Infrastructure as Code w/Terraform

Tejas Parikh (t.parikh@northeastern.edu)

CSYE 6225

Northeastern University

Install Terraform

- Terraform is distributed as a single binary.
- Install Terraform by unzipping it and moving it to a directory included in your system's PATH .

Setup Environment for AWS

- For Terraform to be able to make changes in your AWS account, you will need to set the AWS credentials for the IAM user as the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.

```
$ export AWS_ACCESS_KEY_ID=(your access key id)
$ export AWS_SECRET_ACCESS_KEY=(your secret access key)
```

- In addition to environment variables, Terraform supports the same authentication mechanisms as all AWS CLI and SDK tools. Therefore, it'll also be able to use credentials in $HOME/.aws/credentials.

Writing Terraform Files

- Terraform code is written in the HashiCorp Configuration Language (HCL) in files with the extension *.tf*.
- You can write Terraform code in just about any text editor.

Provider

- The first step to using Terraform is to configure the provider(s) you want to use.

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

Initialize Working Directory

- The terraform binary contains the basic functionality for Terraform, but it does not come with the code for any of the providers (e.g., the AWS provider, Azure provider, GCP provider, etc.), so when you're first starting to use Terraform, you need to run terraform init to tell Terraform to scan the code, figure out which providers you're using, and download the code for them.
- By default, the provider code will be downloaded into a .terraform folder, which is Terraform's scratch directory (you may want to add it to .gitignore).
- You need to run init any time you start with new Terraform code.
- It is safe to run init multiple times (the command is idempotent).

Create an Execution Plan

- The terraform plan command is used to create an execution plan.
- Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.
- This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state.
- For example, terraform plan might be run before committing a change to version control, to create confidence that it will behave as expected.

Apply Changes

- The terraform apply command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a terraform plan execution plan.

Terraform State

- Terraform must store state about your managed infrastructure and configuration.
- This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.
- This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.
- Terraform uses this local state to create plans and make changes to your infrastructure.
- Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.
- Every time you run Terraform, it can fetch the latest status of resource from AWS and compare that to what's in your Terraform configurations to determine what changes need to be applied.
- The output of the plan command is a diff between the code on your computer and the infrastructure deployed in the real world, as discovered via IDs in the state file.

Destroy Terraform Managed Infrastructure

- The terraform destroy command is used to destroy the Terraform-managed infrastructure.

Configuring .gitignore

.terraform

*.tfstate

*.tfstate.backup

Git will ignore the *.terraform* folder, which Terraform uses as a temporary scratch directory, as well as *.tfstate files, which Terraform uses to store state.

Resources & Modules

- 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.
- A group of resources can be gathered into a *module*, which creates a larger unit of configuration.
- A resource describes a single infrastructure object, while a module might describe a set of objects and the necessary relationships between them in order to create a higher-level system.
- A Terraform configuration consists of a root module, where evaluation begins, along with a tree of child modules created when one module calls another.

Resource

- The general syntax for creating a resource in Terraform is

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

Referencing Resources

Syntax: <PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>

Variables

Declaring an Input Variable

Each input variable accepted by a module must be declared using a `variable` block:

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

The label after the `variable` keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

Using Input Variable Values

Within the module that declared a variable, its value can be accessed from within expressions as `var.<NAME>`, where `<NAME>` matches the label given in the declaration block:

```
resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami           = var.image_id
}
```

The value assigned to a variable can be accessed only from expressions within the module where it was declared.

Variable – Type Constraints

- The type argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable.
- If no type constraint is set then a value of any type is accepted.
- While type constraints are optional, it is recommend specifying them; they serve as easy reminders for users of the module, and allow Terraform to return a helpful error message if the wrong type is used.

Additional Resources

See Lecture Page