02. Deploying Infrastructure with Terraform

Contents

- Creating first EC2 instance with Terraform
- Terraform Code - First EC2 Instance
- Understanding Resources & Providers
- Destroying Infrastructure with Terraform
- Understanding Terraform State files
- Understanding Desired & Current States
- Challenges with the current state on computed values
- Terraform Provider Versioning
- Deploying Infrastructure with Terraform

Terraform Workflow Overview

- Core workflow
  - **Write** - Author infrastructure as code.
    - Create templates in HCL (HashiCorp Configuration Language) (xxx.tf files)
    - Templates incorporated into version control systems (VCS)
  - **Initialize** Terraform in directory where terraform templates live
    - Installs providers and modules
  - **Validate** configuration for syntax
  - **Plan** - Preview changes before applying.
    - Validate the templates (terraform validate) and other tools (e.g. tf-lint)
    - terraform plan -> verify what changes will be performed
  - **Apply** - Provision reproducible infrastructure.
    - terraform apply
    - Verify state changes
  - **Destroy**
- This course will constantly revolve around these steps and the underlying concept of "state"

Terraform Workflow and CLI commands

- Main steps when working with Terraform
  - Create Terraform Configuration Files (typically .tf) that represent (declarative) the cloud resources we want Terraform to deploy
  - *terraform fmt* – format config files (indent, brackets, etc...)
  - ***terraform init*** - Initialize Terraform in the directory containing the .tf files
  - *terraform validate* – validate that config files are syntactically correct
  - ***terraform plan*** - Visualize what changes will terraform apply in the cloud provider
  - ***terraform apply*** – Actually implement those changes
- In addition, there are a number of additional *terraform* subcommands (console, state, etc.)

Working with Configuration Files (a.k.a. Templates)

Working with Configuration Files

- Key Concept: Resources – explored in detail in next chapters
- Main goal of HCL (HashiCorp Configuration Language): declaring **resources** representing real world infrastructure objects
  - Example: VMs, Subnets, or Security Groups deployed in a specific cloud provider
- Each resource block describes one or more infrastructure objects, such as virtual machines, virtual networks, or higher-level components such as DNS records.
- Syntax Overview
  - Terraform Language docs

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
  required_version = ">= 1.3.0"
}

provider "aws" {
  profile = "tf"
  region  = "eu-west-1"
}

variable "my_ami" {
  description = "ami for EC2 instance"
  type = string
  default = "ami-0b752bf1df193a6c4"
}

resource "aws_instance" "server1" {
  ami = var.my_ami
  instance_type = "t2.micro"

  tags = {
    Name = "vm1"
    project = "acme"
  }
}
```

Terraform Code
First EC2 Instance

- Demo
  - Overview of simple template
  - Apply terraform workflow
    - terraform init -> installs provider(s), etc…
    - terraform fmt -> formats code
    - terraform validate -> syntax validation
    - terraform plan -> show what will be done
    - terraform apply -> actually configure
    - terraform state commands

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
  required_version = ">= 1.3.0"
}

provider "aws" {
  profile = "tf"
  region  = "eu-west-1"
}

variable "my_ami" {
  description = "ami for EC2 instance"
  type = string
  default = "ami-0b752bf1df193a6c4"
}

resource "aws_instance" "server1" {
  ami = var.my_ami
  instance_type = "t2.micro"

  tags = {
    Name = "vm1"
    project = "acme"
  }
}
```

Working with Configuration Files (a.k.a. Templates)

Working with Configuration Files

- Key Concept: Resources – explored in detail in next chapters
- Main goal of HCL (HashiCorp Configuration Language): declaring **resources** representing real world infrastructure objects
  - Example: VMs, Subnets, or Network Security Groups deployed in a specific cloud provider)
- Each resource block describes one or more infrastructure objects, such as virtual machines, virtual networks, or higher-level components such as DNS records.
- Syntax Overview
  - Terraform Language [docs]

Using an IDE and Version Control

- Examples in this course use VS Code
- Install TF specific extensions : HashiCorp Terraform
  - Uses Terraform Language Server (terraform-ls) that provides Terraform specifics to many IDEs/Editors
- Includes built-in version control and integration with github
  - Terraform Specific .gitignore file from GitHub

Formatting terraform files: terraform fmt

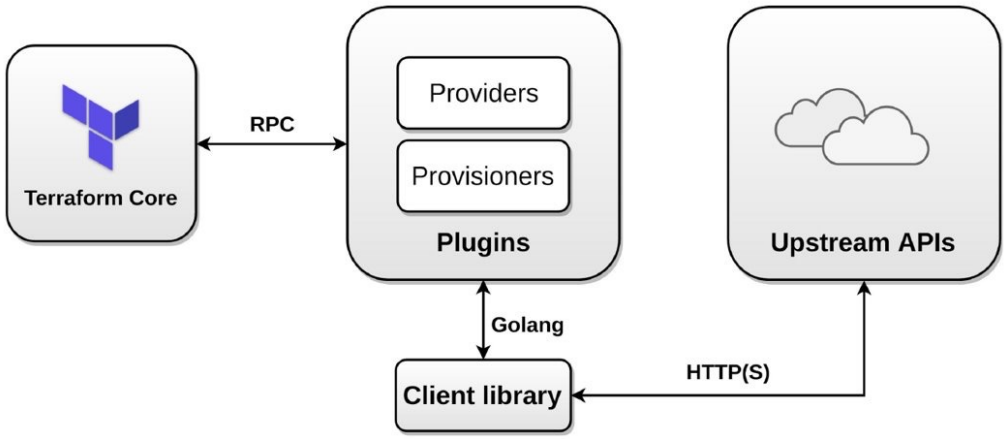
```
$ terraform fmt -help
Usage: terraform [global options] fmt [options] [DIR]

       Rewrites all Terraform configuration files to a canonical format. Both
       configuration files (.tf) and variables files (.tfvars) are updated.
       JSON files (.tf.json or .tfvars.json) are not modified.

       If DIR is not specified then the current working directory will be used.
       If DIR is "-" then content will be read from STDIN. The given content must
       be in the Terraform language native syntax; JSON is not supported.

Options:

  -list=false    Don't list files whose formatting differs
                 (always disabled if using STDIN)

  -write=false   Don't write to source files
                 (always disabled if using STDIN or -check)

  -diff          Display diffs of formatting changes

  -check         Check if the input is formatted. Exit status will be 0 if all
                 input is properly formatted and non-zero otherwise.

  -no-color      If specified, output won't contain any color.

  -recursive     Also process files in subdirectories. By default, only the
                 given directory (or current directory) is processed.
```

```
$ terraform fmt -diff
web.server.tf
--- old/web.server.tf
+++ new/web.server.tf
@@ -24,9 +24,9 @@

 resource "aws_instance" "test_userdata" {
   ami                         = data.aws_ami.amazon_linux.id
-  instance_type = var.ec2_instance_type
+  instance_type               = var.ec2_instance_type
   vpc_security_group_ids      = [aws_security_group.sec_web_test.id]
-  associate_public_ip_address                    = true
+  associate_public_ip_address = true
   subnet_id                   = element(tolist(data.aws_subnet_ids.sub_ids.ids), 0)

   user_data = local.user_data_temp
```

Initializing Working Directories *terraform init*

Terraform Providers (AWS) – Quick Review

- Terraform = Terraform Core + Providers
- Each provider is a plugin extension to terraform core for a specific API (e.g. for a specific cloud provider)
- AWS Provider Documentation
- General doc on terraform providers
- This course: focus on ”aws” provider



terraform init

- *terraform init* documentation
- The terraform init command is used to initialize a working directory containing Terraform configuration files.
- This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control.
- It is safe to run this command multiple times.
- After running *terraform init*, terraform scans the .tf files for provider information. It then downloads the required providers (say, *azurerm*) to specific subdirectories under directory *.terraform*
  - This behavior can be partially modified by specifying caching for plugins as seen in a previous module
  - **Note**: in addition to providers, terraform init also downloads modules referred to in the configuration file(s) (modules discussed in subsequent chapters)

Terraform Validate

terraform validate

- Docs

- Validate runs checks that verify whether a configuration is syntactically valid and internally consistent
- Useful for general verification of reusable modules
- Related also to variable validation (see example in modules section)
- It is safe to run this command automatically, for example as a post-save check in a text editor or as a test step for a re-usable module in a CI system.

Terraform Plan

terraform plan

- Docs
- The terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure. By default, when Terraform creates a plan it:
  - Reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
  - Compares the current configuration to the prior state and noting any differences.
  - Proposes a set of change actions that should, if applied, make the remote objects match the configuration.
- In addition to "normal" mode, there are two alternative **planning modes**
  - Destroy : creates a plan whose goal is to destroy all remote objects that currently exist
  - Refresh-only : for example to start working on reconciliation with changes made outside of terraform
- Planning **options**
  - -refresh=false
    - Use carefully - See HashiCorp Announcement for specifics
  - -replace=<someresource> (related to "taint" now deprecated – see section on managing state)
  - -target=<someresource>
  - -var and –var-file - set some variables => demo changing –var='project=acme03' and see what happens...
  - -destroy : speculative destroy plan - "see what would happen"

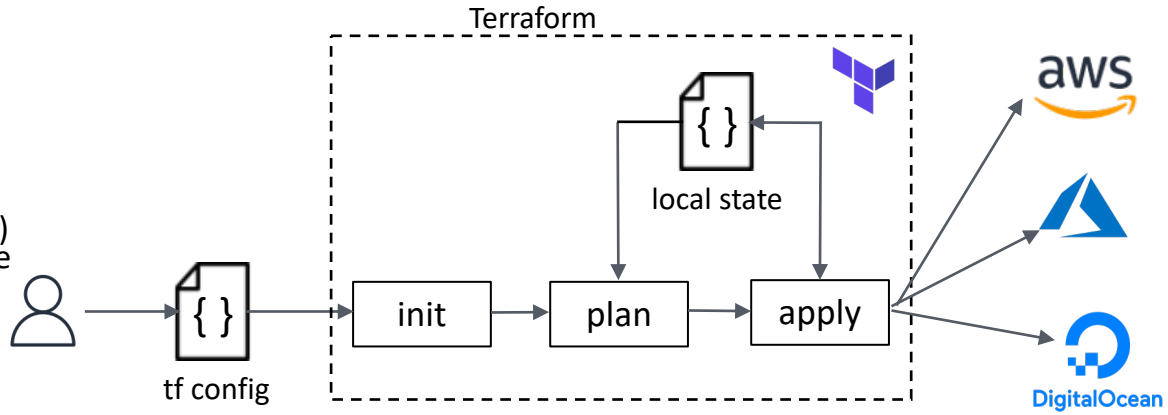
Terraform Apply

terraform apply

- Docs
- In a way, this is the most important command, since it executes the actions proposed in a Terraform plan.
  - See options in docs
- Do not skip plan...
- Do not use –auto-approve (do as I say, not as I do … :-))
- Specific documentation section on using Terraform in automation

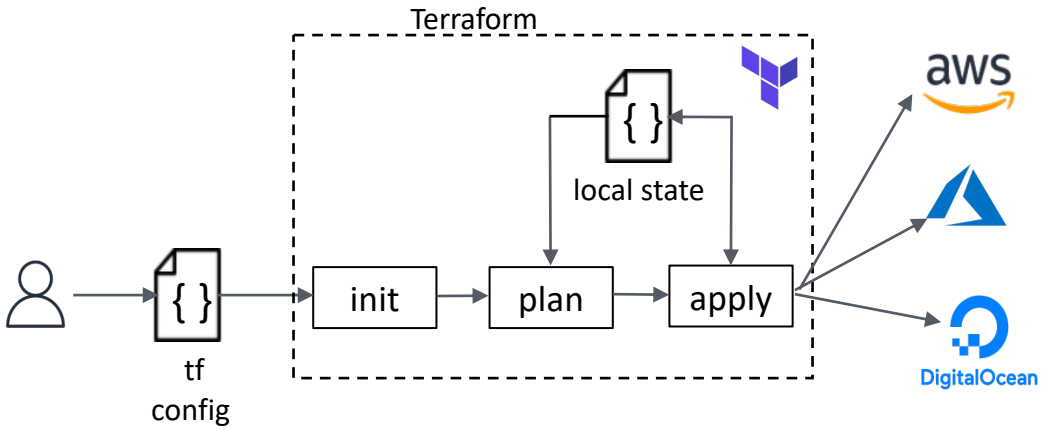
Terraform State – An introduction

- The concept of "state" is key to understanding and working with Terraform
- Full course chapter devoted to state.
- A state file is created or modified every time we run terraform apply
- The state file captures "Terraform's view" of the resources it has created and is managing
- Terraform must store state about your managed infrastructure and configuration.
  - This state is used by Terraform to map real world resources (real world state) to your configuration (desired state), 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 the 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.



Terraform State (2)

- The primary purpose of Terraform state is to **store bindings** between objects in a remote system (e.g. actual VMs, Vnets, subnets, resource groups) and resources declared in your configuration.
  - When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.
  - Terraform expects to "own" and control the remote object. Any changes to the VM, VPC, etc.. Should be performed through Terraform. Otherwise problems arise (see discusion of drift in next slides)
- Maintaining state is not trivial. Are there alternatives? : Terraform's own explanation



Terraform Destroy

terraform destroy

- Docs
- 'terraform destroy' is really an alias for the official command 'terraform apply -destroy'
- Note that for automation and pipelines, HashiCorp recommends using 'terraform apply –destroy’
- Recall also 'terraform plan –destroy' - Runs terraform plan "knowing" that the purpose of the exercise is to destroy the infrastructure.

Other Terraform Commands

terraform output

- Display output values
  - Reading output values from state file is discouraged (considered internal API)
- By default individual outputs are quoted (since around TF 0.13)
- For post processing, use json output and filters (e.g. with jq). Simple example below.
- Convenient way to SSH to instances example in gist

```
rafa@rp3:simple.ec2.flask.ddb$ terraform output -json | jq
{
  "ami": {
    "sensitive": false,
    "type": "string",
    "value": "ami-0a8dc52684ee2fee2"
  },
  "key_name": {
    "sensitive": false,
    "type": "string",
    "value": "net03"
  },
  "public_ip": {
    "sensitive": false,
    "type": "string",
    "value": "34.248.171.175"
  },
```

```
rafa@rp3:simple.ec2.flask.ddb$ terraform output public_ip
"34.248.171.175"
rafa@rp3:simple.ec2.flask.ddb$ terraform output -json  | jq -r .public_ip.value
34.248.171.175
rafa@rp3:simple.ec2.flask.ddb$ ping $(terraform output -json  | jq -r .public_ip.value)
PING 34.248.171.175 (34.248.171.175) 56(84) bytes of data.
64 bytes from 34.248.171.175: icmp_seq=1 ttl=234 time=39.4 ms
64 bytes from 34.248.171.175: icmp_seq=2 ttl=234 time=40.1 ms
```

terraform console

- Docs
- Very useful command to explore and troubleshoot
- This command provides an interactive command-line console for evaluating and experimenting with expressions. This is useful for testing interpolations before using them in configurations, and for interacting with any values currently saved in state.

- Note: in some environments (e.g. AWS) running *terraform console* locks the remote state file, preventing other users from updating state
  - One way to minimize the impact of this behavior is to use console as part of a bash pipe:
  - echo 'element([1,2,3],2)' | terraform console
  - 3

Dependencies (AWS)

- Terraform Documentation: depends_on and dependencies tutorial
- Terraform uses dependency information to determine the correct order in which to create the different resources. To do so, it creates a dependency graph of all of the resources defined by the configuration.
- The *depends_on* meta-argument is used in cases where terraform cannot automatically infer dependencies between resources and modules
  - See example involving iam_role in *depends_on* documentation
- Important warning from Terraform:
  - The depends_on argument should be **used only as a last resort**.
  - When using it, **always include a comment explaining why** it is being used, to help future maintainers understand the purpose of the additional dependency.
- Note: depends_on support in modules was added in version 0.13. Previous versions of terraform supported depends_on only for resources

```
resource "aws_iam_role" "example" {
  name = "example"

  # assume_role_policy is omitted for brevity in this example. See the
  # documentation for aws_iam_role for a complete example.
  assume_role_policy = "..."
}

resource "aws_iam_instance_profile" "example" {
  # Because this expression refers to the role, Terraform can infer
  # automatically that the role must be created first.
  role = aws_iam_role.example.name
}

resource "aws_iam_role_policy" "example" {
  name   = "example"
  role   = aws_iam_role.example.name
  policy = jsonencode({
    "Statement" = [{
      # This policy allows software running on the EC2 instance to
      # access the S3 API.
      "Action" = "s3:*",
      "Effect" = "Allow",
    }],
  })
}

resource "aws_instance" "example" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  # Terraform can infer from this that the instance profile must
  # be created before the EC2 instance.
  iam_instance_profile = aws_iam_instance_profile.example

  # However, if software running in this EC2 instance needs access
  # to the S3 API in order to boot properly, there is also a "hidden"
  # dependency on the aws_iam_role_policy that Terraform cannot
  # automatically infer, so it must be declared explicitly:
  depends_on = [
    aws_iam_role_policy.example,
  ]
}
```

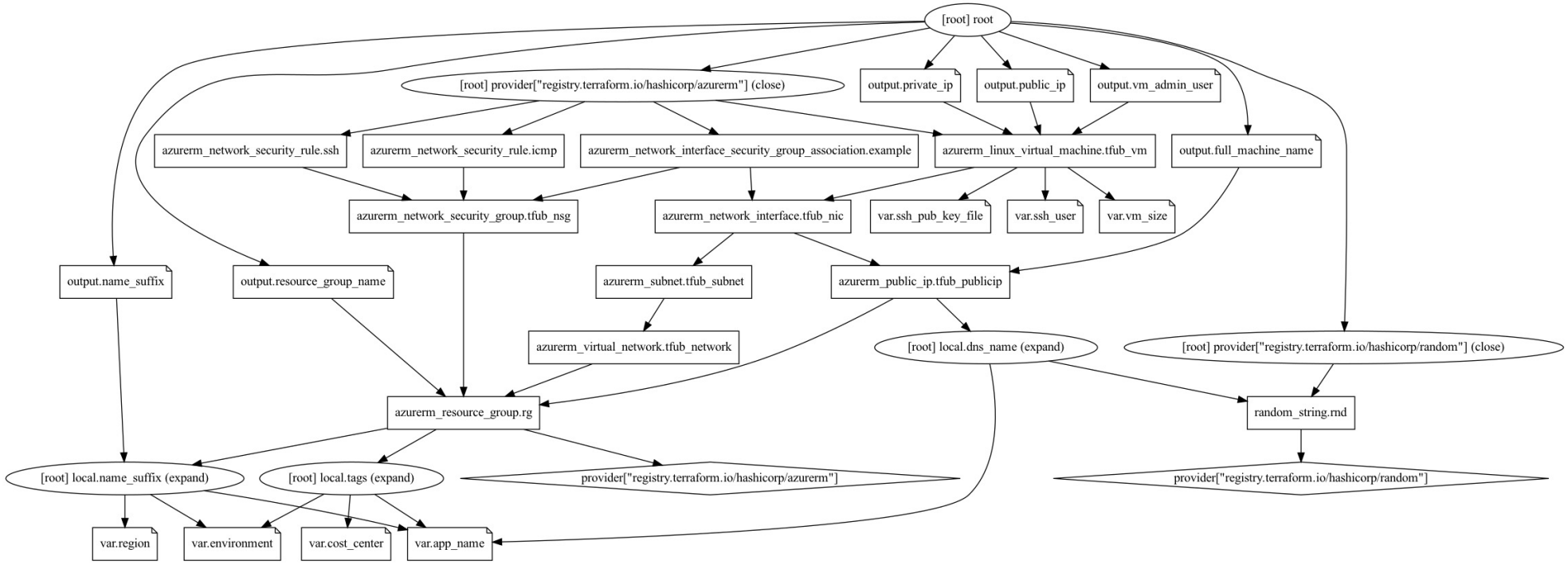
terraform graph command

- Used to generate a visual representation of either a configuration or execution plan.
- The output is in the DOT format, which can be used by GraphViz to generate charts.
- Terraform graph internals

```
rafa@rp3:three-tier$terraform graph -help
Usage: terraform [global options] graph [options]

  Outputs the visual execution graph of Terraform resources according to
  either the current configuration or an execution plan.

  The graph is outputted in DOT format. The typical program that can
  read this format is GraphViz, but many web services are also available
  to read this format.

  The -type flag can be used to control the type of graph shown. Terraform
  creates different graphs for different operations. See the options below
  for the list of types supported. The default type is "plan" if a
  configuration is given, and "apply" if a plan file is passed as an
  argument.

Options:

  -plan=tfplan     Render graph using the specified plan file instead of the
                   configuration in the current directory.

  -draw-cycles     Highlight any cycles in the graph with colored edges.
                   This helps when diagnosing cycle errors.

  -type=plan       Type of graph to output. Can be: plan, plan-destroy, apply
                   validate, input, refresh.

  -module-depth=n  (deprecated) In prior versions of Terraform, specified the
                                depth of modules to show in the output.
```

Viewing terraform graph output - Graphviz

- Install GraphViz (for “dot” utility)
- Generate PNG file with command:
  - terraform graph | dot -Tpng > graph.png
- Sample output for a single Azure Linux VM



Terraform Graphs - Other tools

- Some tools attempt to improve / manage the terraform generated graphs
  - Graphviz-online
  - Terraform-graph-beautifier
    - Generates web page – allows excluding some elements (e.g. --exclude "var.*")
    - Allows moving and placing elements once rendered
- Consider using 3rd party tools that use Cloud Provider API to interrogate cloud and generate graphic documentation (e.g. Lucidscale ($$$))