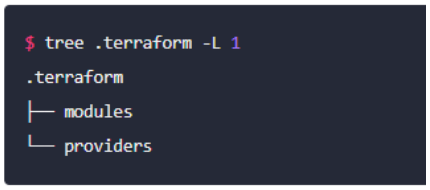
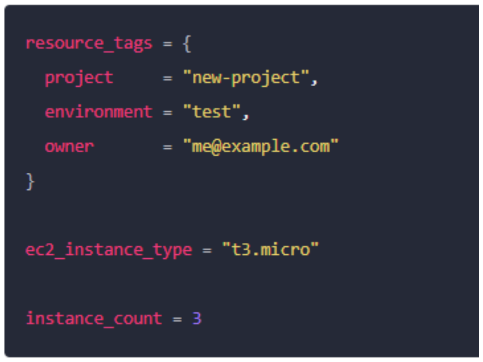
Files and Directories	Code in the Terraform language is stored in plain text files with the .tf file extension. **A *module*** is a collection of .tf and/or .tf.json files kept together in a directory. **The Root Module:** Terraform always runs in the context of a single root module. Terraform consists of a root module and the tree of child modules. And complete dir/file structure here: **.terraform.lock.hcl** .tfvars .terraform directory
Dependency Lock File	A terraform configuration may refer to two different kinds of external dependency. 1. Providers: Plugins for terraform. (AWS, Azure RM) 2. Modules: splitting out groups of Terraform configuration into reusable abstractions. Version constraints within the configuration itself determine which versions of dependencies are potentially compatible, but after selecting a specific version of each dependency Terraform remembers the decisions it made in a dependency lock file (**.terraform.lock.hcl**) so that it can (by default) make the same decisions again in future. Terraform automatically creates or updates the dependency lock file each time you run terraform init command. Should include the file in version control. If a particular provider already has a selection recorded in the lock file, Terraform will always re-select that version for installation, even if a newer version has become available. You can override that behavior by adding the -upgrade option when you run terraform init.

.terraform.lock.hcl

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/random" {
  version     = "3.1.0"
  constraints = "3.1.0"
  hashes = [
    "h1:rKYu5ZUbXwrLG1w81k7H3nce/Ys6yAxXhWcbtk36HjY=",
    ## ...
    "zh:f7605bd1437752114baf601bdf6931debe6dc6bfe3006eb7e9bb9080931dca8a",
  ]
}

provider "registry.terraform.io/kreuzwerker/docker" {
  version     = "2.16.0"
  constraints = "~> 2.16.0"
  hashes = [
    "h1:FyU8TUgpwfu+O+k+Uu5N58I/JWlEZk2PzQLJMluuaIQ=",
    ## ...
    "zh:fd634e973eb2b6483a1ce9251801a393d04cb496f8e83ffcf3f0c4cad8c18f4c",
  ]
}
```

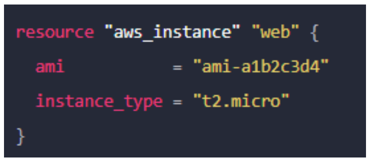
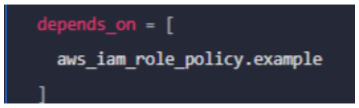
.terraform **directory**	Terraform uses the .terraform directory to store the project's providers and modules. Terraform will refer to these components when you run validate, plan, and apply. View the .terraform directory structure.

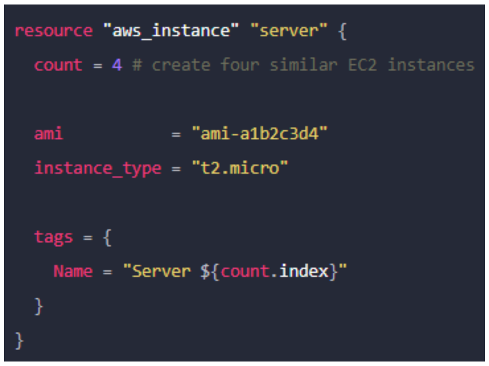
	
	`$ tree .terraform -L 1`
	`.terraform`
	`├── modules`
	`└── providers`

terraform.tfvars	Whenever you execute a plan, destroy, or apply with any variable unassigned, Terraform will prompt you for a value. Entering variable values manually is time consuming and error prone, so Terraform provides several other ways to assign values to variables.
	
	Create a file named terraform.tfvars with the following contents.
	
	
	`resource_tags = {`
	`  project     = "new-project",`
	`  environment = "test",`
	`  owner       = "me@example.com"`
	`}`
	
	`ec2_instance_type = "t3.micro"`
	
	`instance_count = 3`
	
	Terraform automatically loads all files in the current directory with the exact name terraform.tfvars or matching *.auto.tfvars. You can also use the -var-file flag to specify other files by name.

Priority Order of Variables	Priority Order of Variables

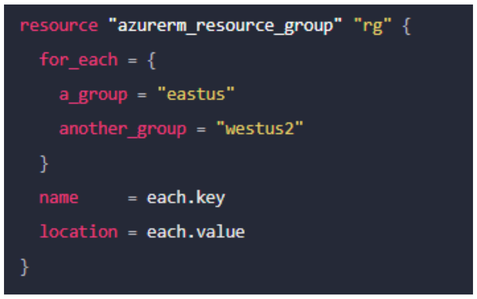
	• A variable value file explicitly referenced using a "-var" flag. • A ".tfvars" file explicitly referenced using a "-var-file" flag. • A file with the ".auto.tfvars" extension. • A file called "terraform.tfvars". • An environment variable with the "TF_VAR_name" format. • The default value in the variable definition.

Resource Blocks	**Resource Syntax**
	
	`resource "aws_instance" "web" {`
	`  ami           = "ami-a1b2c3d4"`
	`  instance_type = "t2.micro"`
	`}`
	
	depends_on: to handle hidden resource or module dependencies that Terraform can not automatically infer.
	
	`  depends_on = [`
	`    aws_iam_role_policy.example`
	`  ]`
	
	Count: creates multiple instances of resource or module.



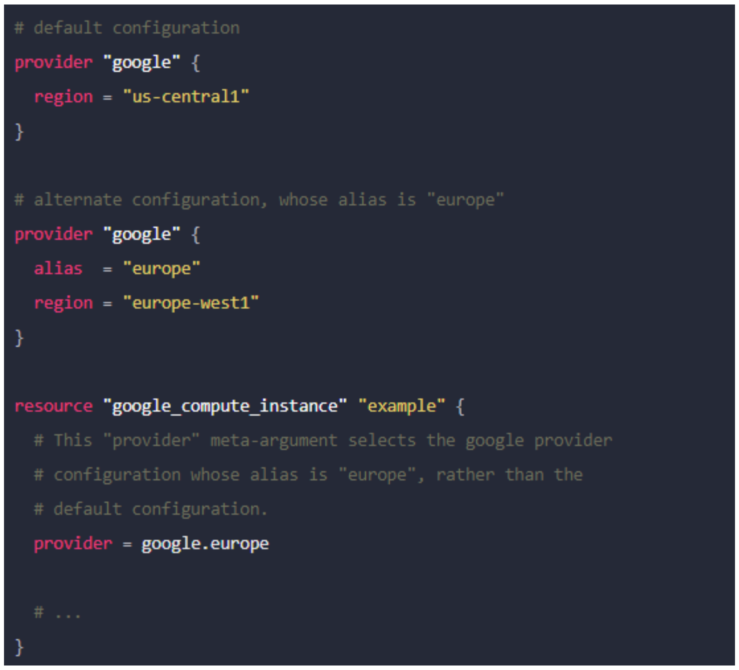
```
resource "aws_instance" "server" {
  count = 4 # create four similar EC2 instances

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${count.index}"
  }
}
```

For_each: accepts a map or set of strings, and creates an instance for each item in that map or set.



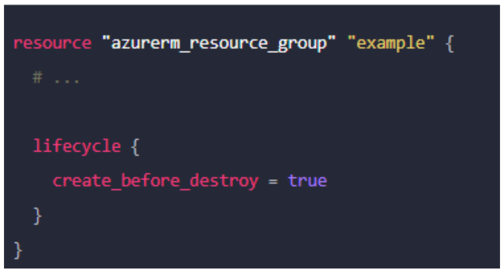
```
resource "azurerm_resource_group" "rg" {
  for_each = {
    a_group = "eastus"
    another_group = "westus2"
  }
  name     = each.key
  location = each.value
}
```

Provider:



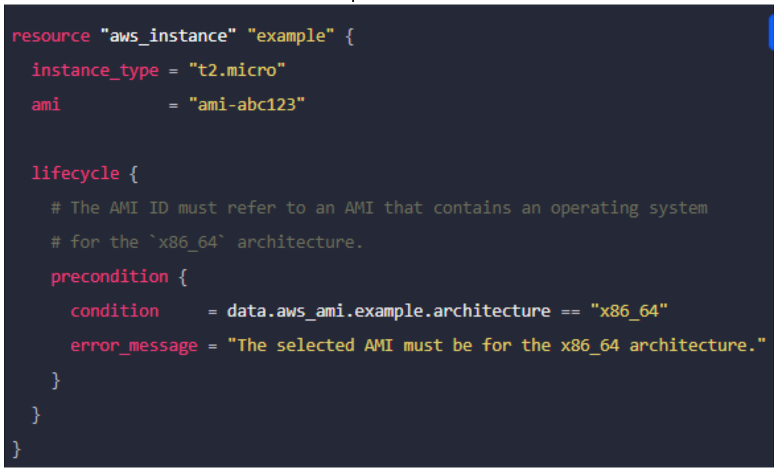
```
# default configuration
provider "google" {
  region = "us-central1"
}

# alternate configuration, whose alias is "europe"
provider "google" {
  alias  = "europe"
  region = "europe-west1"
}

resource "google_compute_instance" "example" {
  # This "provider" meta-argument selects the google provider
  # configuration whose alias is "europe", rather than the
  # default configuration.
  provider = google.europe

  # ...
}
```

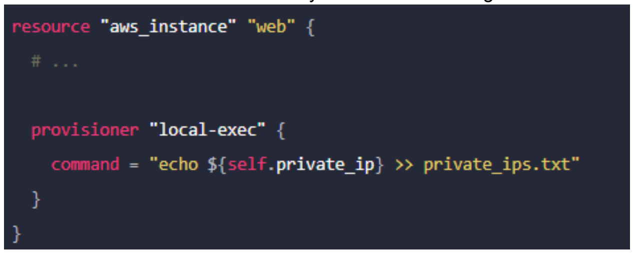
Lifecycle:

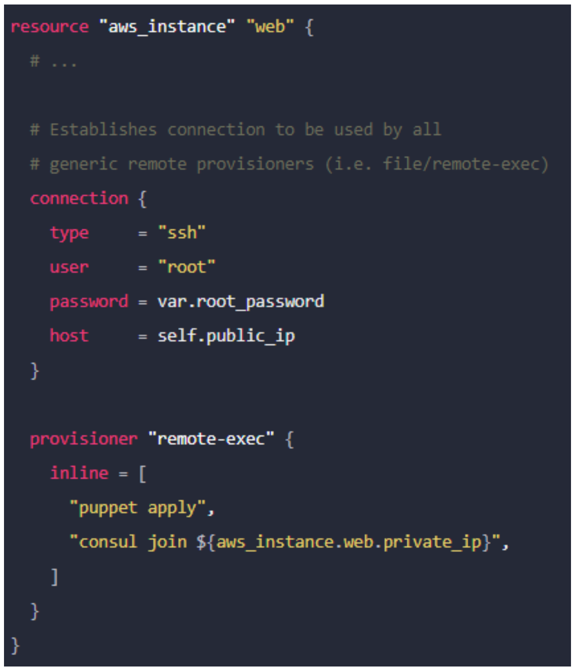


```
resource "azurerm_resource_group" "example" {
  # ...

  lifecycle {
    create_before_destroy = true
  }
}
```

The arguments available within lifecycle block are:
create_before_destroy

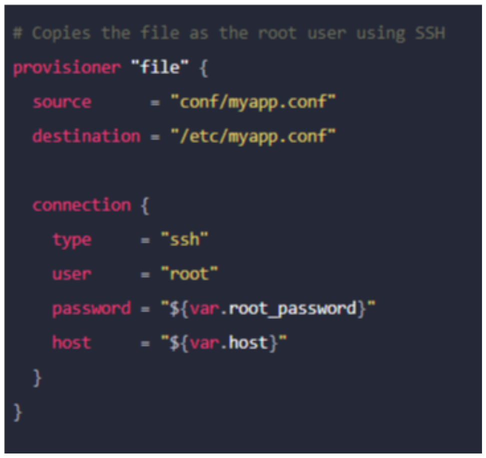
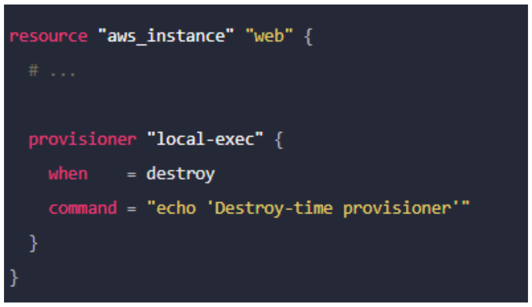
	prevent_destroy
	Ignore_changes
	replace_triggered_by
	
	Custom Condition Checks:
	You can add precondition and postcondition blocks with a lifecycle block to specify assumptions and guarantees about how resources and data sources operate
	
	`resource "aws_instance" "example" {`
	`  instance_type = "t2.micro"`
	`  ami           = "ami-abc123"`
	
	`  lifecycle {`
	`    # The AMI ID must refer to an AMI that contains an operating system`
	`    # for the `x86_64` architecture.`
	`    precondition {`
	`      condition     = data.aws_ami.example.architecture == "x86_64"`
	`      error_message = "The selected AMI must be for the x86_64 architecture."`
	`    }`
	`  }`
	`}`

Provisioners:	It is recommended to not use a provisioner unless there is no alternative.
	
	Local-exec Provisioner:
	Provisioner commands are run locally on machine running terraform.
	
	`resource "aws_instance" "web" {`
	`  # ...`
	
	`  provisioner "local-exec" {`
	`    command = "echo ${self.private_ip} >> private_ips.txt"`
	`  }`
	`}`
	
	Remote-exec Provisioner:
	
	Commands are run on remote resources after it is created. Requires connection.



```
resource "aws_instance" "web" {
  # ...

  # Establishes connection to be used by all
  # generic remote provisioners (i.e. file/remote-exec)
  connection {
    type     = "ssh"
    user     = "root"
    password = var.root_password
    host     = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "puppet apply",
      "consul join ${aws_instance.web.private_ip}",
    ]
  }
}
```

File Provisioner: Copies files or directories from local to remote.

	 `# Copies the file as the root user using SSH`
	`provisioner "file" {`
	`  source      = "conf/myapp.conf"`
	`  destination = "/etc/myapp.conf"`
	
	`  connection {`
	`    type     = "ssh"`
	`    user     = "root"`
	`    password = "${var.root_password}"`
	`    host     = "${var.host}"`
	`  }`
	`}`
	By default provisioners are run when resources are created. If when = destroy the provisioner will run when a resource defined within is destroyed.
	 `resource "aws_instance" "web" {`
	`  # ...`
	
	`  provisioner "local-exec" {`
	`    when    = destroy`
	`    command = "echo 'Destroy-time provisioner'"`
	`  }`
	`}`
	On_failure = continue or fail

Provider configuration	Terraform configurations must declare which providers they require so that Terraform can install and use them.

Requiring Providers

Each Terraform module must declare which providers it requires, so that Terraform can install and use them. Provider requirements are declared in a required_providers block.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}
```

Default Provider Configurations

A provider block without an alias argument is the default configuration for that provider.

Credentials:

```
provider "aws" {
  region     = "us-west-2"
  access_key = "my-access-key"
  secret_key = "my-secret-key"
}
```

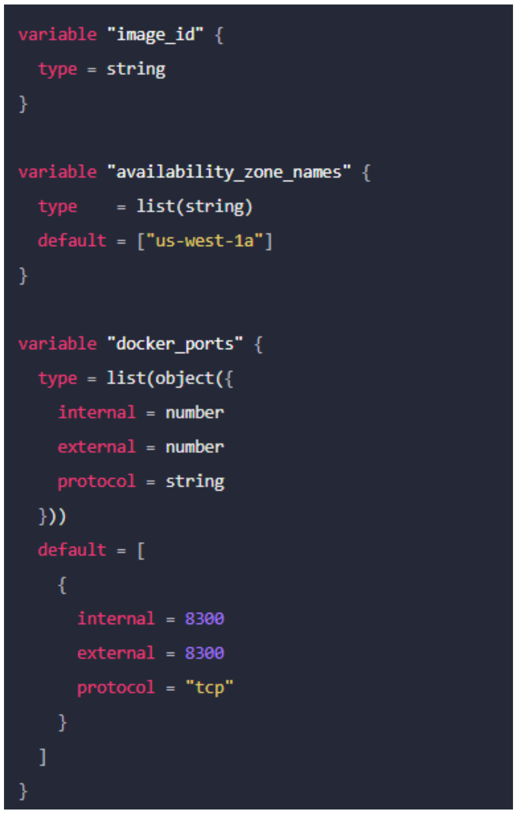
Environment Variables

	Credentials can be provided by using the AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and optionally AWS_SESSION_TOKEN environment variables
	
	`provider "aws" {`
	`  shared_config_files      = ["/Users/tf_user/.aws/conf"]`
	`  shared_credentials_files = ["/Users/tf_user/.aws/creds"]`
	`  profile                  = "customprofile"`
	`}`
	
	`provider "aws" {`
	`  assume_role {`
	`    role_arn     = "arn:aws:iam::123456789012:role/ROLE_NAME"`
	`    session_name = "SESSION_NAME"`
	`    external_id  = "EXTERNAL_ID"`
	`  }`
	`}`

	`provider "azurerm" {`
	`  features {}`
	
	`  use_msi = true`
	
	`  backend "azurerm" {`
	`    storage_account_name = "abcd1234"`
	`    container_name       = "tfstate"`
	`    key                  = "prod.terraform.tfstate"`
	`    subscription_id      = "00000000-0000-0000-0000-000000000000"`
	`    tenant_id            = "00000000-0000-0000-0000-000000000000"`
	`  }`
	`}`

Variables and Outputs	● Input Variables serve as parameters for a Terraform module, so users can customize behavior without editing the source. ● Output Values are like return values for a Terraform module. ● Local Values are a convenience feature for assigning a short name to an expression.

Input Variables



```
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"
    }
  ]
}
```

Arguments:

- default - A default value which then makes the variable optional.
- type - This argument specifies what value types are accepted for the variable.
- description - This specifies the input variable's documentation.
- validation - A block to define validation rules, usually in addition to type constraints.
- sensitive - Limits Terraform UI output when the variable is used in configuration.

- nullable - Specify if the variable can be null within the module.

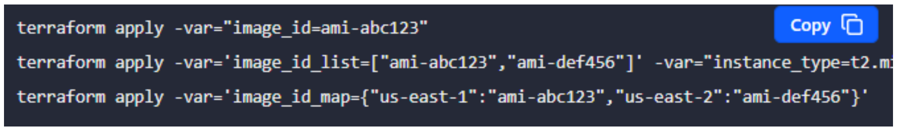
Types:

- string
- number
- bool

The type constructors allow you to specify complex types such as collections:

- list(<TYPE>)
- set(<TYPE>)
- map(<TYPE>)
- object({<ATTR NAME> = <TYPE>, ... })
- tuple([<TYPE>, ...])

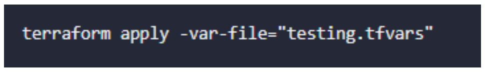
Variables on the Command Line



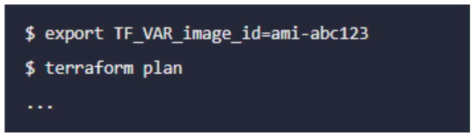
```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -var="instance_type=t2.m
terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'
```

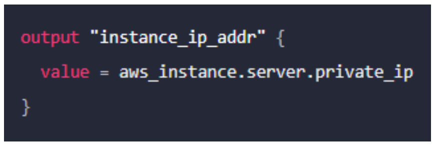
Variable Definitions (.tfvars) Files

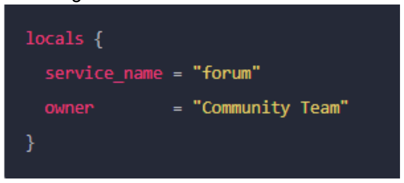
To set lots of variables, it is more convenient to specify their values in a *variable definitions file*

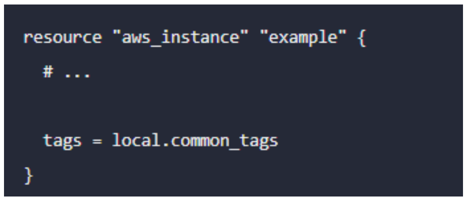


```
terraform apply -var-file="testing.tfvars"
```

	Environment Variables
	Terraform searches the environment of its own process for environment variables named TF_VAR_ followed by the name of a declared variable.
	
	`$ export TF_VAR_image_id=ami-abc123`
	`$ terraform plan`
	`...`
	Variable Definition Precedence:
	● Environment variables
	● Terraform.tfvars
	● Any var or var file option provided on command line.

Output variables	
	`output "instance_ip_addr" {`
	`  value = aws_instance.server.private_ip`
	`}`

Local Values	A local value assigns a name to an expression, so you can use the name multiple times within a module instead of repeating the expression.
	Declaring local value
	
	`locals {`
	`  service_name = "forum"`
	`  owner        = "Community Team"`
	`}`
	Using local value



```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

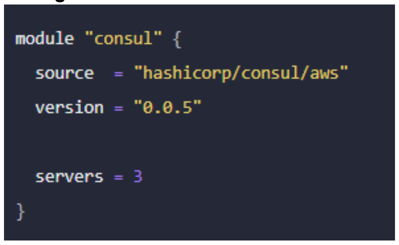
Modules

Modules are containers for multiple resources that are used together. A module consists of a collection of .tf and/or .tf.json files kept together in a directory.

The Root Module
Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory.

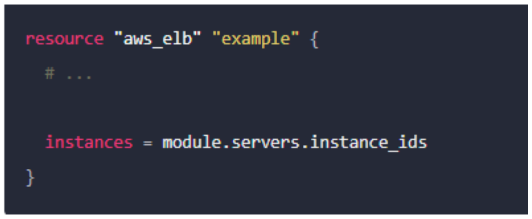
Child Modules
A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration.

Calling a Child Module



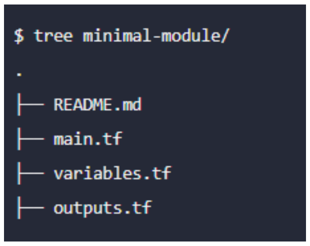
```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"

  servers = 3
}
```

Accessing Module Output Values



```
resource "aws_elb" "example" {
  # ...

  instances = module.servers.instance_ids
}
```

Module structure
Minimum structure:



```
$ tree minimal-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```

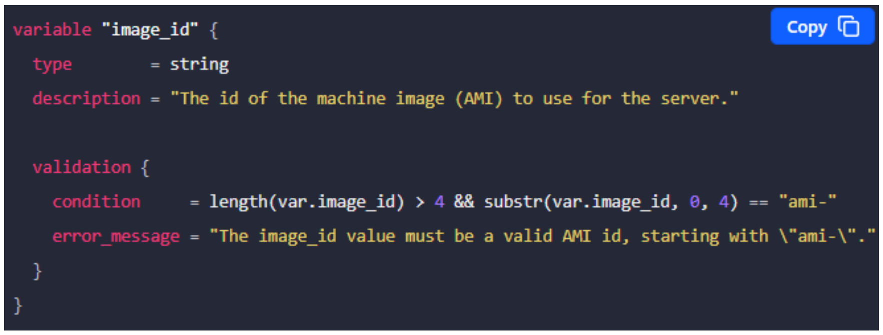
Complete

	`$ tree complete-module/`
	`.`
	`├── README.md`
	`├── main.tf`
	`├── variables.tf`
	`├── outputs.tf`
	`├── ...`
	`├── modules/`
	`│   ├── nestedA/`
	`│   │   ├── README.md`
	`│   │   ├── variables.tf`
	`│   │   ├── main.tf`
	`│   │   ├── outputs.tf`
	`│   ├── nestedB/`
	`│   ├── .../`
	`├── examples/`
	`│   ├── exampleA/`
	`│   │   ├── main.tf`
	`│   ├── exampleB/`
	`│   ├── .../`

Expressions	Conditional Expressions
	`condition ? true_val : false_val`
	for Expressions
	`[for s in var.list : upper(s)]`

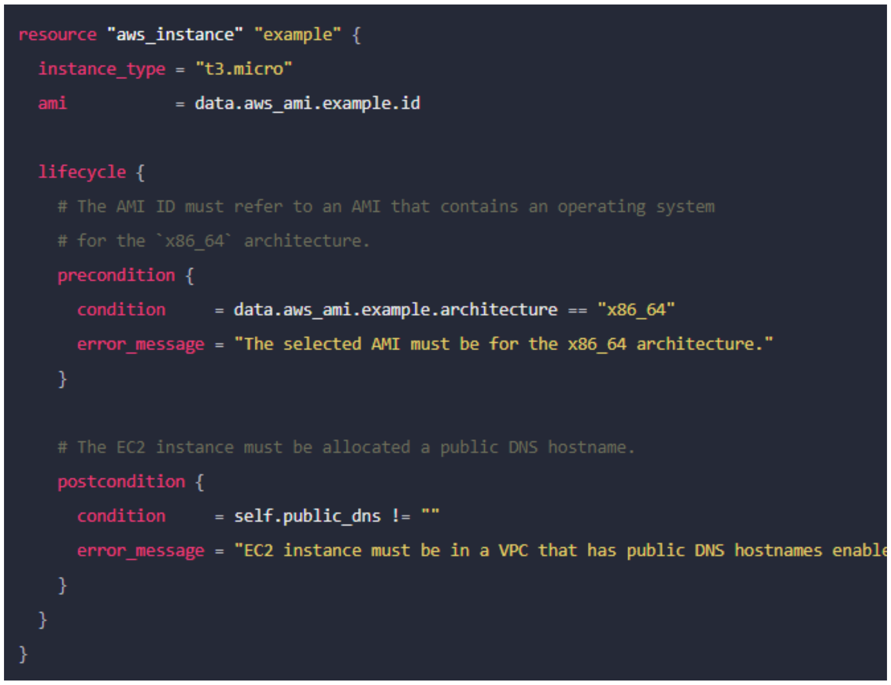
Custom Condition Checks

Input Variable Validation



```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."
  }
}
```

Preconditions and Postconditions



```
resource "aws_instance" "example" {
  instance_type = "t3.micro"
  ami           = data.aws_ami.example.id

  lifecycle {
    # The AMI ID must refer to an AMI that contains an operating system
    # for the `x86_64` architecture.
    precondition {
      condition     = data.aws_ami.example.architecture == "x86_64"
      error_message = "The selected AMI must be for the x86_64 architecture."
    }

    # The EC2 instance must be allocated a public DNS hostname.
    postcondition {
      condition     = self.public_dns != ""
      error_message = "EC2 instance must be in a VPC that has public DNS hostnames enable
    }
  }
}
```

Condition Expressions

Input variable validation, preconditions, and postconditions all require a condition argument. This is a boolean expression that should return true if the intended assumption or guarantee is fulfilled or false if it does not.

Logical Operators

Use the logical operators && (AND), || (OR), and ! (NOT) to combine multiple conditions together.

```
condition = var.name != "" && lower(var.name) == var.name
```

contains **Function**

```
condition = contains(["STAGE", "PROD"], var.environment)
```

length **Function**

Use the length function to test a collection's length and require a non-empty list or map.

```
condition = length(var.items) != 0
```

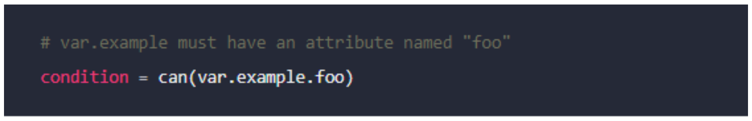
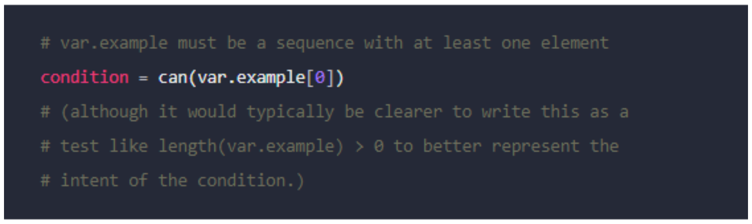
for **Expressions**

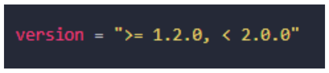
Use for expressions in conjunction with the functions alltrue and anytrue to test whether a condition holds for all or for any elements of a collection.

```
condition = alltrue([
  for v in var.instances : contains(["t2.micro", "m3.medium"], v.type)
])
```

can **Function**

Use the can function to concisely use the validity of an expression as a condition. It returns true if its given expression evaluates successfully and false if it returns any error, so you can use various other functions that typically return errors as a part of your condition expressions.

	 `# var.example must have an attribute named "foo"`
	`condition = can(var.example.foo)`
	 `# var.example must be a sequence with at least one element`
	`condition = can(var.example[0])`
	`# (although it would typically be clearer to write this as a`
	`# test like length(var.example) > 0 to better represent the`
	`# intent of the condition.)`

Version Constraints	Terraform's syntax for version constraints is very similar to the syntax used by other dependency management systems like Bundler and NPM.  `version = ">= 1.2.0, < 2.0.0"` The following operators are valid: ● = (or no operator): Allows only one exact version number. Cannot be combined with other conditions. ● !=: Excludes an exact version number. ● >, >=, <, <=: Comparisons against a specified version, allowing versions for which the comparison is true. "Greater-than" requests newer versions, and "less-than" requests older versions. ● ~>: Allows only the *rightmost* version component to increment. For example, to allow new patch releases within a specific minor release, use the full version number: ~> 1.0.4 will allow installation of 1.0.5 and 1.0.10 but not 1.1.0. This is usually called the pessimistic constraint operator.
Functions	format Function

```
> format("Hello, %s!", "Ander")
Hello, Ander!
> format("There are %d lights", 4)
There are 4 lights
```

join Function

```
join(separator, list)
```

split Function

```
split(separator, string)
```

lookup Function

```
lookup(map, key, default)
```

range Function

```
range(max)
range(start, limit)
range(start, limit, step)
```

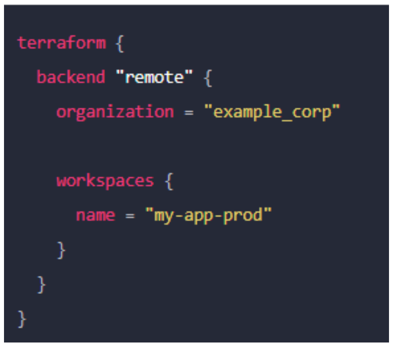
file **Function**

file reads the contents of a file at the given path and returns them as a string.

	`> file("${path.module}/hello.txt")`
	`Hello World`
	
	templatefile **Function**
	
	templatefile reads the file at the given path and renders its content as a template using a supplied set of template variables.
	
	`templatefile(path, vars)`
	
	filebase64 **Function**
	
	filebase64 reads the contents of a file at the given path and returns them as a base64-encoded string.

Terraform Settings	Terraform Block Syntax
	
	`terraform {`
	`  # ...`
	`}`
	
	Configuring Terraform Cloud
	
	Configuring a Terraform Backend
	
	The nested backend block configures which state backend Terraform should use.

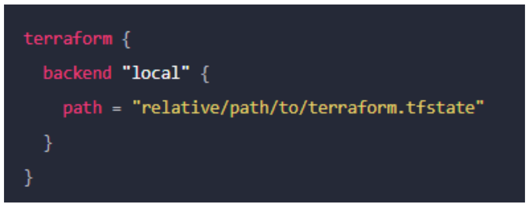
A backend defines where Terraform stores its state data files.



```
terraform {
  backend "remote" {
    organization = "example_corp"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

Available Backends

Local:



```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

Azurerm

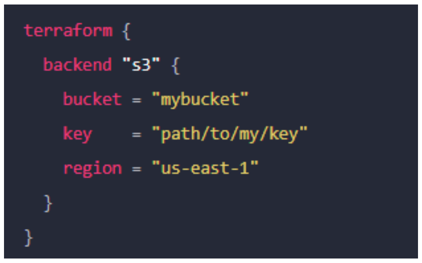
When authenticating using the Azure CLI or a Service Principal (either with a Client Certificate or a Client Secret):

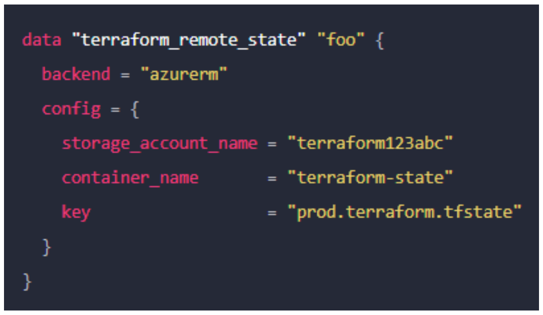
```
terraform {
  backend "azurerm" {
    resource_group_name  = "StorageAccount-ResourceGroup"
    storage_account_name = "abcd1234"
    container_name       = "tfstate"
    key                  = "prod.terraform.tfstate"
  }
}
```

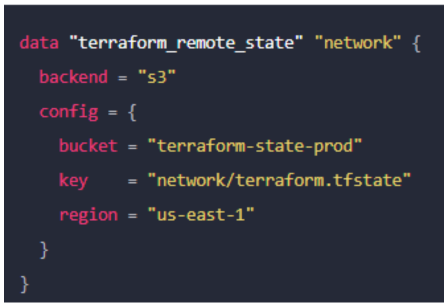
When authenticating using Azure AD Authentication:

```
terraform {
  backend "azurerm" {
    storage_account_name = "abcd1234"
    container_name       = "tfstate"
    key                  = "prod.terraform.tfstate"
    use_azuread_auth     = true
    subscription_id      = "00000000-0000-0000-0000-000000000000"
    tenant_id            = "00000000-0000-0000-0000-000000000000"
  }
}
```

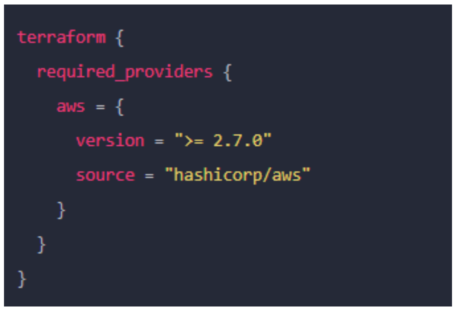
S3

	 `terraform {`
	`  backend "s3" {`
	`    bucket = "mybucket"`
	`    key    = "path/to/my/key"`
	`    region = "us-east-1"`
	`  }`
	`}`

Data Sources	*Remote_state*
	When authenticating using a Service Principal (either with a Client Certificate or a Client Secret):
	 `data "terraform_remote_state" "foo" {`
	`  backend = "azurerm"`
	`  config = {`
	`    storage_account_name = "terraform123abc"`
	`    container_name       = "terraform-state"`
	`    key                  = "prod.terraform.tfstate"`
	`  }`
	`}`
	
	S3

	
	```
	data "terraform_remote_state" "network" {
	  backend = "s3"
	  config = {
	    bucket = "terraform-state-prod"
	    key    = "network/terraform.tfstate"
	    region = "us-east-1"
	  }
	}
	```

Specifying a Required Terraform Version	The required_version setting accepts a version constraint string, which specifies which versions of Terraform can be used with your configuration. If the running version of Terraform doesn't match the constraints specified, Terraform will produce an error and exit without taking any further actions.

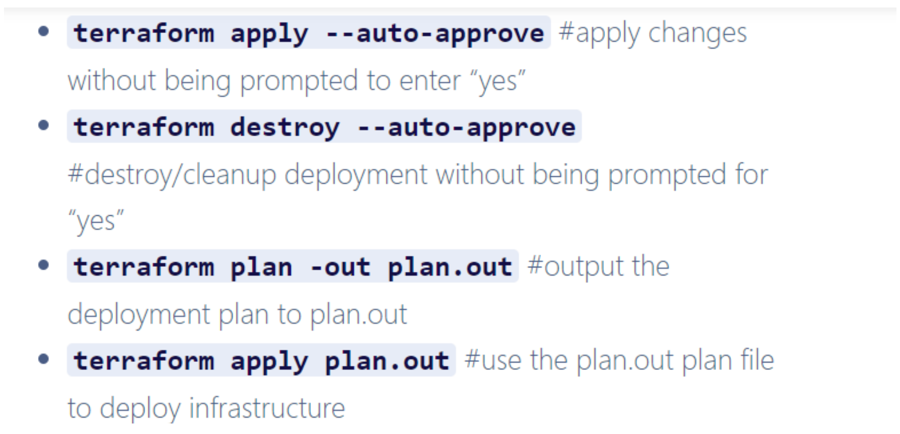
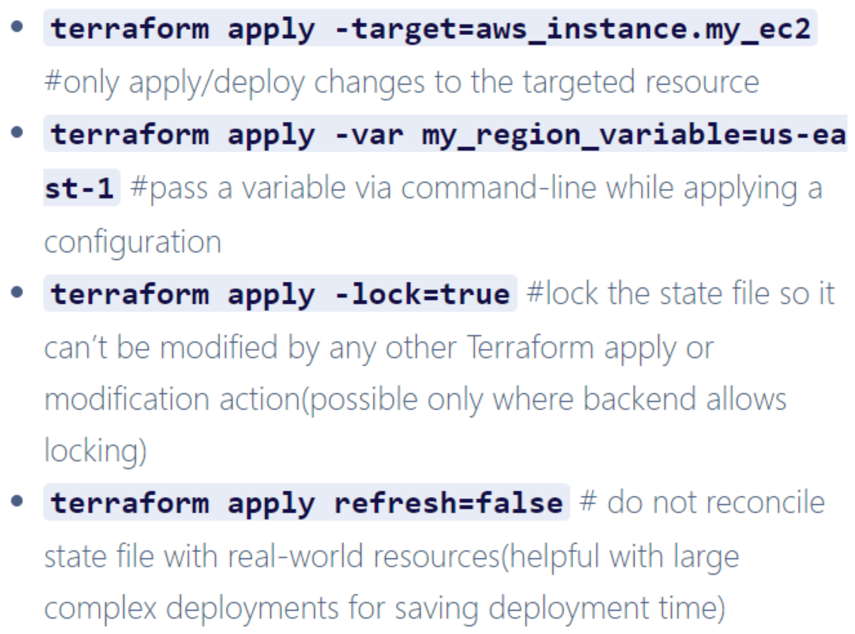
Specifying Provider Requirements	The required_providers block specifies all of the providers required by the current module, mapping each local provider name to a source address and a version constraint.
	
	```
	terraform {
	  required_providers {
	    aws = {
	      version = ">= 2.7.0"
	      source = "hashicorp/aws"
	    }
	  }
	}
	```

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.

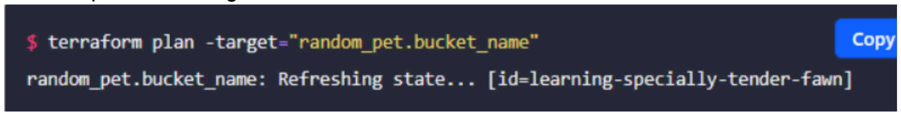
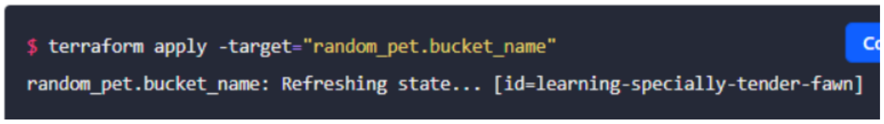
	Terraform uses this local state to create plans and make changes to your infrastructure. Prior to any operation, Terraform does a refresh to update the state with the real infrastructure. The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances 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. Inspection and Modification Add terraform state commands

Terraform CLI

Initialize Terraform Configuration	1. Terraform **downloads the modules** referenced in the configuration. 2. Terraform **initializes the backend**. 3. Terraform **downloads the providers** referenced in the configuration. 4. Terraform **creates a lock file**, which records the versions and hashes of the providers used in this run. • `terraform init` #initialize directory, pull down providers

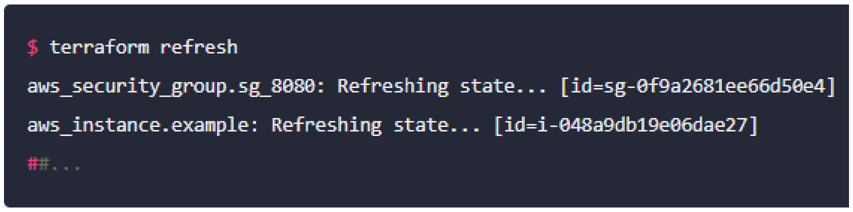
Plan, Deploy and Cleanup Infrastructure	 • `terraform apply --auto-approve` #apply changes without being prompted to enter "yes" • `terraform destroy --auto-approve` #destroy/cleanup deployment without being prompted for "yes" • `terraform plan -out plan.out` #output the deployment plan to plan.out • `terraform apply plan.out` #use the plan.out plan file to deploy infrastructure  • `terraform apply -target=aws_instance.my_ec2` #only apply/deploy changes to the targeted resource • `terraform apply -var my_region_variable=us-east-1` #pass a variable via command-line while applying a configuration • `terraform apply -lock=true` #lock the state file so it can't be modified by any other Terraform apply or modification action(possible only where backend allows locking) • `terraform apply refresh=false` # do not reconcile state file with real-world resources(helpful with large complex deployments for saving deployment time)

	• `terraform apply --parallelism=5` #number of simultaneous resource operations • `terraform refresh` #reconcile the state in Terraform state file with real-world resources

Target Resources	Create a plan which targets a resources:
	
	`$ terraform plan -target="random_pet.bucket_name"`
	`random_pet.bucket_name: Refreshing state... [id=learning-specially-tender-fawn]`
	Apply the change to only the bucket name.
	
	`$ terraform apply -target="random_pet.bucket_name"`
	`random_pet.bucket_name: Refreshing state... [id=learning-specially-tender-fawn]`

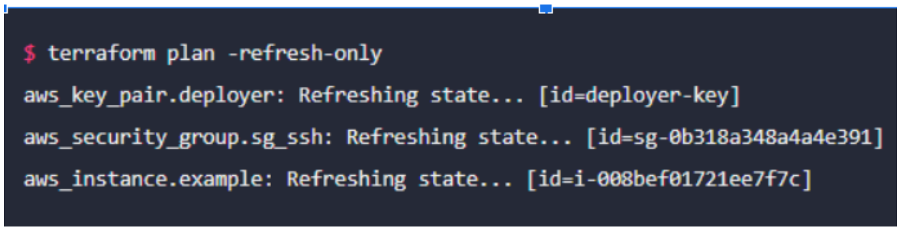
Terraform Workspaces	• `terraform workspace new mynewworkspace` #create a new workspace • `terraform workspace select default` #change to the selected workspace • `terraform workspace list` #list out all workspaces

Terraform State Manipulation	• `terraform state show aws_instance.my_ec2` #show details stored in Terraform state for the resource • `terraform state pull > terraform.tfstate` #download and output terraform state to a file • `terraform state mv aws_iam_role.my_ssm_role module.custom_module` #move a resource tracked via state to different module • `terraform state replace-provider hashicorp/aws registry.custom.com/aws` #replace an existing provider with another • `terraform state list` #list out all the resources tracked via the current state file • `terraform state rm  aws_instance.myinstace` #unmanage a resource, delete it from Terraform state file

The terraform refresh command updates the state file when physical resources change outside of the Terraform workflow.	
	`$ terraform refresh`
	`aws_security_group.sg_8080: Refreshing state... [id=sg-0f9a2681ee66d50e4]`
	`aws_instance.example: Refreshing state... [id=i-048a9db19e06dae27]`
	`##...`
	
	Run a refresh-only plan

By default, Terraform compares your state file to real infrastructure whenever you invoke terraform plan or terraform apply. The refresh updates your state file in-memory to reflect the actual configuration of your infrastructure. This ensures that Terraform determines the correct changes to make to your resources.

If you suspect that your infrastructure configuration changed outside of the Terraform workflow, you can use a -refresh-only flag to inspect what the changes to your state file would be.

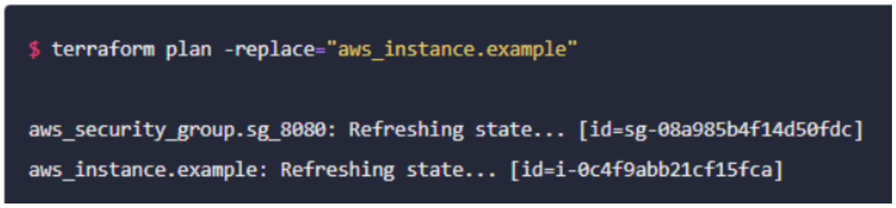


```
$ terraform plan -refresh-only
aws_key_pair.deployer: Refreshing state... [id=deployer-key]
aws_security_group.sg_ssh: Refreshing state... [id=sg-0b318a348a4a4e391]
aws_instance.example: Refreshing state... [id=i-008bef01721ee7f7c]
```

A refresh-only operation does not attempt to modify your infrastructure to match your Terraform configuration -- it only gives you the option to review and track the drift in your state file.

If you ran terraform plan or terraform apply without the -refresh-only flag now, Terraform would attempt to revert your manual changes. Instead, you will update your configuration to associate your EC2 instance with both security groups.

Terraform Import And Outputs	• `terraform import aws_instance.new_ec2_instance i-abcd1234` #import EC2 instance with id i-abcd1234 into the Terraform resource named “new_ec2_instance” of type “aws_instance” • `terraform import 'aws_instance.new_ec2_instance[0]' i-abcd1234` #same as above, imports a real-world resource into an instance of Terraform resource • `terraform output` #list all outputs as stated in code • `terraform output instance_public_ip` # list out a specific declared output

Terraform Taint/Untaint(mark/unmark resource for recreation -> delete and then recreate)	• `terraform taint aws_instance.my_ec2` #taints resource to be recreated on next apply
	• `terraform untaint aws_instance.my_ec2` #Remove taint from a resource
	• `terraform force-unlock LOCK_ID` #forcefully unlock a locked state file, LOCK_ID provided when locking the State file beforehand
	
	Run terraform plan -replace="aws_instance.example" to see the actions Terraform would take if you replaced the instance.
	
	`$ terraform plan -replace="aws_instance.example"`
	
	`aws_security_group.sg_8080: Refreshing state... [id=sg-08a985b4f14d50fdc]`
	`aws_instance.example: Refreshing state... [id=i-0c4f9abb21cf15fca]`
