Terraform

Terraform Course Outline

What is Terraform?

- Infrastructure as a Code
- Automation of Infrastructure
- Keep your Infrastructure in certain state (Compliant)
  - E.G. 2 Web Instances with 2 Volumes and 1 load Balancer
- Make your Infrastructure auditable
  - You can keep your infrastructure change history in version control system like GIT

Terraform

- Ansible, Chef, Puppet, Saltstack have focus on automating **installation and configuration of** software
  - Keeping the **machine** in compliance, in a certain state
- Terraform can automate provisioning of the **infrastructure itself**
  - **E.g. Using AWS/Azure/Google Cloud/DigitalOcean**
  - Works well with automation software like ansible to install software after the infrastructure is provisioned

Installing Terraform

- https://www.terraform.io/downloads.html
- https://releases.hashicorp.com/terraform/0.12.9/terraform_0.12.9_linux_amd64.zip
  - yum update -y
  - yum install wget -y
  - wget -q https://releases.hashicorp.com/terraform/0.12.9/terraform_0.12.9_linux_amd64.zip
  - yum install unzip -y
  - unzip -o terraform_0.12.9_linux_amd64.zip -d /usr/local/bin
  - rm terraform_0.12.9_linux_amd64.zip

Spin AWS Instance using Terraform

- https://www.terraform.io/downloads.html
  - https://releases.hashicorp.com/terraform/0.12.9/terraform_0.12.9_linux_amd64.zip
  - yum update -y
  - yum install wget -y
  - wget -q https://releases.hashicorp.com/terraform/0.12.9/terraform_0.12.9_linux_amd64.zip
  - yum install unzip -y
  - unzip -o terraform_0.12.9_linux_amd64.zip -d /usr/local/bin
  - rm terraform_0.12.9_linux_amd64.zip

Spin AWS Instance using Terraform

- Open AWS Account
- Create IAM admin user
- Create Terraform file to spin up t2.micro instance
- Run terraform apply

Terraform HCL

- HCL – Hashocorp Configuration Language

“${var.mymap[“mykey”]}” ,

element(var.mylist, 1)

Slice(var.mylist, 0, 2)

```
variable "myvar" {
  type = "string"
  default = "hello terraform"
}

variable "mymap" {
  type = map(string)
  default = {
    mykey = "my value"
  }
}

variable "mylist" {
  type = list
  default = [1,2,3]
}
~
~
```

```
->> terraform-test $ vim main.tf
->> terraform-test $ terraform version
Terraform v0.12.12
->> terraform-test $ terraform console
> var.myvar
hello terraform
> "${var.myvar}"
hello terraform
> exit
->> terraform-test $ vim main.tf

[No write since last change]
No manual entry for }

shell returned 1

Press ENTER or type command to continue
->> terraform-test $ terraform console
> var.myvar
hello terraform
> var.mymap
{
  "mykey" = "my value"
}
```

Variables

- Terraform variables completed reworked in 0.12 release
- You can now have more control over variables and have for and for-each loops which is not possible in earlier version
- You do not have to specify types in variables but it is recommended
- Terraform’s Simple Variable Type
  - String
  - Number
  - Bool

Variables

- Terraform’s Complex Variable Type
  - List(type)
  - Set(type)
  - Map(type)
  - Object({<ATTR_NAME>=<TYPE>,…}
  - Tuple([<TYPE>,…])
    - List: [0,1,5,2]
    - Map: {“key” = “value”}
- A list is always ordered it always returned 0,1,5,2 not 5,1,2,0
- A “set” is like list but doesn’t keep the order you put in and can only contain unique values
  - In set it become [1,2,5]

Variables

- Object is like a map but each element can have different type
- For example

```
{
  firstname = “John”,
 Housenumber = 10
}
```

- A tuple is like list but each element can have different type

```
[0, “string”, false]
```

- The most common types are List and Map the other ones are only used sporadically
- The one you most remember is string, number, bool and the list & map

Variables

- Everything in one file is not great
- Use variable to **hide secrets**
  - You don’t’ have to store aws creds in git repo
- Use Variable that **might change**
  - AMIs are different per region
- Use Variable to make yourself easier to reuse terraform files

Variables

instance.tf

```
provider "aws" {
  access_key = "ACCESS_KEY_HERE"
  secret_key = "SECRET_KEY_HERE"
  region     = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-0d729a60"
  instance_type = "t2.micro"
}
```

Variables

provider.tf

```
provider "aws" {
    access_key = "${var.AWS_ACCESS_KEY}"
    secret_key = "${var.AWS_SECRET_KEY}"
    region = "${var.AWS_REGION}"
}
```

vars.tf

```
variable "AWS_ACCESS_KEY" {}
variable "AWS_SECRET_KEY" {}
variable "AWS_REGION" {
  default = "eu-west-1"
}
```

terraform.tfvars

```
AWS_ACCESS_KEY = ""
AWS_SECRET_KEY = ""
AWS_REGION = ""
```

instance.tf

```
resource "aws_instance" "example" {
  ami           = "ami-0d729a60"
  instance_type = "t2.micro"
}
```

Variables

provider.tf

```
provider "aws" {
    access_key = "${var.AWS_ACCESS_KEY}"
    secret_key = "${var.AWS_SECRET_KEY}"
    region = "${var.AWS_REGION}"
}
```

instance.tf

```
resource "aws_instance" "example" {
  ami           = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"
}
```

vars.tf

```
variable "AWS_ACCESS_KEY" {}
variable "AWS_SECRET_KEY" {}
variable "AWS_REGION" {
  default = "eu-west-1"
}
variable "AMIS" {
  type = "map"
  default = {
    us-east-1 = "ami-13be557e"
    us-west-2 = "ami-06b94666"
    eu-west-1 = "ami-0d729a60"
  }
}
```

https://cloud-images.ubuntu.com/locator/ec2/

Software Provisioning

- Open AWS Account
- Create IAM admin user
- Create Terraform file to spin up t2.micro instance
- Run terraform apply

Outputting Attributes

- Terraform keeps attribute of all the resources you create
  - E.g. the aws_instance resource has the attribute public_ip
- Those attribute can be queried and outputted
- This can be useful just to output valuable information or to feed information to external software

Outputting Attributes

- Use “output” to display the public IP address of an AWS resource:

```
resource "aws_instance" "example" {
  ami           = “${lookup(var.AMIS, var.AWS_REGION)}”
  instance_type = “t2.micro"
}

output "ip" {
    value = “${aws_instance.example.public_ip}”
}
```

- You can refer to any attribute by specifying the following elements in your variable:
  - The resource type: aws_instance
  - The resource name: example

Outputting Attributes

- You can also use the attributes in a **script**:

```
resource "aws_instance" "example" {
  ami           = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"
  provisioner "local-exec" {
      command = "echo ${aws_instance.example.private_ip} >> private_ips.txt"
    }
}
```

- Useful for instance to start automation scripts after infrastructure provisioning
- You can populate the IP addresses in an **ansible host** file

Outputting Attributes

Remote State

- Terraform keeps **remote state** of the infrastructure
- It is stored in file called **terraform.tfstate**
- There is also backup of previous state in **terraform.tfstate.backup**
- When we execute terraform **apply**, a new terraform.tfstate and backup is written
- This is how terraform keeps track of **remote state**
  - If remote state changes and you hit terraform apply again terraform will make changes to meet the **correct remote state** again
  - E.g. if you **terminate** an instance that managed by terraform after terraform apply it will be **started** again

Remote State

- You can keep the terraform.tfstate in **version control**
  - E.g. git
- It gives you a **history** of your terraform.tfstate file (which is just a big json file)
- It allows you to **collaborate** with other team members
  - Unfortunately you can get conflicts when 2 people work at the same time
- Local state works well in the beginning, but when your project becomes bigger, you might want to store your state **remote**

Remote State

- The terraform state can be saved remote, using the **backend** functionality in terraform
- The default is a **local backend** (the local terraform state file)
- Other backend includes
  - **s3**(with locking mechanism using DynamoDB)
  - **Consul**(with locking)
  - **terraform enterprise** (the commercial solution)

Remote State

- Using backend functionality has definitely beinifits:
  - Working in a team: it allows for **collaboration**, the remote state will always be **available** for the whole team
  - The state file is store locally. Possible **sensitive information** is now only stored in the remote state
  - Some backends will enable **remote operations.** The terraform apply will then run completely remote. These are called enhanced backend.

  https://www.terraform.io/docs/backends/types/index.html

Remote State

- There are 2 steps to configure a remote state:
  - Add the backend code to a .tf file
  - Run the initialization process
- To configure a consul remote state, you can add a file backend.tf with the following contents:

```
terraform {
  backend “consul” {
    address=“default.consul.io” #hostname of consul cluster
    path     = “terraform/myproject”
}
}
```

Remote State

- You an also store state in S3

```
terraform {
  backend “s3” {
    bucket = “mybucket”
    key      = “terraform/myproject” #directory in bucket
     region = “us-east-2”
}
}
```

- When using S3 remote state. It’s best to configure the AWS credentials:

```
aws configure
```

- Next step, **terraform init**:

Remote State

- Using a remote store for the terraform state will ensure that you always have the **latest version** of the state
- It avoids to **commit and push** the terraform.tfstate to version control
- Terraform remote state don’t always support locking
  - Documentation always mentions if locking is available for remote store.
  - S3 and consul both support remote store

Remote State

- You can also specify a (read-only) remote store directly in the .tf file

```
data "terraform_remote_state" "aws-state" {
  backend = "s3"
  config {
    bucket = "terraform-state"
    key = "terraform.tfstate"
    access_key = "${var.AWS_ACCESS_KEY}"
    secret_key = "${var.AWS_SECRET_KEY}"
    region = "${var.AWS_REGION}"
  }
}
```

- This is only useful as read only feed from your remote file
- Actually it's a data source
- Useful to generate outputs

Demo: Remote State

- Create s3 bucket
- aws configure
  - As in case of backed we can not use access key as variables
  - **dnf install python3-pip**
  - **pip3 install awscli --upgrade –user**
  - **vi ~/.bash_profile**

    export PATH=~/.local/bin:$PATH

    source ~/.bash_profile
  - **aws --version**
  - **aws configure**

Datasources

- For certain providers (like AWS), terraform provides data sources
- Data sources provides you with dynamic information
  - A lot of data is available by AWS in structured format using their API
  - Terraform also exposes this information using data sources
- Examples:
  - List of AMIs
  - List of Availability Zones

Datasources

- Another Example is the data source that gives you all IP address in use by AWS
- This is useful if you want to filter traffic based on an AWS region
- We could manage security groups as well

Template Provider

- Template provider can help creating **customized configuration files**
- You can build **templates based on variables** from terraform resource attributes (e.g. a public IP address)
- The result is a string that can be used as a variable in terraform
  - The string contains a template
  - E.g. a configuration file
- Can be used to create generic templates or cloud init configs
- If we want to pass user-data that depends on other information in terraform (e.g. IP address) you can use the provider template

Template Provider

- First you create a template file

```
#! /bin/bash
echo "database-ip = ${myip} >> /etc/myapp.config
```

- Then we will create template file resource that will read the template file and replace ${myip} with the IP address of an AWS instance created by terraform

```
data "template_file" "my-template"{
 template = "${file("templates/init.tpl")}"
  vars {
    myip = "${aws.instance_database1.private_ip}"
         }
  }
```

Template Provider

- Then you can use the my template resource when creating a new instance

```
#Create a Web server
resource "aws_instance" "web"{
  user_data = "${data.template_file.my-template.rendered}"
}
```

- When terraform runs it will see that it needs to spin up the database1 instance then generate the template and only then generate the web instance

- The web instance will have template injected in user_data and when it launches the user_data will create a file /etc/myapp.config with the ip address of the database

Other Provider

- Terraform is a tool to create and manage infrastructure resources
- Terraform has many provider to choose from
  - AWS the most popular one
  - Potentially any company that opens API can be used as a terraform provider
- Some of the popular Cloud Provider
  - Google Cloud
  - Azure
  - Heroku
  - Digital Ocean
- An on premise/Private Cloud:
  - Vmware Cloud/vsphere/OpenStack
- https://www.terraform.io/docs/providers/

Modules

- You can use modules to make your terraform more organized
- Use third party modules
  - Modules from github
- Reuse parts of your code
  - E.g. to setup network in AWS – the Virtual Private Network (VPC)

Modules

- Use a module from git

```
module “module-example” {
    source = “github.com/maheshkharwadkar/terraform-module-example”
}
```

- Use a module from a local folder

```
module “module-example” {
        source= “./module-example”
}
```

- Pass Argument to module

```
module “module-example” {
        source = “./module-example”
        region = “us-east-1”
        ip-range = “10.0.0.0/8”
        cluster-size = 3
}
```

Modules

- Pass arguments to the module

```
module "module-example" {
   source = "./module-example"
   region = "us-west-1"
   ip-range = "10.0.0.0/8"
   cluster-size = "3"
}
```

- Use the **output** from the module in the main part of your code:

```
output "some-output" {
   value = "${module.module-example.aws-cluster}"
}
```

- Inside the module folder, you just have again terraform files:

module-example/vars.tf

```
variable "region" {} # the input parameters
variable "ip-range" {}
variable "cluster-size" {}
```

module-example/cluster.tf

```
# vars can be used here
resource "aws_instance" "instance-1" { ... }
resource "aws_instance" "instance-2" { ... }
resource "aws_instance" "instance-3" { ... }
```

module-example/output.tf

```
output "aws-cluster" {
   value = "${aws_instance.instance-1.public_ip},${aws_instance.instance-2.public_ip},${aws_instance.instance-2.public_ip}"
}
```

Terraform Command Overview

- Terraform is very much focused on the **resource definition**
- It has **limited toolset** available to modify, import, create these resource definition
  - Still every new release there are new features coming out to make it easier to handle your resource

Terraform Command Overview

Command	Description
terraform apply	Applies State
destroy	Destroys all terraform managed state
fmt	Rewrite terraform config files to canonical format and style
get	Download and update modules
graph	Create visual representation of configuration or execution plan
Import [options] ADDRESSID	Import will try and find the infrastructure resource identified with ID and import the state into terraform.tfstate with resource id ADDRESS
output [options] [name]	Output any of your resource. Using NAME will only output a specific resource.
plan	Shows changes to be made to the infrastructure
refresh	Refresh remote state. Can identify differences between state file and remote state

Interpolation

- Terraform

Functions

- You can use build in functions in your terraform resources
- These functions are called with syntax **name(arg1, arg2, …)** and wrapped with ${…}
  - For example **${file(“mykey.pub”)} would** read the contents of the public key file. Here **file is a function.**
- Ref: https://www.terraform.io/docs/configuration-0-11/interpolation.html

Terraform Functions

Function	Description	Example
basename(path)	Returns the filename (last element) of a path	basename("/home/edward/file.txt") returns **file.txt**
coalesce(string1, string2, ...) coalescelist(list1, list2, ...)	Returns the first non-empty value Returns the first non-empty list	coalesce("", "", "hello") returns **hello**
element(list, index)	Returns a single element from a list at the given index	element(module.vpc.public_subnets, count.index)
format(format, args, ...) formatlist(format, args, ...)	Formats a string/list according to the given format	format("server-%03d", count.index + 1) returns **server-001**, **server-002**,

Terraform Functions

Function	Description	Example
index(list, elem)	Finds the index of a given element in a list	index(aws_instance.foo.*.tags.Env, "prod")
join(delim, list)	Joins a list together with a delimiter	join(“,”, var.AMIS) returns “**ami-123,ami-456,ami-789**”
list(item1, item2, …)	create a new list	join(":", list(“a","b","c")) returns **a:b:c**
lookup(map, key, [default])	Perform a lookup on a map, using “key”. Returns value representing “key” in the map	lookup(map("k", "v"), "k", "not found") returns “**v**”

Terraform Functions

Function	Description	Example
lower(string)	returns lowercase value of “string”	lower(“Hello”) returns **hello**
map(key, value, …)	returns a new map using key:value	map("k", “v”, “k2”, “v2”) returns: **{ "k" = “v”, “k2” = “v2” }**
merge(map1, map2, …)	Merges maps (union)	merge(map("k", “v”), map(“k2”, “v2”)) returns: **{ "k" = “v”, “k2” = “v2” }**
replace(string, search, replace)	Performs a search and replace on string	replace("aaab", "a", "b") returns **bbbb**

Terraform Functions

Function	Description	Example
split(delim, string)	splits a string into a list	split(",", “a,b,c,d") returns **[“a", “b", "c", “d”]**
substr(string, offset, length)	extract substring from string	substr("abcde", -3, 3) returns **cde**
timestamp()	returns RFC 3339 timestamp	"Server started at ${timestamp()}" Server started at 2018-06-16T18:46:46Z
upper(string)	Returns uppercased string	upper(“string") returns **STRING**

Terraform Functions

Function	Description	Example
uuid()	Returns a UUID string in RFC 4122 v4 format	uuid() returns: **65b8cf0a-685d-3295-73c1-1393ef71bcd6**
values(map)	returns values of a map	values(map("k","v","k2","v2")) returns **["v", "v2"]**

Terraform With AWS

instance.tf

```
resource "aws_instance" "example" {
  ...
}

resource "aws_ebs_volume" "ebs-volume-1" {
    availability_zone = "eu-west-1a"
    size = 20
    type = "gp2" # General Purpose storage, can also be standard or io1 or st1
    tags {
        Name = "extra volume data"
    }
}

resource "aws_volume_attachment" "ebs-volume-1-attachment" {
  device_name = "/dev/xvdh"
  volume_id = "${aws_ebs_volume.ebs-volume-1.id}"
  instance_id = "${aws_instance.example.id}"
}
```

Terraform With AWS

- In the previous example we added an **extra** volume
- The **root volume** of 8 GB still exists
- If you want to increase the storage or type of the root volume, you can use **root_block_device** within the aws_instance resource

```
resource "aws_instance" "example" {
  ...
  root_block_device {
    volume_size = 16
    volume_type = “gp2"
    delete_on_termination = true  # whether to delete the root block device when the instance gets terminated or not
  }
}
```

Terraform With Loops

- Starting terraform 0.12 we can use **for** and **for_each** loop
- The for-loop feature scan help you **to loop over variables** , teranform it and output it in a different format
- For example:
  - [for s in [“this is a”, “list”] : upper(s)]
  - You can loop over list [1,2,3,4] or map like {“key” = “value”}
  - We can transform them by doing calculation or a string operation
  - Then you can output them as list or map

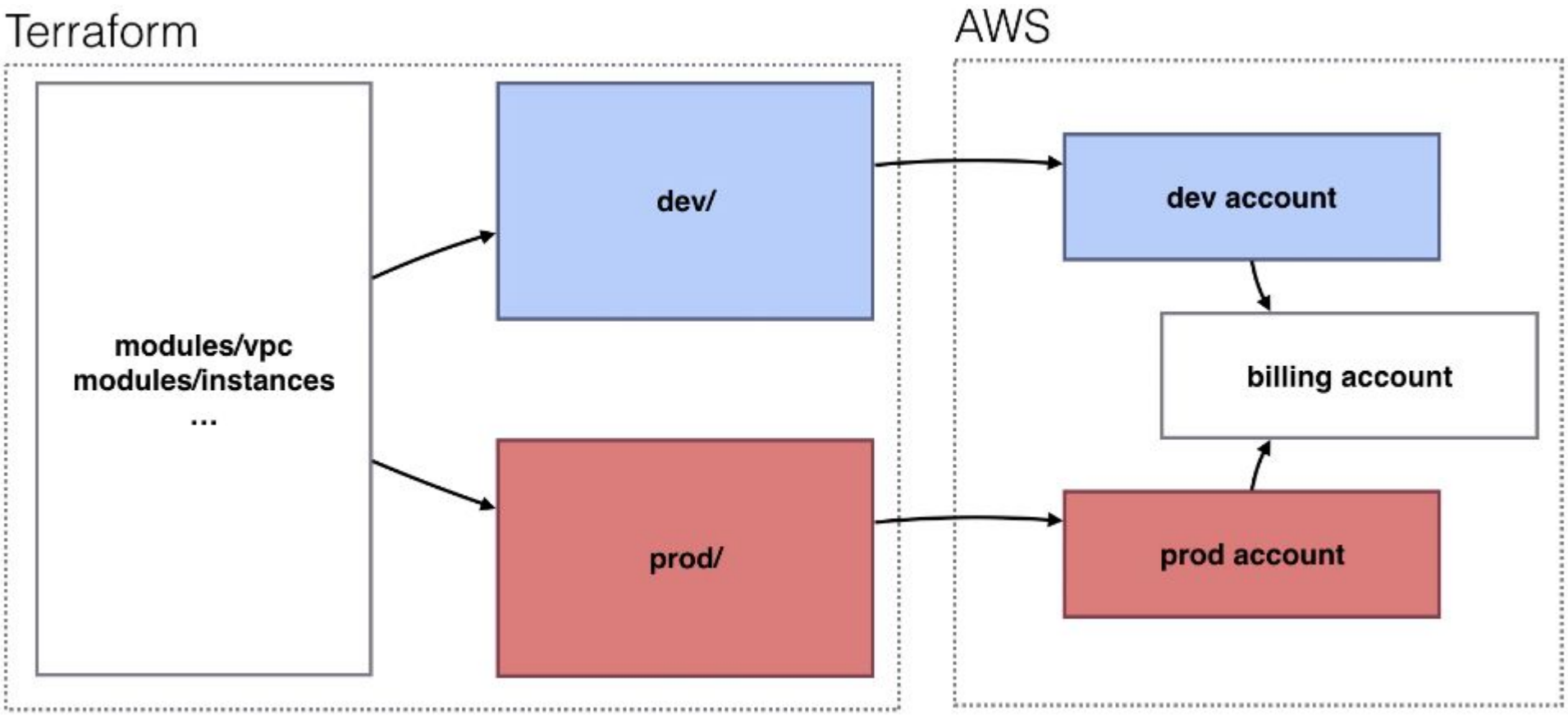
Terraform With Loops

- **For loops** are typically used when **assigning a value to an argument**
- For example:
  - security_groups = [“sg-12345”, sg-567”]
  - This could be replaced by for loop if you need to transform the input data
  - Tags = { Name = “resource name” }
    - This map can be “hardcoded” or which can be output of a for loop
- For_each loops are not used when assigning a value to an argument but rather to **repeat nested blocks**

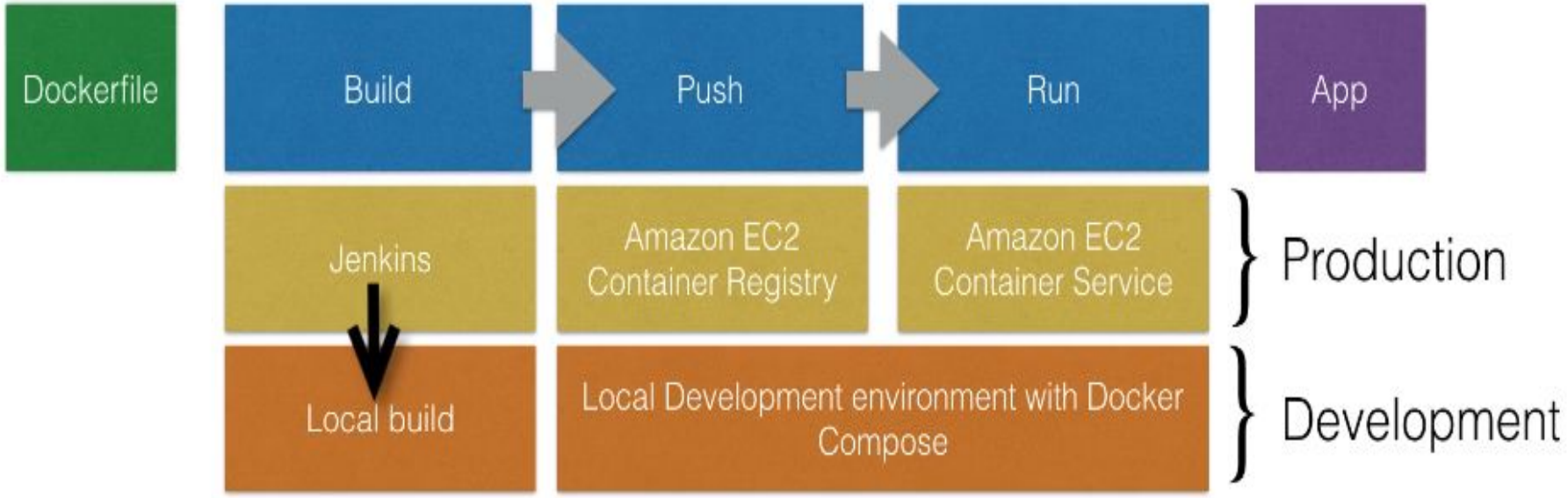
```
resource "aws_security_group" "example" {
 name = "example"

 ingress {                # this is the nested block, you can not use for loops here, as there is no “value”, it’s always a “literal”
  …
 }

 ingress {                # second nested block
  from_port   = 443
  to_port     = 443
```

Terraform With AWS

Project Structure



Docker on AWS



ECS

- Running dockerized app on ECS cluster
- Need to Dockerize app
- Now that we have dockerized app and uploaded on ECR we can start ECS cluster
- ECS – EC2 Container services will manage your docker containers
- You just need to start an autoscaling group with custom AMI
  - The custom AMI contains the ECS agent
- Once ECS cluster is online, tasks and services can be started on the cluster
- ECS cluster is group of EC2 instances with ecs agent on. ECS service will manage them.
- Ref : https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html

ECS Task Defination

- Before Docker app can be launched a tasks definition needs to be provided
- Task definition describes what docker container to be run on cluster:
  - Specifies docker image (the docker image in ECR)
  - Max CPU usage, Max memory usage
  - Whether containers to be linked (e.g. link app container with db container)
  - Environment Variables (e.g. credentials)
  - And any other container specific definitions
-

ECS Service Definition

- A service definition going to run specific amount of containers based on the task definition
- A service is always running if the container stops, it will be restarted
- A service can be scaled you can run 1 instance of container or multiple
- You can put an ELB in front of service
- You typically run multiple containers spread over Availability zones
  - If one containers fails, your load balancer stops sending traffic to it
  - Running multiple instances with ELB/ALB allows you have HA

ECS Task Definition

- A service definition going to run specific amount of containers based on the task definition
- A service is always running if the container stops, it will be restarted
- A service can be scaled you can run 1 instance of container or multiple
- You can put an ELB in front of service
- You typically run multiple containers spread over Availability zones
  - If one containers fails, your load balancer stops sending traffic to it
  - Running multiple instances with ELB/ALB allows you have HA

Terraform Modules

- Terraform modules are **powerful way to reuse code.**
- You can either use external module **or write module yourself**
- External module can help you **setting up infra without much effort**
  - When modules are managed by community you will get updates and fixes
- Writing modules gives you full flexibility
- It you maintain module in **git repo** you can even reuse the module over multiple projects

AWS EKS

- Amazon **Elastic Container Service for Kubernetes** is a highly available, scalable and secure kubernetes service
- It is general available in Jun 2018
- Kubernetes is an alternative to ECS
  - ECS is **AWS specific,** where as kubernetes can run on any public cloud providers or on-prem as well
  - Both are container orchestration solution

AWS EKS

- AWS EKS provides **managed Kubernetes master nodes**
  - There is no master node to manage
  - The master node are **multi A-Z** to provide redundancy
  - The master node will scale automatically when necessary
    - In case of own kubernetes cluster we need to manage master node if the worker nodes increased
- Secure By Default: **EKS integrates with IAM**