TERRAFORM

What is IAC?

- Infrastructure as Code (IaC) is the management of infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model, using the same versioning as DevOps team uses for source code.
- The same source code generates the same binary, an IaC model generates the same environment every time it is applied.
- IaC is a key DevOps practice and is used in conjunction with continuous delivery.

IAC tools

- Terraform
- AWS CloudFormation
- Azure Resource Manager and Google Cloud Deployment Manager
- Chef
- Puppet
- Saltstack
- Ansible
- Juju
- Docker
- Vagrant
- Pallet
- (R)?ex
- CFEngine
- NixOS

Terraform

- Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
- Configuration files describe to Terraform the components needed to run a single application or your entire datacenter.
- Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure.
- As the configuration changes, Terraform can determine what changed and create incremental execution plans which can be applied.

Execution Plans

- Terraform has a "planning" step where it generates an execution plan.
- The execution plan shows what Terraform will do when you call apply.
- This lets you avoid any surprises when Terraform manipulates infrastructure.

Terraform - Advantages

- Multi Provider
- Syntax
- Additional Resources
- State
- Dry Runs
- Data Sources
- Data Sources

Terraform - Disadvantages

- Resource Availability and Dependency
- Parameter Files
- Key Vault Access
- Error Handling

Plugins

- Terraform is built on a plugin-based architecture. All providers and provisioners that are used in Terraform configurations are plugins, even the core types such as AWS and Heroku. Users of Terraform can write new plugins in order to support new functionality in 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.

State - Inspection and Modification

- While the format of the state files are just JSON, direct file editing of the state is discouraged. Terraform provides the terraform state command to perform basic modifications of the state using the CLI.
- The CLI usage and output of the state commands is structured to be friendly for Unix tools such as grep, awk, etc. Additionally, the CLI insulates users from any format changes within the state itself. The Terraform project will keep the CLI working while the state format underneath it may shift.
- Finally, the CLI manages backups for you automatically. If you make a mistake modifying your state, the state CLI will always have a backup available for you that you can restore.

State - Format

- The state is in JSON format and Terraform will promise backwards compatibility with the state file.
- The JSON format makes it easy to write tools around the state if you want or to modify it by hand in the case of a Terraform bug.
- The "version" field on the state contents allows us to transparently move the format forward if we make modifications.

Providers

- Terraform is used to create, manage, and update infrastructure resources such as physical machines, VMs, network switches, containers, and more. Almost any infrastructure type can be represented as a resource in Terraform.
- A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g. AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Enterprise, DNSimple, CloudFlare).
- https://www.terraform.io/docs/providers/index.html

Provisioners

- Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc.
- Provisioners are added directly to any resource:

```
resource "aws_instance" "web" {
  # ...
  provisioner "local-exec" {
    command = "echo ${self.private_ip} > file.txt"
  }
}
```

- For provisioners other than local execution, you must specify connection settings so Terraform knows how to communicate with the resource.
- https://www.terraform.io/docs/provisioners/index.html

Installing Terraform

- Terraform is distributed as a binary package for all supported platforms and architectures.
- To install Terraform, find the appropriate package for your system and download it. Terraform is packaged as a zip archive.
- After downloading Terraform, unzip the package.
- Terraform runs as a single binary named "terraform".
- The final step is to make sure that the terraform binary is available on the PATH.
- Linux/MAC: https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux-unix
- Windows: https://stackoverflow.com/questions/1618280/where-can-i-set-path-to-make-exe-on-windows

Installing terraform in RHEL

- wget https://releases.hashicorp.com/terraform/0.11.8/terraform_0.11.8_linux_amd64.zip
- https://releases.hashicorp.com/terraform/0.12.16/terraform_0.12.16_linux_amd64.zip
- unzip terraform_0.11.8_linux_amd64.zip
- sudo mv terraform /usr/local/bin
- terraform --version

Terraform configuration

- Terraform uses text files to describe infrastructure and to set variables. These text files are called Terraform configurations and end in .tf.
- The format of the configuration files can be in two formats: Terraform format and JSON.
- The Terraform format is more human-readable, supports comments, and is the generally recommended format for most Terraform files.
- The JSON format is meant for machines to create, modify, and update.
- Terraform format ends in .tf and JSON format ends in .tf.json.

Terraform Commands (CLI)

- Terraform is controlled via a very easy to use command-line interface (CLI). Terraform is only a single command-line application: terraform. This application then takes a subcommand such as "apply" or "plan".
- To view a list of the available commands at any time, just run terraform with no arguments:

```
$terraform
```

- To get help for any specific command, pass the -h flag to the relevant subcommand. For example, to see help about the graph subcommand:

```
$ terraform graph -h
```

- Commonly used commands are: init, plan, apply, destroy

Command: init

- 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.

Command: init - Usage

- terraform init [options] [DIR]
- General Options
- The following options apply to all (or several of) the initialization steps:
  - ✓ -input=true Ask for input if necessary. If false, will error if input was required.
  - ✓ -lock=false Disable locking of state files during state-related operations.
  - ✓ -lock-timeout=<duration> Override the time Terraform will wait to acquire a state lock. The default is 0s (zero seconds), which causes immediate failure if the lock is already held by another process.
  - ✓ -no-color Disable color codes in the command output.
  - ✓ -upgrade Opt to upgrade modules and plugins as part of their respective installation steps.

Command: init - Backend Initialization

- During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings.
- Re-running init with an already-initalized backend will update the working directory to use the new backend settings. Depending on what changed, this may result in interactive prompts to confirm migration of workspace states. The -force-copy option suppresses these prompts and answers "yes" to the migration questions. The -reconfigure option disregards any existing configuration, preventing migration of any existing state.
- To skip backend configuration, use -backend=false. Note that some other init steps require an initialized backend, so it is recommended to use this flag only when the working directory was already previously initialized for a backend.
- The -backend-config=... option can be used for partial backend configuration, in situations where the backend settings are dynamic or sensitive and so cannot be statically specified in the configuration file.

Command: init - Plugin Installation

- During init, Terraform searches the configuration for both direct and indirect references to providers and attempts to load the required plugins.
- For providers distributed by HashiCorp, init will automatically download and install plugins if necessary. Plugins can also be manually installed in the user plugins directory, located at ~/.terraform.d/plugins on most operating systems and %APPDATA%\terraform.d\plugins on Windows.
- On subsequent runs, init only installs providers without acceptable versions installed. (This includes newly added providers, and providers whose installed versions can't meet the current version constraints.) Use -upgrade if you want to update all providers to the newest acceptable version.
- We can modify terraform init's plugin behavior with the following options:
  - ✓ -upgrade — Update all previously installed plugins to the newest version that complies with the configuration's version constraints. This option does not apply to manually installed plugins.
  - ✓ -get-plugins=false — Skips plugin installation. Terraform will use plugins installed in the user plugins directory, and any plugins already installed for the current working directory. If the installed plugins aren't sufficient for the configuration, init fails.
  - ✓ -plugin-dir=PATH — Skips plugin installation and loads plugins only from the specified directory. This ignores the user plugins directory and any plugins already installed in the current working directory. To restore the default behavior after using this option, run init again and pass an empty string to -plugin-dir.
  - ✓ -verify-plugins=false — Skips release signature validation when installing downloaded plugins (not recommended). Official plugin releases are digitally signed by HashiCorp, and Terraform verifies these signatures when automatically downloading plugins. This option disables that verification. (Terraform does not check signatures for manually installed plugins.)

Command: init - Running terraform init in automation

- For teams that use Terraform as a key part of a change management and deployment pipeline, it can be desirable to orchestrate Terraform runs in some sort of automation in order to ensure consistency between runs and provide other interesting features such as integration with version control hooks.
- There are some special concerns when running init in such an environment, including optionally making plugins available locally to avoid repeated re-installation.

Command: 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.
- The optional -out argument can be used to save the generated plan to a file for later execution with terraform apply, which can be useful when running Terraform in automation.

Command: plan -Usage

- terraform plan [options] [dir-or-plan]
- By default, plan requires no flags and looks in the current directory for the configuration and state file to refresh.
- If the command is given an existing saved plan as an argument, the command will output the contents of the saved plan. In this scenario, the plan command will not modify the given plan. This can be used to inspect a planfile.
- The command-line flags are all optional. The list of available flags are:

✓ -destroy - If set, generates a plan to destroy all the known resources.

✓ -detailed-exitcode - Return a detailed exit code when the command exits. When provided, this argument changes the exit codes and their meanings to provide more granular information about what the resulting plan contains:

✓ 0 = Succeeded with empty diff (no changes)

✓ 1 = Error

✓ 2 = Succeeded with non-empty diff (changes present)

- ✓ -input=true - Ask for input for variables if not directly set.
- ✓ -lock=true - Lock the state file when locking is supported.
- ✓ -lock-timeout=0s - Duration to retry a state lock.
- ✓ -module-depth=n - Speifies the depth of modules to show in the output. This does not affect the plan itself, only the output shown. By default, this is -1, which will expand all.
- ✓ -no-color - Disables output with coloring.
- ✓ -out=path - The path to save the generated execution plan. This plan can then be used with terraform apply to be certain that only the changes shown in this plan are applied. Read the warning on saved plans below.
- ✓ -parallelism=n - Limit the number of concurrent operation as Terraform walks the graph.
- ✓ -refresh=true - Update the state prior to checking for differences.
- ✓ -state=path - Path to the state file. Defaults to "terraform.tfstate". Ignored when remote state is used.
- ✓ -target=resource - A Resource Address to target. This flag can be used multiple times. See below for more information.
- ✓ -var 'foo=bar' - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as HCL, so list and map values can be specified via this flag.
- ✓ -var-file=foo - Set variables in the Terraform configuration from a variable file. If a terraform.tfvars or any .auto.tfvars files are present in the current directory, they will be automatically loaded. terraform.tfvars is loaded first and the .auto.tfvars files after in alphabetical order. Any files specified by -var-file override any values set automatically from files in the working directory. This flag can be used multiple times.

Command: plan - Security Warning

- Saved plan files (with the -out flag) encode the configuration, state, diff, and variables. Variables are often used to store secrets. Therefore, the plan file can potentially store secrets.
- Terraform itself does not encrypt the plan file. It is highly recommended to encrypt the plan file if you intend to transfer it or keep it at rest for an extended period.

Command: apply

- 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.

Command: apply - Usage

- terraform apply [options] [dir-or-plan]
- By default, apply scans the current directory for the configuration and applies the changes appropriately. However, a path to another configuration or an execution plan can be provided. Explicit execution plans files can be used to split plan and apply into separate steps within automation systems.
- The command-line flags are all optional. The list of available flags are:
  - ✓-backup=path - Path to the backup file. Defaults to -state-out with the ".backup" extension. Disabled by setting to "-".
  - ✓-lock=true - Lock the state file when locking is supported.
  - ✓-lock-timeout=0s - Duration to retry a state lock.
  - ✓-input=true - Ask for input for variables if not directly set.
  - ✓-auto-approve - Skip interactive approval of plan before applying.
  - ✓-no-color - Disables output with coloring.

Command: apply - Usage

- ✓ -parallelism=n - Limit the number of concurrent operation as Terraform walks the graph.
- ✓ -refresh=true - Update the state for each resource prior to planning and applying. This has no effect if a plan file is given directly to apply.
- ✓ -state=path - Path to the state file. Defaults to "terraform.tfstate". Ignored when remote state is used.
- ✓ -state-out=path - Path to write updated state file. By default, the -state path will be used. Ignored when remote state is used.
- ✓ -target=resource - A Resource Address to target. For more information, see the targeting docs from terraform plan.
- ✓ -var 'foo=bar' - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as HCL, so list and map values can be specified via this flag.
- ✓ -var-file=foo - Set variables in the Terraform configuration from a variable file. If a terraform.tfvars or any .auto.tfvars files are present in the current directory, they will be automatically loaded. terraform.tfvars is loaded first and the .auto.tfvars files after in alphabetical order. Any files specified by -var-file override any values set automatically from files in the working directory. This flag can be used multiple times.

Command: destroy

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

» **Usage**

- Usage: terraform destroy [options] [dir]
- Infrastructure managed by Terraform will be destroyed. This will ask for confirmation before destroying.
- This command accepts all the arguments and flags that the apply command accepts, except for a plan file argument.
- If -auto-approve is set, then the destroy confirmation will not be shown.
- The -target flag, instead of affecting "dependencies" will instead also destroy any resources that depend on the target(s) specified.
- The behavior of any terraform destroy command can be previewed at any time with an equivalent terraform plan -destroy command.

Import

- Terraform can import existing infrastructure. This allows you take resources you've created by some other means and bring it under Terraform management.
- This is a great way to slowly transition infrastructure to Terraform, or to be able to be confident that you can use Terraform in the future if it potentially doesn't support every feature you need today.

» **Currently State Only**

- The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.
- Because of this, prior to running terraform import it is necessary to write manually a resource configuration block for the resource, to which the imported object will be attached.
- While this may seem tedious, it still gives Terraform users an avenue for importing existing resources. A future version of Terraform will fully generate configuration, significantly simplifying this process.

Import Usage

- The terraform import command is used to import existing infrastructure.
- The command currently can only import one resource at a time. This means you can't yet point Terraform import to an entire collection of resources such as an AWS VPC and import all of it. This workflow will be improved in a future version of Terraform.
- To import a resource, first write a resource block for it in your configuration, establishing the name by which it will be known to Terraform:

```
resource "aws_instance" "example" {
  # ...instance configuration...
}
```

- The name "example" here is local to the module where it is declared and is chosen by the configuration author. This is distinct from any ID issued by the remote system, which may change over time while the resource name remains constant.

Import Usage

- If desired, you can leave the body of the resource block blank for now and return to fill it in once the instance is imported.
- Now terraform import can be run to attach an existing instance to this resource configuration:

```
$ terraform import aws_instance.example i-abcd1234
```

- This command locates the AWS instance with ID i-abcd1234 and attaches its existing settings, as described by the EC2 API, to the name aws_instance.example in the Terraform state.
- It is also possible to import to resources in child modules and to single instances of a resource with count set. See Resource Addressing for more details on how to specify a target resource.
- The syntax of the given ID is dependent on the resource type being imported. For example, AWS instances use an opaque ID issued by the EC2 API, but AWS Route53 Zones use the domain name itself. Consult the documentation for each importable resource for details on what form of ID is required.
- As a result of the above command, the resource is recorded in the state file. You can now run terraform plan to see how the configuration compares to the imported resource, and make any adjustments to the configuration to align with the current (or desired) state of the imported object.

Creation-Time Provisioners

- Provisioners by default run when the resource they are defined within is created. Creation-time provisioners are only run during creation, not during updating or any other lifecycle. They are meant to perform bootstrapping of a system.
- If a creation-time provisioner fails, the resource is marked as tainted. A tainted resource will be planned for destruction and recreation upon the next terraform apply. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is ***tainting***.

Destroy-Time Provisioners

- If when = "destroy" is specified, the provisioner will run when the resource it is defined within is destroyed.
- Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will error and rerun the provisioners again on the next terraform apply. Due to this behavior, care should be taken for destroy provisioners to be safe to run multiple times.

Multiple Provisioners

- Multiple provisioners can be specified within a resource. Multiple provisioners are executed in the order they're defined in the configuration file.
- You may also mix and match creation and destruction provisioners. Only the provisioners that are valid for a given operation will be run. Those valid provisioners will be run in the order they're defined in the configuration file.
- Example of multiple provisioners:

```
resource "aws_instance" "web" {
  # ...
  provisioner "local-exec" {
    command = "echo first"
  }
  provisioner "local-exec" {
    command = "echo second"
  }
}
```

Failure Behavior

- By default, provisioners that fail will also cause the Terraform apply itself to error. The on_failure setting can be used to change this. The allowed values are:
- "continue" - Ignore the error and continue with creation or destruction.
- "fail" - Error (the default behavior). If this is a creation provisioner, taint the resource.
- Example:

```
resource "aws_instance" "web" {
  # ...
  provisioner "local-exec" {
    command    = "echo ${self.private_ip} > file.txt"
    on_failure = "continue"
  }
}
```

Modules

- Modules in Terraform are self-contained packages of Terraform configurations that are managed as a group. Modules are used to create reusable components in Terraform as well as for basic code organization.
- Modules are very easy to both use and create. Depending on what you're looking to do first, use the navigation on the left to dive into how modules work.

» Definitions

- Root module That is the current working directory when you run terraform apply or get, holding the Terraform configuration files. It is itself a valid module.

Backends

- A "backend" in Terraform determines how state is loaded and how an operation such as apply is executed. This abstraction enables non-local file state storage, remote execution, etc.
- By default, Terraform uses the "local" backend, which is the normal behavior of Terraform you're used to. This is the backend that was being invoked throughout the introduction.
- Here are some of the benefits of backends:
    - ✓ Working in a team: Backends can store their state remotely and protect that state with locks to prevent corruption. Some backends such as Terraform Enterprise even automatically store a history of all state revisions.
    - ✓ Keeping sensitive information off disk: State is retrieved from backends on demand and only stored in memory. If you're using a backend such as Amazon S3, the only location the state ever is persisted is in S3.
    - ✓ Remote operations: For larger infrastructures or certain changes, terraform apply can take a long, long time. Some backends support remote operations which enable the operation to execute remotely. You can then turn off your computer and your operation will still complete. Paired with remote state storage and locking above, this also helps in team environments.