Modern Infrastructure with Terraform
1. Introduction to Terraform
2. Infrastructure as Code
3. Variables, outputs, and meta-parameters
4. Command line interface
5. State
6. Modules
Introduction
Terraform Provisions Infrastructure Resources
Terraform's Goals

Unify the view of resources using infrastructure as code

Support the modern data center (IaaS, PaaS, SaaS)

Expose a way to safely and predictably change infrastructure

Provide a workflow that is technology agnostic
Terraform vs. Other Tools

Provides a high-level abstraction of infrastructure (IaC)

Allows for composition and combination

Supports parallel management of resources (graph, fast)

Separates planning from execution (dry-run)
Glossary
Provider

A provider is an abstraction of the API/service provider such as AWS, GCP, DNSimple, or Fastly. Providers typically require some sort of configuration data such as an API key or credential file.
Resource

A resource represents a component of a provider such as an "AWS instance", "DNSimple Record", or "Fastly service". Resources have both *arguments* (inputs) and *attributes* (outputs) which are specific to the resource. Resources also have meta-parameters such as *count* and *lifecycle*. 
(Resource) Argument

An argument is an input or configuration option to a resource. An AWS EC2 instance accepts `ami` as an input parameter. This makes `ami` an argument to the `aws_instance` resource.
An attribute is an output or computed value available only after resource creation. An AWS EC2 instance provides `public_ip` as an output parameter. This makes `public_ip` an attribute to the `aws_instance` resource. This makes sense, because an instance's IP address is assigned during creation.
**Graph**

The graph is the internal structure for Terraform's resource dependencies and order. The graph implements a directed acyclic graph (DAG) which allows Terraform to optimize for parallelism while adhering to dependency ordering. It is possible to generate the graph as a DOT file for human viewing.
(Remote/Local) State

Terraform stores the last-known arguments and attributes for all resources. These contents known as "state" can be stored locally as a JSON file (local state) or stored in a remote shared location like Atlas (remote state).
Module

A module is a self-contained package of Terraform configurations. Modules are like abstract classes that are imported into other Terraform configurations.

Parallels: Chef Cookbook, Puppet Module, Ruby gem
Variable

A variable is a user or machine-supplied input in Terraform configurations. Variables can be supplied via environment variables, CLI flags, or variable files. Combined with modules, variables help make Terraform flexible, sharable, and extensible.
Interpolation

Terraform includes a built-in syntax for referencing attributes of other resources. This technique is called interpolation. Terraform also provides built-in functions for performing string manipulations, evaluating math operations, and doing list comprehensions.
HashiCorp Configuration Language (HCL)

Terraform's syntax and interpolation are part of an open source language and specification called HCL.
Infrastructure as Code
Infrastructure as Code

Provide a codified workflow to create infrastructure

Expose a workflow for managing updates to existing infrastructure

Integrate with application code workflows (Git, SCM, Code Review)

Provide modular, sharable components for separation of concerns
Human-readable configuration (HCL) is designed for human consumption so users can quickly interpret and understand their infrastructure configuration.

HCL is fully JSON-compatible for machine-generated configurations.
resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}
Infrastructure as Code (Terraform)

Configuration format is very VCS friendly with support for multi-line lists, trailing commas, and auto-formatting.
resource "aws_instance" "web" {
  ami = "ami-9a562df2"
- instance_type = "t2.micro"
+ instance_type = "m1.small"
}

Reference values from other resources, building the implicit dependency graph.
resource "aws_instance" "web" {
  ami           = "ami-9a562df2"
  instance_type = "t2.micro"
}

resource "dnsimple_record" "web" {
  domain = "hashicorp.com"
  name   = "web"
  ttl    = "3600"
  type   = "A"
  value  = "${aws_instance.web.public_ip}"
Infrastructure as Code (Terraform)

Configuration can be in a single file or split across multiple files. Terraform will merge all files in the current working directory which end in `.tf` or `.tf.json`. 
Terminal

$ ls
main.tf
outputs.tf
variables.tf
Terminal

$ ls
instances.tf
load-balancers.tf
shared.tf
$ ls
everything.tf
Configuration Format
resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}
resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}

resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}
resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}
resource "aws_instance" "web" {
    ami           = "ami-9a562df2"
    instance_type = "t2.micro"
}

resource "aws_instance" "web" {
    ami           = "ami-6b563df1"
    instance_type = "t2.micro"
}
resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}

resource "aws_instance" "web-2" {
  ami = "ami-6b563df1"
  instance_type = "t2.micro"
}
resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}

resource "digitalocean_droplet" "web" {
  ami = "ami-6b563df1"
  instance_type = "t2.micro"
}
resource "aws_instance" "web" {
    ami = "ami-9a562df2"
    instance_type = "t2.micro"
}
resource "aws_instance" "web" {
    ami = "ami-9a562df2"
    instance_type = "t2.micro"
}
resource "aws_instance" "web" {
  ami           = "ami-9a562df2"
  instance_type = "t2.micro"
}

resource "dnsimple_record" "web" {
  domain = "hashicorp.com"
  name   = "web"
  ttl    = "3600"
  type   = "A"
  value  = "${aws_instance.web.public_ip}"}

resource "aws_instance" "web" {
  ami = "ami-9a562df2"
  instance_type = "t2.micro"
}

resource "dnsimple_record" "web" {
  domain = "hashicorp.com"
  name = "web"
  ttl = "3600"
  type = "A"
  value = "${aws_instance.web.public_ip}"
Syntax Highlighting

Plugins for HCL exist for most major editors

If not:

- "javascript" if using // comments
- "ruby" if using # comments

Community leans toward # comments
Exercise: Explore Workstation

SSH into your workstation using the provided credentials.

Change directory into `/workstation` and look inside `main.tf`
# Your security group ID is:
#
#     abcd1234
#
# ...
#

provider "aws" {
    access_key = "AKIAIZT7VATBGPC3AP3Q"
    secret_key = "W/txHyHA8SsevaENCsUIS9/KGoIScy2yoqrfodKz"
    region     = "us-west-1"
}
In the same Terraform configuration named `main.tf` create one AWS instance with the ID "web" with the properties described in the `main.tf` file and:

- ami (see file)
- `instance_type` (`t2.micro`)
- `subnet_id` (see file)
- `vpc_security_group_ids` (see file, HINT: this is an array)
- `tags.Identity` (see file, case-sensitive)
provider "aws" {
  access_key = "AKIAIZT7VATBGPC3AP3Q"
  secret_key = "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"
  region     = "us-west-1"
}

resource "aws_instance" "web" {
  ami           = "ami-31106a51"
  instance_type = "t2.micro"

  subnet_id               = "subnet-c02e6198"
  vpc_security_group_ids = ["sg-b1fe76ca"]

  tags {
    Identity = "..."
  }
}
Resource Graph
The resource graph is an internal representation of all resources and their dependencies.

A human-readable graph can be generated using the `terraform graph` command.

Can optionally draw cycles (advanced).
Run the terraform graph command against the Terraform configurations from the previous exercise.

Paste the output into www.webgraphviz.com to generate the graph.
```bash
$ terraform graph
digraph {
    compound = "true"
    newrank = "true"
    subgraph "root" {
        "[root] aws_instance.web" [label = "aws_instance.web", shape ...]
        "[root] provider.aws" [label = "provider.aws", shape = "diamond"]
        "[root] aws_instance.web" -> "[root] provider.aws"
    }
}
```
Exercise: Run `terraform graph`
Terraform Graph

Useful for visualizing infrastructure and dependencies

Builds upon existing visualization technologies and open formats such as DOT
Command Line Interface
All interactions with Terraform occur via the CLI.

Terraform is a local tool (runs on the current machine).
Exercise: Run `terraform help`

Run `terraform help` to generate the full list of Terraform commands.
$ terraform help

**Common commands:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>Builds or changes infrastructure</td>
</tr>
<tr>
<td>destroy</td>
<td>Destroy Terraform-managed infrastructure</td>
</tr>
<tr>
<td>fmt</td>
<td>Rewrites config files to canonical format</td>
</tr>
<tr>
<td>get</td>
<td>Download and install modules for the configuration</td>
</tr>
<tr>
<td>graph</td>
<td>Create a visual graph of Terraform resources</td>
</tr>
<tr>
<td>import</td>
<td>Import existing infrastructure into Terraform</td>
</tr>
<tr>
<td>init</td>
<td>Initializes Terraform configuration from a module</td>
</tr>
<tr>
<td>output</td>
<td>Read an output from a state file</td>
</tr>
<tr>
<td>plan</td>
<td>Generate and show an execution plan</td>
</tr>
<tr>
<td>push</td>
<td>Upload this Terraform module to Atlas to run</td>
</tr>
<tr>
<td>refresh</td>
<td>Update local state file against real resources</td>
</tr>
<tr>
<td>remote</td>
<td>Configure remote state storage</td>
</tr>
<tr>
<td>show</td>
<td>Inspect Terraform state or plan</td>
</tr>
<tr>
<td>taint</td>
<td>Manually mark a resource for recreation</td>
</tr>
</tbody>
</table>

# ...
Command: `terraform plan`

The plan shows you what will happen.

You can save plans to guarantee what will happen.

Plans show reasons for certain actions (such as re-create).

Prior to Terraform, users had to guess change ordering, parallelization, and rollout effect.
Command: terraform plan

+ indicates a resource will be created

- indicates a resource will be destroyed

~ indicates a resource will be updated in-place

−/+ indicates a resource will be destroyed and re-created
Exercise: Run `terraform plan`

Run `terraform plan` on the Terraform files created in the previous section. Leave the output on the screen for the instructor to see.
$ terraform plan

+ aws_instance.web
  ami:                      "ami-31106a51"
  availability_zone:       "<computed>"
  ebs_block_device.#:      "<computed>"
  ephemeral_block_device.#: "<computed>"
  instance_state:         "<computed>"
  instance_type:          "t2.micro"
  key_name:               "<computed>"
  placement_group:        "<computed>"
  private_dns:            "<computed>"
  private_ip:             "<computed>"
  public_dns:             "<computed>"
  public_ip:              "<computed>"
  root_block_device.#:    "<computed>"
  security_groups.#:      "<computed>"

# ...
Command: `terraform apply`

- Executes changes in order based on the resource graph
- Parallelizes changes when possible
- Handles and recovers transient errors
Exercise: Run `terraform apply`

Run `terraform apply` on the Terraform files created in the previous section. This will create real resources in AWS.
$ terraform apply
aws_instance.web: Creating...
   ami:       "" => "ami-31106a51"

   # ...

   source_dest_check: "" => "true"
   subnet_id:        "" => "subnet-f6e6a5dc"

aws_instance.web: Still creating... (10s elapsed)
aws_instance.web: Still creating... (20s elapsed)
aws_instance.web: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Command: `terraform apply`

- Updates existing resources when updates are allowed
- Re-creates existing resources when updates are not allowed
Edit the main Terraform file and add at least two tags to the AWS instance.

HINT: You may need to look at Terraform's documentation for the syntax (terraform.io).
resource "aws_instance" "web" {
  ami            = "ami-50759d3d"
  instance_type = "t2.micro"

  subnet_id = "sg-22a34f8"
  vpc_security_group_ids = ["vpc-ab2da4e"]

  tags {
    Identity = "..."
    Foo      = "bar"
    Zip      = "zap"
  }
}
$ terraform plan

~ aws_instance.web
  tags.#:   "1" => "3"
  tags.Foo: "" => "bar"
  tags.Zip: "" => "zap"

Plan: 0 to add, 1 to change, 0 to destroy.
$ terraform apply
aws_instance.web: Refreshing state... (ID: i-02f5717f1a84502ed)
aws_instance.web: Modifying...
  tags.#:   "1" => "3"
  tags.Foo: "" => "bar"
  tags.Zip: "" => "zap"
aws_instance.web: Modifications complete

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
Command: terraform show

Displays human-friendly state

Can be used to get on-the-fly information
Exercise: `terraform show`

Run `terraform show` and get the IP address of the instance just created.
$ terraform show
aws_instance.web:
  id = i-02f5717f1a84502ed
  ami = ami-31106a51
  availability_zone = us-east-1c
  disable_api_termination = false
  ebs_block_device.# = 0
  ebs_optimized = false
  ephemeral_block_device.# = 0
  iam_instance_profile =
  instance_state = running
  instance_type = t2.micro
  key_name =
  monitoring = false
  private_dns = ip-10-0-1-237.ec2.internal
  private_ip = 10.0.1.237
  public_dns = ec2-52-201-233-198.compute-1.amazonaws.com
  public_ip = 52.201.233.198
$ ping 52.201.233.198
PING 52.201.233.198 (52.201.233.198) 56(84) bytes of data.
64 bytes from 52.201.233.198: icmp_seq=1 ttl=63 time=0.807 ms
64 bytes from 52.201.233.198: icmp_seq=2 ttl=63 time=0.770 ms
Variables, Outputs, & Meta Parameters
Variables

Define the parameterization of Terraform configurations

Can have defaults, be provided with a variables file, asked for at execution, or overridden via the CLI

Values can be strings or maps

Must be defined before used
variable "aws_access_key" {}
Exercise: Create Variables

Create three variables in the Terraform configuration:

- `aws_access_key` (no default value)
- `aws_secret_key` (no default value)
- `aws_region` (default: "us-east-1")
variable "aws_access_key" {}  
variable "aws_secret_key" {}  
variable "aws_region" {  
    default = "us-west-1"  
}  

provider "aws" {  
    access_key = "AKIAIZT7VATBGPC3AP3Q"  
    secret_key = "W/txHyHA8SsevaENCsUIS9/KGoIScy2yoqrfdKz"  
    region     = "us-west-1"  
}  

# ...
Variables are available in the interpolation syntax.
variable "aws_access_key" {}
variable "aws_secret_key" {}
variable "aws_region" {
  default = "us-west-1"
}

provider "aws" {
  access_key = "${var.aws_access_key}"
  secret_key = "..."
  region     = "..."
}

# ...

Exercise: Reference Variables

Replace the hard-coded values in the provider with references to the newly-defined variables using the interpolation syntax.
variable "aws_access_key" {}  
variable "aws_secret_key" {}  
variable "aws_region" {  
    default = "us-west-1"
}

provider "aws" {  
    access_key = "${var.aws_access_key}"  
    secret_key = "${var.aws_secret_key}"  
    region = "${var.aws_region}"
}

# ...
Exercise: Run `terraform plan`

Run `terraform plan` now that we the variables are parameterized. Observe what happens.
$ terraform plan
var.aws_access_key
    Enter a value:

<CTRL + C>
Exercise: Create tfvars file

Uncomment the lines in `terraform.tfvars` so Terraform picks up the defined variables.
aws_access_key = "AKIAIZT7VATBGC3AP3Q"
aws_secret_key = "W/txHyHA8SsevaENCsUIS9/KGoIScy2yoqrfodKz"
$ terraform plan
aws_instance.web: Refreshing state... (ID: i-02f5717f1a84502ed)

No changes. Infrastructure is up-to-date. This means that Terraform could not detect any differences between your configuration and the real physical resources that exist. As a result, Terraform doesn't need to do anything.
Outputs define values that will be highlighted to the user when Terraform applies.

Outputs can be queried using the `terraform output` command.
resource "aws_instance" "web" {
    # ...
}

output "public_ip" {
    value = "${aws_instance.web.public_ip}" 
}
Create a new output variable named "address" which outputs the instance's `public_dns` attribute.
resource "aws_instance" "web" {
  ami = "ami-50759d3d"
  instance_type = "t2.micro"
  subnet_id = "sg-22a34f8"
  vpc_security_group_ids = ["vpc-ab2da4e"]
}
output "public_ip" {
  value = "${aws_instance.web.public_ip}" 
}
output "public_dns" {
  value = "${aws_instance.web.public_dns}" 
}
Exercise: Run `terraform apply`

Run `terraform apply` to pick up the new output values.
$ terraform apply
aws_instance.web: Refreshing state... (ID: i-02f5717f1a84502ed)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

public_dns = ec2-52-201-233-198.compute-1.amazonaws.com
public_ip = 52.201.233.198
Exercise: Query Output

Use the `terraform output` command to query for all outputs.

Use the `terraform output` command to query for a single output.
$ terraform output
public_dns = ec2-52-201-233-198.compute-1.amazonaws.com
public_ip = 52.201.233.198
$ terraform output public_dns
c2-52-201-233-198.compute-1.amazonaws.com
$ ping $(terraform output public_dns)
PING 54.193.3.16 (54.193.3.16) 56(84) bytes of data.
64 bytes from 54.193.3.16: icmp_seq=1 ttl=63 time=0.757 ms
64 bytes from 54.193.3.16: icmp_seq=2 ttl=63 time=0.680 ms
# ...
Meta-parameters allow for higher-level control flow and lifecycle management in Terraform.
**Count**

The `count` attribute allows for N number of identical resources to be created. This removes the need for complex variable iteration with "for" or "while" loops.
**Meta-Parameters**

**Depends On**

The `depends_on` attribute allows for declaration of explicit dependencies. This is useful where interpolation is not required, but explicit ordering is desired.
Meta-Parameters

 Lifecycle

The `lifecycle` attribute allow explicit configuration of resource lifecycle such as preventing destruction or ignoring property changes. This is an advanced option and is not recommended for users who are getting started with Terraform.
Increase the `count` parameter on the AWS instance to two (2).

Run `terraform plan` and verify the output.

Run `terraform apply` to apply the changes.
resource "aws_instance" "web" {
  count = "2"
  ami   = "ami-50759d3d"
  instance_type = "t2.micro"

  subnet_id          = "sg-22a34f8"
  vpc_security_group_ids = ["vpc-ab2da4e"]

  tags {
    # ...
  }
}

# ...

$ terraform plan

+ aws_instance.web.1
  ami: "ami-31106a51"
  availability_zone: "<computed>"
  ebs_block_device.#: "<computed>"
  ephemeral_block_device.#: "<computed>"
  instance_state: "<computed>"
  instance_type: "t2.micro"
  key_name: "<computed>"
  placement_group: "<computed>"
  private_dns: "<computed>"
  private_ip: "<computed>"
  public_dns: "<computed>"
  public_ip: "<computed>"
  root_block_device.#: "<computed>"
  security_groups.#: "<computed>"
  source_dest_check: "true"
$ terraform apply
aws_instance.web.0: Refreshing state... (ID: i-02f5717f1a84502ed)
aws_instance.web.1: Creating...
  ami:                               "" => "ami-31106a51"
  availability_zone:                "" => "<computed>"
  ebs_block_device.#:              "" => "<computed>"
  vpc_security_group_ids.#:         "" => "1"
  vpc_security_group_ids.3898593916: "" => "sg-1cf17a67"
# ...
aws_instance.web.1: Still creating... (10s elapsed)
aws_instance.web.1: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
$ terraform output
$ terraform output
The state file has no outputs defined. Define an output in your configuration with the `output` directive and re-run `terraform apply` for it to become available.
What Happened?

Terraform defaults all resources to a count of "1".

By increasing the count, we now have an array of resources instead of a single resource.

Terraform does not know which instance to get the output from.
Exercise: Modify Output

Modify the outputs to print out the value for the *first* instance.

HINT: the array is zero-indexed (.0.public_ip)
output "public_ip" {
  value = "${aws_instance.web.0.public_ip}" 
}

output "public_dns" {
  value = "${aws_instance.web.0.public_dns}" 
}
$ terraform apply
aws_instance.web.0: Refreshing state... (ID: i-02f5717f1a84502ed)
aws_instance.web.1: Refreshing state... (ID: i-01f0c344e8e0cf568)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

public_dns = ec2-52-201-233-198.compute-1.amazonaws.com
public_ip = 52.201.233.198
Exercise: List all instance IP and DNS addresses

Modify the outputs to print out the IP and DNS for all instances.

HINT: there is a splat operator (*)
output "public_ip" {
  value = "${join("", ",", aws_instance.web.*.public_ip)}"
}

output "public_dns" {
  value = "${join("", ",", aws_instance.web.*.public_dns)}"
}
$ terraform apply
aws_instance.web.0: Refreshing state... (ID: i-02f5717f1a84502ed)
aws_instance.web.1: Refreshing state... (ID: i-01f0c344e8e0cf568)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

  public_dns = ec2-54-86-255-249.compute-1.amazonaws.com, ec2-54-165-126-194.compute-1.amazonaws.com
  public_ip  = 54.86.255.249, 54.165.126.194
# Coming soon (Terraform 0.7+!)
output "public_ip" {
    value = ["${aws_instance.web.*.public_ip}"]
}

output "public_dns" {
    value = ["${aws_instance.web.*.public_dns}"]
}
Modules
Modules

Portable Terraform configurations (packages)

Allow separation of concerns and responsibilities among teams

Parallels: Chef Cookbook, Puppet Module, Ruby gem
Spoiler Alert!

Modules are just Terraform configurations inside a folder - there's nothing special about them.
Variables => Arguments

Outputs => Attributes

Configuration is sandbox/blackbox

Individual resource arguments are not accessible in module
$ tree my-module
my-module
  |-- main.tf
variable "argument_1" {}  
variable "argument_2" {}  

resource "aws_instance" "db" {  
  # ...
}  

output "address" {  
  value = "${aws_instance.db.public_dns}"  
}
module "my-module" {
    source = "../../my-module" # Source can be any URL or file path

    argument_1 = "value"
    argument_2 = "value"
}

output "example" {
    value = "${module.my-module.address}"
}
module "my-module" {
  source = "../../my-module" # Source can be any URL or file path

  argument_1 = "value"
  argument_2 = "value"
}

output "example" {
  value = "${module.my-module.aws_instance.db.public_dns}"
}
module "consul" {
    source = "github.com/sethvargo/tf-consul-atlas-join"
    ami    = "ami-31106a51"

    subnet_id    = "subnet-f6e6a5dc"
    security_group = "sg-1cf17a67"
    key_name     = "${var.key_name}"

    atlas_environment = "${var.atlas_environment}"
    atlas_token      = "${var.atlas_token}"}

State
Terraform stores the state of your managed infrastructure from the last time Terraform was run.

Terraform uses this state to create plans and make changes to your infrastructure.

It is critical that this state is maintained appropriately so future runs operate as expected.
$ cat terraform.tfstate
{
  "version": 2,
  "terraform_version": "0.7.0",
  "serial": 5,
  "modules": [
    {
      "path": ["root"],
      "outputs": {
        "public_dns": {
          "sensitive": false,
          "type": "list",
          "value": [
            "ec2-52-201-233-198.compute-1.amazonaws.com",
            "ec2-54-82-235-215.compute-1.amazonaws.com",
            "ec2-54-82-235-215.compute-1.amazonaws.com"
          ]
        }
      }
    }
  ]
}
State is stored locally on one machine in JSON format

Generally acceptable for individuals and small teams

Does not scale to large teams

Resolving JSON git-diffs is hard
Remote State

State is stored on a shared remote source such as Atlas or Consul

Remote storage is responsible for handling merging and locking

Unnecessary overhead for small teams

Best-suited for large or distributed teams
Multi-Provider
All resources depend on a provider.

Providers must be configured with arguments (varies by provider).
provider "aws" {
    access_key = "AKIAIZT7VATBGPC3AP3Q"
    secret_key = "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"
    region     = "us-west-1"
}
Add a new provider for `dnsimple` in `dns.tf`.

HINT: You may need to look on Terraform's website to find the required arguments to the provider.
provider "dnsimple" {
  email = "sethvargo+terraform@gmail.com"
  token = "..."
}

Exercise: Create DNS record

Create a new `dnsimple_record` resource.

- domain (`terraform.rocks`)
- type (A)
- name (`your unique identity`)
- value (the public IP of your first web instance)

Do **NOT** hardcode the IP address! Use interpolation to make the IP address lookup dynamic.
provider "dnsimple" {
  email = "sethvargo+terraform@gmail.com"
  token = "..."
}

resource "dnsimple_record" "example" {
  domain = "terraform.rocks"
  type   = "A"
  name   = "...
  value  = "${aws_instance.web.public_ip}"
Exercise: Plan and Apply

Run `terraform plan` to verify the changes.

Run `terraform apply` to apply them.
$ terraform plan

+ dnsimple_record.web
  domain: "" => "terraform.rocks"
  domain_id: "" => "<computed>"
  hostname: "" => "<computed>"
  name: "" => "testing"
  priority: "" => "<computed>"
  ttl: "" => "3600"
  type: "" => "A"
  value: "" => "54.86.255.249"
$ terraform apply
aws_instance.web.0: Refreshing state... (ID: i-035fee6dd97089b75)
aws_instance.web.1: Refreshing state... (ID: i-0406ad0449f dbb4fc)
dnsimple_record.web: Creating...
  domain:    "" => "terraform.rocks"
domain_id: "" => "<computed>"
hostname:  "" => "<computed>"
name:      "" => "testing"
priority:  "" => "<computed>"
ttl:       "" => "3600"
type:      "" => "A"
value:     "" => "54.86.255.249"
dnsimple_record.web: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
$ dig <identity>.terraform.rocks
; <<>> DiG 9.8.3-P1 <<>> <identity>.terraform.rocks
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31653
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
<identity>.terraform.rocks.            IN     A

;; ANSWER SECTION:
<identity>.terraform.rocks.    3420    IN     A     52.201.233.198

;; Query time: 40 msec
;; SERVER: 10.137.0.1#53(10.137.0.1)
;; MSG SIZE  rcvd: 57
Using Atlas
Demo
Terraform Enterprise: Review

Integrates with SCM and application workflows

Provides a dry-run experience for infrastructure changes

Continuous Integration, but for Infrastructure as Code
Destroy
Command: `terraform destroy`

Destroys running infrastructure

Does not touch infrastructure not managed by Terraform
Exercise: `terraform destroy`

Run `terraform plan -destroy` to plan a destroy.

Run `terraform destroy` to destroy the instance and DNS record.
$ terraform plan -destroy
- aws_instance.web.0
- dnsimple_record.example

Plan: 0 to add, 0 to change, 2 to destroy
$ terraform destroy

Do you really want to destroy?
   Terraform will delete all your managed infrastructure.
   There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

aws_instance.web.0: Refreshing state... (ID: i-035fee6dd97089b75)
dnsimple_record.example: Refreshing state... (ID: 5916121)
dnsimple_record.example: Destroying...
dnsimple_record.example: Destruction complete
aws_instance.web.0: Destroying...
aws_instance.web.0: Still destroying... (10s elapsed)
aws_instance.web.0: Destruction complete

Apply complete! Resources: 0 added, 0 changed, 2 destroyed