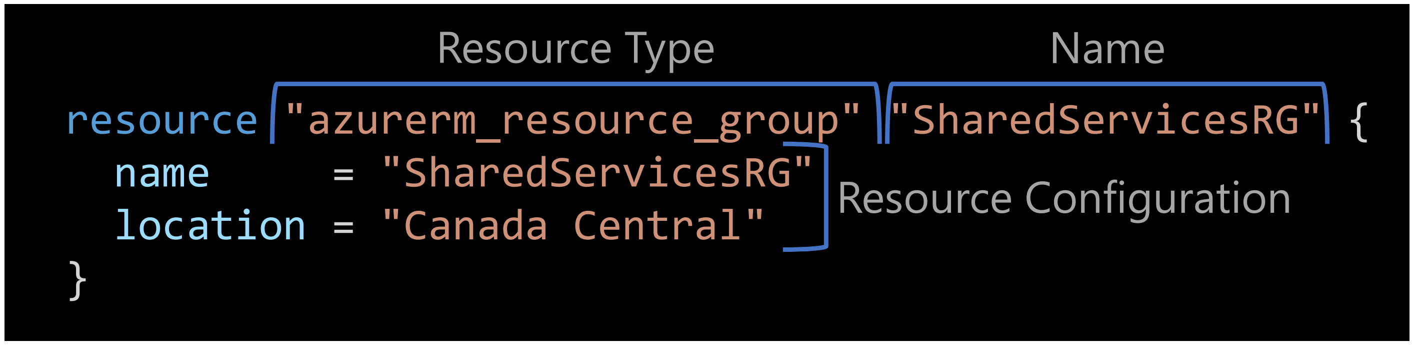
## How does it work?

- Terraform **init**
  - Initializes the current working directory
- Terraform **plan**
  - Execution plan to validate against existing environment
- Terraform **apply**
  - Deploys and updates resources
- Terraform **destroy**
  - Removes all resources defined in a configuration



## Resources

- Resources (e.g. an NGINX server) have
  - Type
  - A **required Provider**
  - Name (internal to Terraform)
  - Configuration with Deployment Details



```
resource "azurerm_resource_group" "SharedServicesRG" {
  name     = "SharedServicesRG"
  location = "Canada Central"
}
```

## Backends

- Determines how state is loaded and how an operation such as **apply** is executed
- By default, Terraform uses the "local" backend
- Benefits of backends:
  - Working in a team
    - Store state remotely and protect state with locks to prevent corruption
  - Keeping sensitive information off disk
    - Retrieved on demand and only stored in memory
  - Remote operations
    - Remote execution

## State

- State keeps track of the all managed resources and their associated properties with current values.
- **Essential for managing changes to infrastructure over time**
- Preserve the state file for the entire life cycle of the resources
  - You can create a separate state file per resource group, application, shared service (ie. core networking)
  - Terraform Workspaces should also be used to separate application and environment boundaries
- Recommended to use a **remote backend** to save state in centralized, secure storage
  - Example: Storage account, Terraform Cloud, Terraform Enterprise, Artifactory, Consul
- You must initialize the Terraform State
  - This is what **terraform init** does

## State

**Secrets (like usernames/passwords, access keys/tokens, etc.) can be written to your state file!**

## Providers

- The provider block is used to configure the named provider
- Is responsible for creating and managing resources, and for all other interactions including authentication
- The version argument is optional, but recommended

```
provider "azurerm" {
  version = ">=2.0.0"
  subscription_id = "<<REMOVED>>"
  client_id       = "<<REMOVED>>"
  client_secret   = "<<REMOVED>>"
  tenant_id       = "<<REMOVED>>"
}
```

# Variables

- Parameterize the configurations
- If no value is assigned to a variable and the variable has a default key in its declaration, that value will be used for the variable
- Can be provided…
  - Within the Terraform template
  - Within its own Terraform template file
  - Within a .TFVARS files
  - Through command-line
  - Through Environment variables

## Variables

```
variable "location" {
  type        = string
  default     = "westus"
  description = "Specify a location see: az acc
ount list-locations -o table"
}


resource "azurerm_resource_group" "example" {

  name = var.resource_group_name

  location = var.location

}
```

### Variable Precedence

a. Command Line
b. From a File
c. From Environment Variables
d. UI Input

## Dependencies

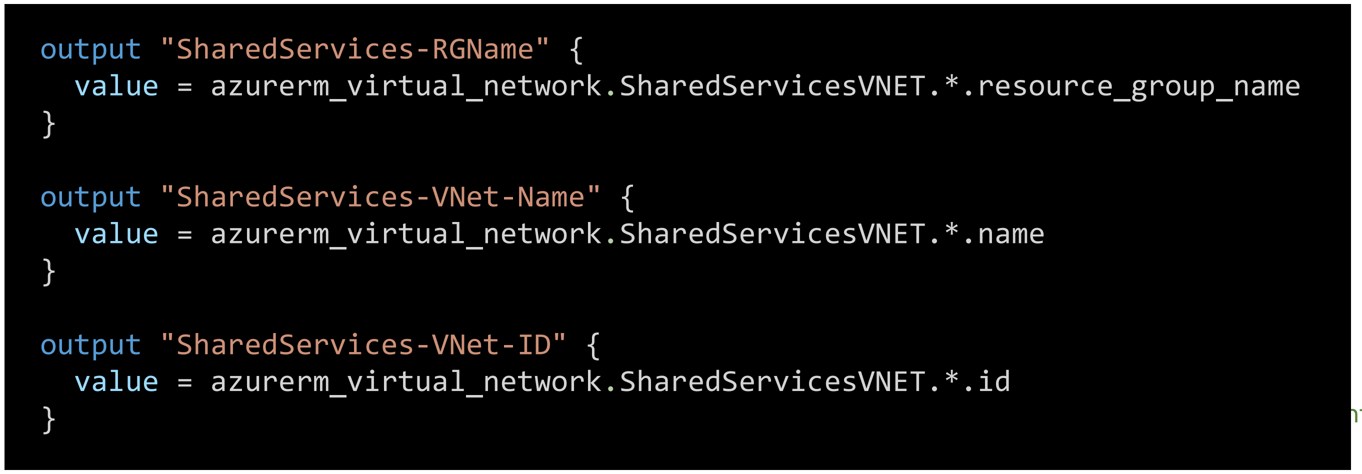
- *Implicit* dependencies, which Terraform and the provider determine automatically based on the configuration
- *Explicit* dependencies, which you define using the depends_on meta-argument

# Dependencies

```
resource "azurerm_resource_group" "example" {

  name = var.resource_group_name

  location = var.location

}

resource "azurerm_storage_account" "example" {

  name = "storageaccountname"

  resource_group_name = azurerm_resource_group.example.name

  location = azurerm_resource_group.example.location

  account_tier = "Standard"

  account_replication_type = "GRS"
  tags = {

    environment = "staging"

  }

}
```

## Outputs

- Used to organize data to be easily queried and shown back to the Terraform user
- Data is outputted when apply is called
- Outputs can be queried after a run using the terraform output <<output name>> command



```
output "SharedServices-RGName" {
  value = azurerm_virtual_network.SharedServicesVNET.*.resource_group_name
}

output "SharedServices-VNet-Name" {
  value = azurerm_virtual_network.SharedServicesVNET.*.name
}

output "SharedServices-VNet-ID" {
  value = azurerm_virtual_network.SharedServicesVNET.*.id
}
```

## Best Practices

- Use remote backends
- Manage Terraform, providers and modules versions
- Use implicit dependencies
- Use modules (custom or from the HashiCorp public registry https://registry.terraform.io)
- Use ARM templates only if you don't have another choice

## Terraform – Adv. State and Secrets

1. **Pre-requisite #1:** Don’t Store Secrets in Plain Text
2. **Pre-requisite #2**: Keep Your Terraform State Secure

1. **Technique #1:** Environment Variables
2. **Technique #2:** Encrypted Files (e.g., KMS, PGP, SOPS)
3. **Technique #3:** Secret Stores (e.g., Vault, AWS Secrets manager)

## Terraform – Adv. State and Secrets

- Method #1: Terraform has native support for Environment Variables

```
variable "username" {
  description = "The username for the DB master user"
  type       = string
  sensitive = true
}variable "password" {
  description = "The password for the DB master user"
  type       = string
  sensitive = true
}
```

```
resource "aws_db_instance" "example" {
  engine              = "mysql"
  engine_version      = "5.7"
  instance_class      = "db.t2.micro"
  name                = "example"  # Set the secrets from
variables
  username            = var.username
  password            = var.password
}
```

```
export TF_VAR_username='someusername'
export TF_VAR_password='Somepassword'
# neither user name nor password will be found in Version Control
```

## Terraform – Adv. State and Secrets

- Method #2: Use tools for storing sensitive secrets e.g pass

```
$ pass insert  db_username
Enter password for db_user:
$ pass insert db_password
Enter password for db_password
```

- You still have to remember the password ☺.
- Secrets can be read as:

```
$ pass db_username
```

```
export TF_VAR_username=$(pass db_username)
export TF_VAR_password=$(pass db_password)
# neither user name nor password will be found in Version Control
```

## Terraform – Adv. State and Secrets

- Use either
  - AWS KMS
  - Azure Key Vault
  - GCP KMS
- Basic process consists of creating a YML file like below (This file **SHOULD NOT BE** version controlled)

```
username: someuser
password: somepassword
```

## Terraform – Adv. State and Secrets

- Encrypt the file using the corresponding AWS/GCP/Azure KMS Key
- Check in the encrypted file into version control.
- Use the encrypted file like so:

```
data "aws_kms_secrets" "creds" {
  secret {
    name    = "db"
    payload = file("${path.module}/db-creds.yml.encrypted")
  }
}
locals {
  db_creds = yamldecode(data.aws_kms_secrets.creds.plaintext["db"])
}
…
Username = locals.db_creds.username
```

## Terraform – Adv. State and Secrets

- Use SOPS and Terragrunt (out of scope) to simplify the workflow of editing and working with secret files.

## Modular Code

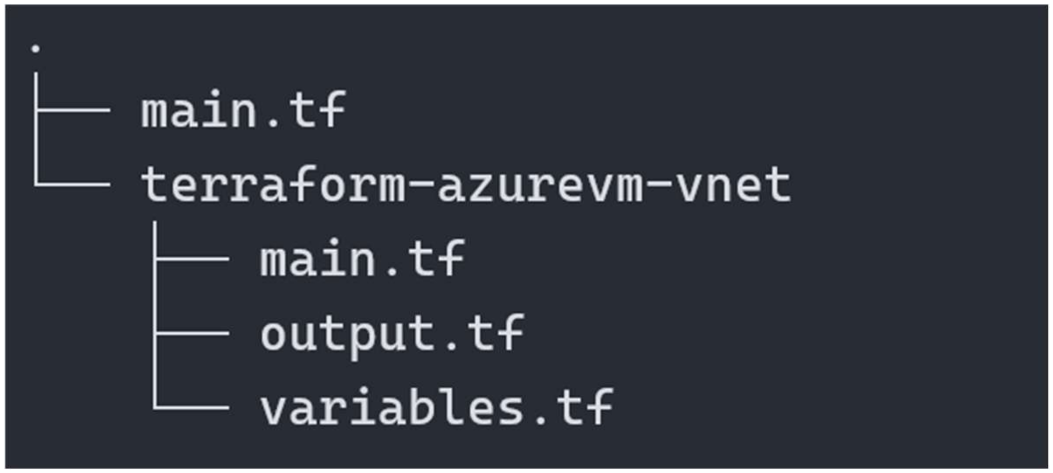
- A *module* is a container for multiple resources that are used together.
- You can use modules to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.
- The .tf files in your working directory when you run terraform plan or terraform apply together form the root module.
- That module may call other modules and connect them together by passing output values from one to input values of another.

## Modular Code - Structure

- Modules use
  - Input Variables
    - Accept Values from calling module
  - Output Values
    - Send Values to calling module
  - Resources
    - Define one or more infra objects
- Each Module is in a separate directory
- Modules can be loaded locally or from a remote repository

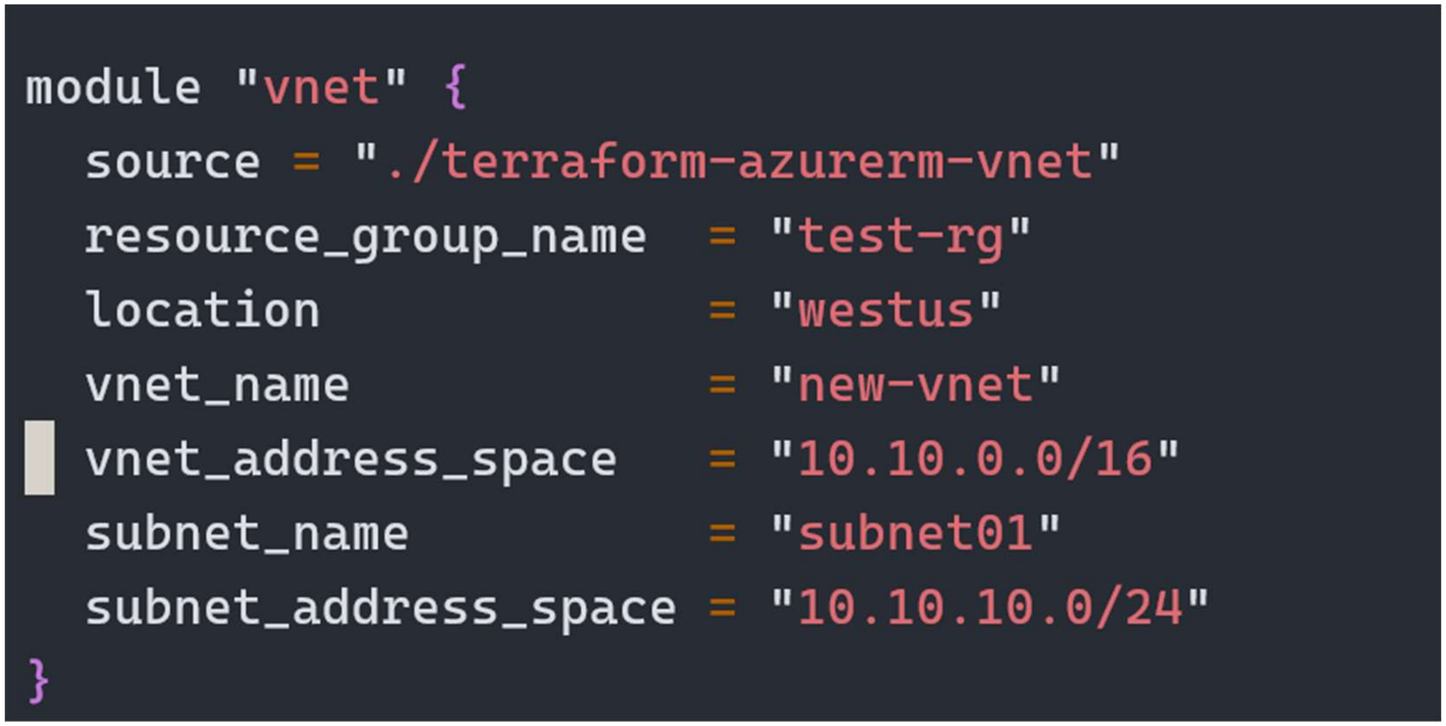
## Modular Code - Structure

- Modules can call other modules using module block
- Examples of a module

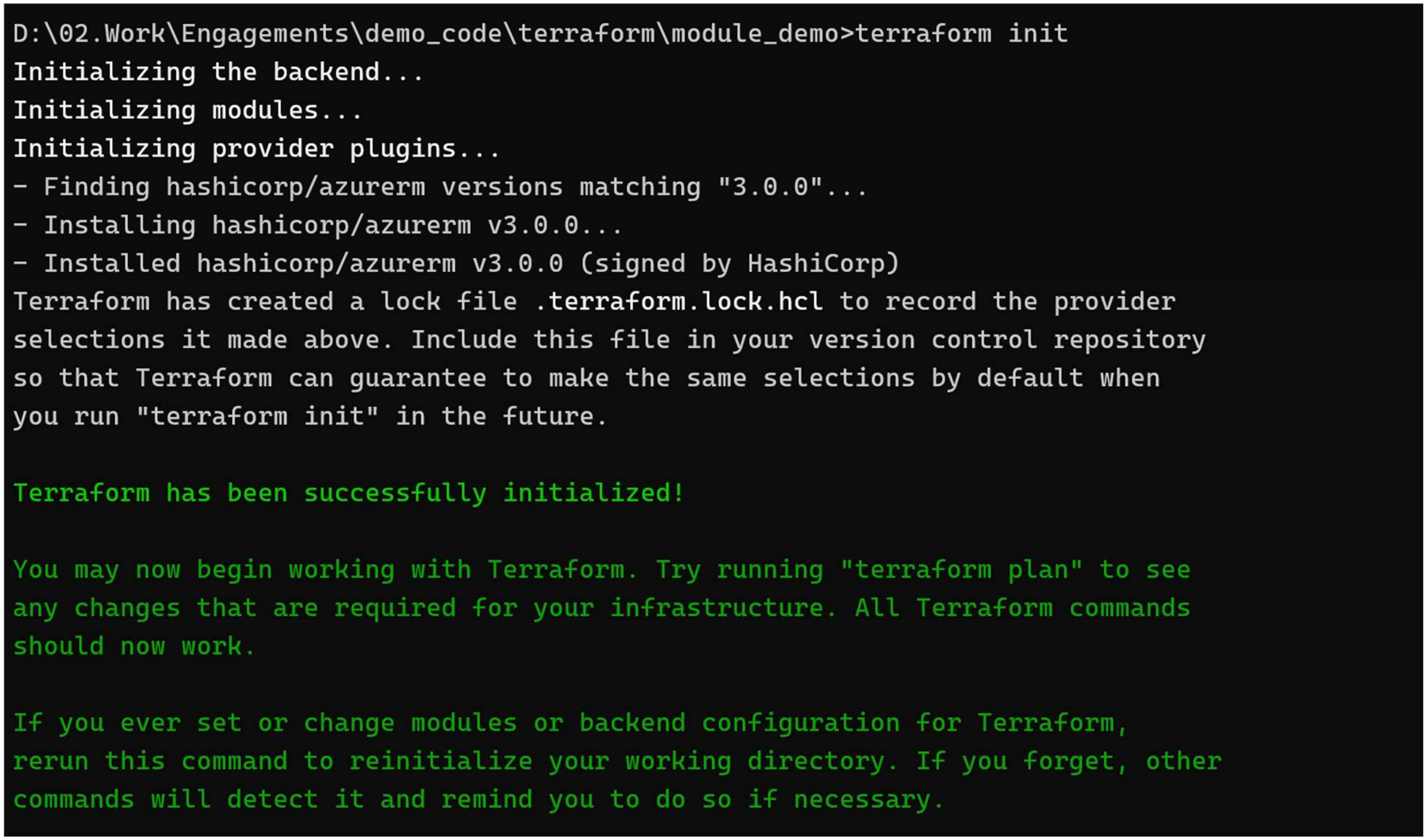


```
.
├── main.tf
└── terraform-azurevm-vnet
    ├── main.tf
    ├── output.tf
    └── variables.tf
```

## Modular Code - Structure



```
module "vnet" {
  source = "./terraform-azurerm-vnet"
  resource_group_name   = "test-rg"
  location              = "westus"
  vnet_name             = "new-vnet"
  vnet_address_space    = "10.10.0.0/16"
  subnet_name           = "subnet01"
  subnet_address_space = "10.10.10.0/24"
}
```



```
D:\02.Work\Engagements\demo_code\terraform\module_demo>terraform init
Initializing the backend...
Initializing modules...
Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "3.0.0"...
- Installing hashicorp/azurerm v3.0.0...
- Installed hashicorp/azurerm v3.0.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

## Modular Code

- Now you can plan and apply.
- Note for this to work, you will need to install the AZ CLI and have a valid Azure Subscription.

## Demo Time

Simple Terraform Deployments

## Q &A

## We're done!

## Thank you for your time and participation.