Daniella Lee – Home Office Preparation Notes

DevOps Tools – Terraform

Table of Contents

[Introduction 2](#_Toc498338476)

[What is Terraform? 2](#_Toc498338477)

[Capabilities 2](#_Toc498338478)

[Key features 2](#_Toc498338479)

[Use cases 3](#_Toc498338480)

[Installation 4](#_Toc498338481)

[Prerequisites 4](#_Toc498338482)

[Commands for Terraform installation on an Amazon Linux instance 6](#_Toc498338483)

[Installing your chosen version 6](#_Toc498338484)

[To carry out updates: 6](#_Toc498338485)

[Configuration basics 7](#_Toc498338486)

[Providers 8](#_Toc498338487)

[Variables 9](#_Toc498338488)

[HashiCorp Configuration Language syntax 9](#_Toc498338489)

[Building infrastructure – Using the AWS provider 10](#_Toc498338490)

[Adapting a hard-coded example 10](#_Toc498338491)

[Parameterising access keys and AMIs 11](#_Toc498338492)

[Example configuration file – Launching an EC2 instance 12](#_Toc498338493)

[Build commands 13](#_Toc498338494)

[Changing infrastructure 14](#_Toc498338495)

[Destroying infrastructure 16](#_Toc498338496)

[Building infrastructure – Provisioning 17](#_Toc498338497)

[Running commands on your Terraform machine (using local-exec) 18](#_Toc498338498)

[Running commands on a created instance (using remote-exec) 18](#_Toc498338499)

[Troubleshooting communication during provisioning 19](#_Toc498338500)

[Key pairs 19](#_Toc498338501)

[Security Groups 20](#_Toc498338502)

[Terraform vs Configuration Management 21](#_Toc498338503)

[Example configuration file – Running commands on a new instance 21](#_Toc498338504)

[Example configuration file – Installing docker on your EC2 instance 22](#_Toc498338505)

# Introduction

## What is Terraform?

Terraform is an open source tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned.

Terraform enables you to safely and predictably create, change, and improve production infrastructure.

WRITE 🡪 INFRASTRUCTURE AS CODE

PLAN 🡪 PREVIEW CHANGES BEFORE APPLYING

CREATE 🡪 REPRODUCIBLE INFRASTRUCTURE

These notes are adapted from the Terraform documentation, which can be found here:

<https://www.terraform.io/intro/index.html>

### Capabilities

MAP RESOURCE DEPENDENCIES

Understand how a minor change could have potential cascading effects across an infrastructure before executing that change.

SEPARATION OF PLAN & APPLY

Separating plans and applies reduces mistakes and uncertainty at scale.

ONE SAFE WORKFLOW ACROSS PROVIDERS

Safely and predictably make changes to infrastructure. Supports AWS, GCP, Azure etc..

### Key features

**Infrastructure as Code**

Infrastructure is described using a high-level configuration syntax. Infrastructure can be shared and re-used, and versioned and treated as you would any other code

**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.

**Resource Graph**

Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources. This gives insight into dependencies in your infrastructure.

**Change Automation**

Complex sets of changes can be applied to your infrastructure with minimal human interaction. With the execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

## Use cases

Terraform has many use cases, which are detailed here:

<https://www.terraform.io/intro/use-cases.html>

These use cases include:

**Multi-Tier Applications**

Terraform is an ideal tool for building and managing these infrastructures. Each tier can be described as a collection of resources, and the dependencies between each tier are handled automatically. Each tier can then be scaled easily using Terraform by modifying a single count configuration value.

**Self-Service Clusters**

Using Terraform, the knowledge of how to build and scale a service can be codified in a configuration. Terraform configurations can be shared within an organization enabling customer teams to use the configuration as a black box and use Terraform as a tool to manage their services.

**Software Demos**

Terraform can aid users to demo software on real infrastructure which more closely matches production environments. Software writers can provide a Terraform configuration to create, provision and bootstrap a demo on cloud providers like AWS.

**Disposable Environments**

Using Terraform, the production environment can be codified and then shared with staging, QA or dev. Terraform can help tame the difficulty of maintaining parallel environments, and makes it practical to elastically create and destroy them.

**Software Defined Networking**

Terraform can be used to codify the configuration for software defined networks. This configuration can then be used by Terraform to automatically setup and modify settings by interfacing with the control layer. This allows configuration to be versioned and changes to be automated. As an example, **AWS VPC** is one of the most commonly used SDN implementations, and can be configured by Terraform.

**Resource Schedulers**

Terraform is not limited to physical providers like AWS. Resource schedulers (e.g. Borg, Mesos, YARN, and Kubernetes) can be treated as a provider, enabling Terraform to request resources from them. This allows Terraform to be used in layers: to setup the physical infrastructure running the schedulers as well as provisioning onto the scheduled grid.

**Multi-Cloud Deployment**

Having a multi-cloud deployment allows for more graceful recovery of the loss of a region or entire provider.

Terraform is cloud-agnostic and allows a single configuration to be used to manage multiple providers, and to even handle cross-cloud dependencies. This simplifies management and orchestration, helping operators build large-scale multi-cloud infrastructures.

# Installation

## Prerequisites

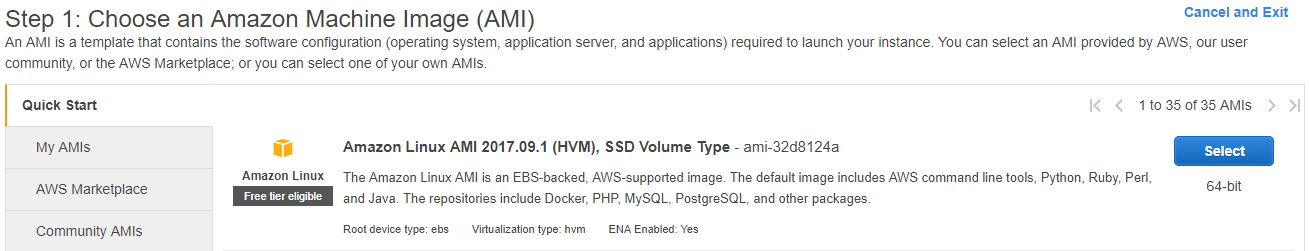
**Amazon Web Services**

**Terraform** can be used to manage **Amazon Web Services** resources. We will be using the AWS provider to create and destroy free tier resources such as **EC2** micro instances.

This guide will take you through installing Terraform on a controlling EC2 instance. From this instance, we will create and provision other EC2 instances

**Amazon Linux AMI 2017.09.1**

When you create your instances, choose the **Amazon Linux** AMI.



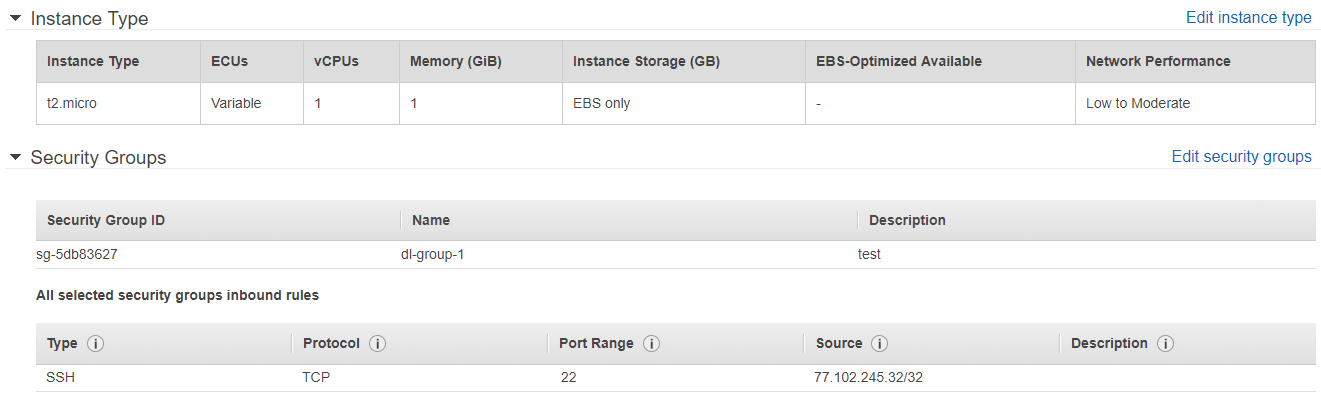
If you wish to use an **Ubuntu** instance, or any other OS type, the Terraform installation instructions would vary, as might some of the other steps later in this guide.

See here for installation instructions for your chosen OS:

<https://www.terraform.io/intro/getting-started/install.html>

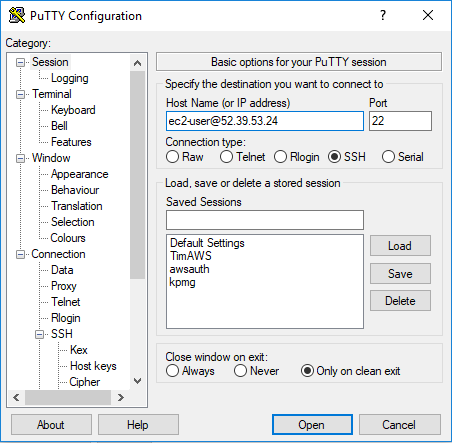
Other Linux distributions might use a different **package manager** than the Amazon Linux instance does. For example, to adapt some of the commands in this guide for use on Ubuntu instances, at the very least you will have to swap out **yum** for **apt-get**.

To **SSH** into your instance, you will require the **private key** (.pem) that you associated with your instance when you created it:

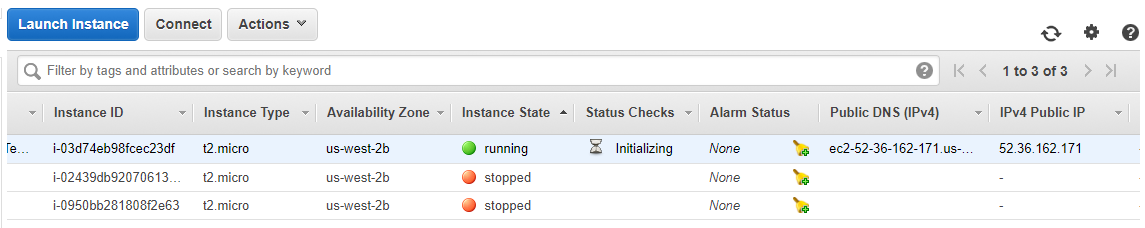


To SSH via a tool such as **puTTY**, you will need to produce a .**ppk** key (e.g. via **puTTYgen**)

When you configure your session, browse for your key under Connection🡪SSH🡪Auth

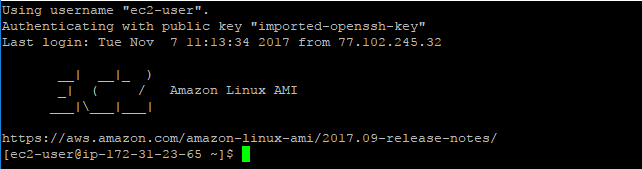


You can obtain your **IP address** from the **EC2 Management Console**.



You will need to login using the default username for your AMI (Ubuntu uses **root**).

Successful login should resemble this:



## Commands for Terraform installation on an Amazon Linux instance

Terraform releases and downloads can be found here:

<https://www.terraform.io/downloads.html>

### Installing your chosen version

Verify that **install** and **unzip** are present. If not, install them now:

**sudo yum install -y zip unzip**

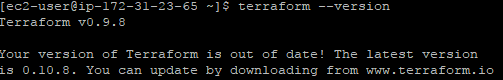
Retrieve your package, unzip it, and move the **terraform** binary to the directory /usr/local/bin

**wget https://releases.hashicorp.com/terraform/0.9.8/terraform\_0.9.8\_linux\_amd64.zip**

**unzip terraform\_0.9.8\_linux\_amd64.zip**

**sudo mv terraform /usr/local/bin/**

Confirm that the terraform binary is accessible using **terraform –version**



### To carry out updates:

Remove the old Terraform binary from /usr/local/bin

**Sudo rm /usr/local/bin/terraform**

Download the new release, unzip the file, and move the binary to /usr/local/bin

**wget https://releases.hashicorp.com/terraform/0.10.8/terraform\_0.10.8\_linux\_amd64.zip**

**unzip terraform\_0.10.8\_linux\_amd64.zip**

**sudo mv terraform /usr/local/bin/**

Confirm the new version:



This may help you troubleshoot any path issues, should you encounter them

<https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux-unix>

# Configuration basics

Terraform configuration process

Terraform uses text files to describe infrastructure and to set variables.

Configuration files can take two formats: Terraform format and **JSON**.

Terraform format ends in **.tf**and JSON format ends in **.tf.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, but can also be used by Terraform operators if you prefer.

When invoking any command that loads the Terraform configuration, Terraform loads all configuration files within the directory specified in alphabetical order. Override files are the exception, as they're loaded after all non-override files, in alphabetical order.

Terraform configurations are declarative, so references to other resources and variables do not depend on the order they're defined.

Terraform loads all configuration files within a directory and appends them together. This means that two resources with the same name are not merged, and will instead cause a validation error.

This is in contrast to overrides, which do merge.

Overrides

Terraform also has a concept of overrides, a way to create files that are loaded last and merged into your configuration, rather than appended.

**Overrides have a few use cases:**

Machines (tools) can create overrides to modify Terraform behaviour without having to edit the Terraform configuration tailored to human readability. Temporary modifications can be made to Terraform configurations without having to modify the configuration itself.

Overrides names must be **override** or end in **\_override**, excluding the extension.

Examples of valid override files are **override.tf**, **override.tf.json**, **temp\_override.tf**.

Override files are loaded last in alphabetical order.

Override files can be in Terraform syntax or JSON, just like non-override Terraform configurations.

Resources

Resources are a component of your infrastructure. Resources can be high or low-level.

A low-level component is one such as a physical server, virtual machine, or container.

A higher-level component might be an email provider, DNS record, or database provider.

A resource configuration looks like the following:

resource "aws\_instance" "web" {

ami **=** "ami-408c7f28"

instance\_type **=** "t1.micro"

}

## Providers

Providers are responsible in Terraform for managing the lifecycle of a **resource**: create, read, update, delete.

Most providers require some sort of configuration to provide authentication information, endpoint URLs, etc. Provider configuration blocks are a way to set this information globally for all matching resources.

A provider configuration looks like the following:

provider "aws" {

access\_key **=** "foo"

secret\_key **=** "bar"

region **=** "us-east-1"

}

Every resource in Terraform is mapped to a provider based on longest-prefix matching.

For example, the **aws\_instance** resource type would map to the AWS provider.

Documentation for the AWS provider can be found here:

<https://www.terraform.io/docs/providers/aws/>

More on providers:

<https://www.terraform.io/docs/providers/index.html>

## Variables

Variables define the parameterization of Terraform configurations.

Variables can be overridden via the CLI.

A variable configuration looks like the following:

variable "key" {

type **=** "string"

}

variable "images" {

type **=** "map"

## HashiCorp Configuration Language syntax

The syntax of Terraform configurations is called HashiCorp Configuration Language (HCL).

It is meant to strike a balance between human readable and editable as well as being machine-friendly.

Basic bullet point reference:

* Single line comments start with #. Multi-line comments are wrapped with /\* and \*/
* Values are assigned with the syntax of key = value (whitespace doesn't matter). The value can be any primitive (string, number, boolean), a list, or a map.
* Strings are in double-quotes.
* Strings can interpolate other values using syntax wrapped in ${}, such as ${var.foo}.
* Multiline strings can use shell-style "here doc" syntax, with the string starting with a marker like <<EOF and then the string ending with EOF on a line of its own. The lines of the string and the end marker must not be indented.
* Numbers are assumed to be base 10. If you prefix a number with 0x, it is treated as a hexadecimal number.
* Boolean values: true, false.
* Lists of primitive types can be made with square brackets ([]). Example: ["foo", "bar", "baz"].
* Maps can be made with braces ({}) and colons (:): { "foo": "bar", "bar": "baz" }.
* Quotes may be omitted on keys, unless the key starts with a number, in which case quotes are required.
* Commas are required between key/value pairs for single line maps. A newline between key/value pairs is sufficient in multi-line maps.

Supporting documentation can be found here:

<https://www.terraform.io/docs/configuration/syntax.html>

# Building infrastructure – Using the AWS provider

## Adapting a hard-coded example

A basic example is taken from:

<https://www.terraform.io/intro/getting-started/build.html>

The following code will launch an EC2 instance in the region US-East-1.

Create a file called **example.tf** with the following code

provider "aws" {

access\_key **=** "ACCESS\_KEY\_HERE"

secret\_key **=** "SECRET\_KEY\_HERE"

region **=** "us-east-1"

}

resource "aws\_instance" "example" {

ami **=** "ami-2757f631"

instance\_type **=** "t2.micro"

}

For Terraform configuration to become truly shareable and version controlled, we need to

remove the hard-coded secrets and ami ID.

We will instead be listing our variables in a file such as this:

variable "access\_key" {}

variable "secret\_key" {}

variable "region" {

default **=** "us-west-2"

}

variable "amis" {

type **=** "map"

default **=** {

"us-east-1" **=** "ami-b374d5a5"

"us-west-2" **=** "ami-4b32be2b"

}

}

We will then create the instance via file such as this:

provider "aws" {

access\_key **=** "${var.access\_key}"

secret\_key **=** "${var.secret\_key}"

region **=** "${var.region}"

}

resource "aws\_instance" "example" {

ami **=** "${lookup(var.amis, var.region)}"

instance\_type **=** "t2.micro"

}

### Parameterising access keys and AMIs

We can use input variables to parameterize the configurations for access keys, AMIs, etc.

Create a file **variables.tf**with the following contents.

variable "access\_key" {}

variable "secret\_key" {}

variable "region" {

default **=** "us-east-1"

}

This defines three variables within your Terraform configuration.

The first two have empty blocks {}. The third sets a default.

If a default value is set, the variable is optional. Otherwise, the variable is required.

If you run **terraform plan** now, Terraform will prompt you for the values for the unset string variables **access\_key** and **secret\_key**

Next, replace the AWS provider configuration in the file **example.tf** with the following:

provider "aws" {

access\_key **=** "${var.access\_key}"

secret\_key **=** "${var.secret\_key}"

region **=** "${var.region}"

}

This uses more interpolations, this time prefixed with var. This tells Terraform that you're accessing variables. This configures the AWS provider with the given variables.

AMIs are specific to the region that is in use. We have used the region US-East-1 as default.

Instead of asking the user to input the proper AMI for the region, we can use **maps**.

Maps are a way to create variables that are lookup tables. In the file **variables.tf** , let's extract our AMIs into a map and add support for the **us-west-2** region as well:

variable "amis" {

type **=** "map"

default **=** {

"us-east-1" **=** "ami-b374d5a5"

"us-west-2" **=** "ami-4b32be2b"

}

}

A variable can have a **map** type assigned explicitly, or it can be implicitly declared as a map by specifying a default value that is a map. The above demonstrates both.

Then, replace the **aws\_instance** block with the following:

resource "aws\_instance" "example" {

ami **=** "${lookup(var.amis, var.region)}"

instance\_type **=** "t2.micro"

}

This introduces a new type of interpolation: a function call.

The **lookup** function does a dynamic lookup in a map for a key.

The key is **var.region**, which specifies that the value of the region variables is the key.

While we don't use it in our example, it is worth noting that you can also do a static lookup of a map directly with **${var.amis["us-east-1"]}**.

More on variables:

<https://www.terraform.io/intro/getting-started/variables.html>

## Example configuration file – Launching an EC2 instance

Save the following to a file named **variables.tf**

variable "access\_key" {}

variable "secret\_key" {}

variable "region" {

default **=** "us-west-2"

}

variable "amis" {

type **=** "map"

default **=** {

"us-east-1" **=** "ami-b374d5a5"

"us-west-2" **=** "ami-4b32be2b"

}

}

Save the following to a file named **example.tf.**

provider "aws" {

access\_key **=** "${var.access\_key}"

secret\_key **=** "${var.secret\_key}"

region **=** "${var.region}"

}

resource "aws\_instance" "example" {

ami **=** "${lookup(var.amis, var.region)}"

instance\_type **=** "t2.micro"

}

We have defined all required attributes, and other optional ones are available e.g. assigning a key pair or a security group to the created instance

In our **variables.tf** file, we could prompt the user for a name to tag their instance with

variable "access\_key" {}

variable "secret\_key" {}

**variable "name" {}**

variable "region" {

default **=** "us-west-2"

}

variable "amis" {

type **=** "map"

default **=** {

"us-east-1" **=** "ami-6057e21a"

"us-west-2" **=** "ami-32d8124a"

}

}

In our **example.tf** file, we could then include tags in the resource block

provider "aws" {

access\_key **=** "${var.access\_key}"

secret\_key **=** "${var.secret\_key}"

region **=** "${var.region}"

}

resource "aws\_instance" "example" {

ami **=** "${lookup(var.amis, var.region)}"

instance\_type **=** "t2.micro"

**tags {**

**Name = "${var.name}"**

**}**

}

Details may be found here:

<https://www.terraform.io/docs/providers/aws/d/instance.html>

## Build commands

terraform init

For a new configuration or after checking out an existing configuration from version control. This initializes various local settings and data that will be used by subsequent commands.

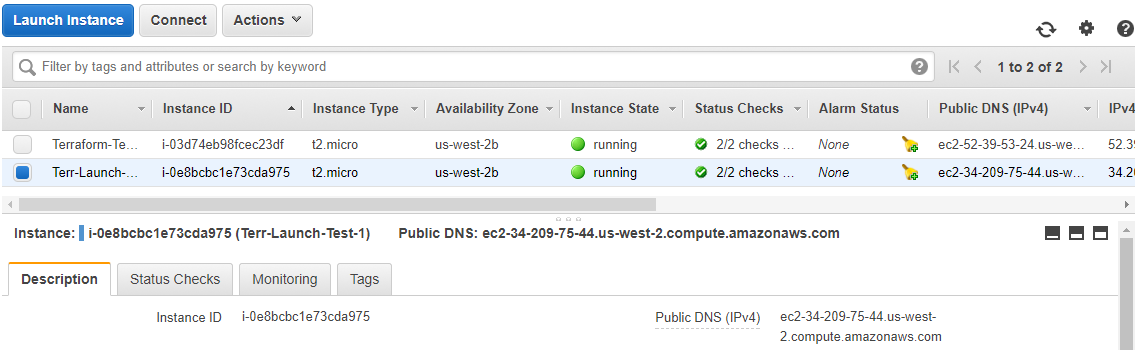
terraform plan

This displays any changes that are required for your infrastructure, it shows us what Terraform would do if we asked it to apply this configuration.

If the plan looks good, and the configuration appears valid, create real resources using terraform apply.

You will be prompted for your access key/secret key in the command line.

Once configuration is complete, navigate to the console to view your new instance



View the current state using terraform show.

$ terraform show

aws\_instance.example:

id = i-32cf65a8

ami = ami-2757f631

availability\_zone = us-east-1a

instance\_state = running

instance\_type = t2.micro

private\_ip = 172.31.30.244

public\_dns = ec2-52-90-212-55.compute-1.amazonaws.com

public\_ip = 52.90.212.55

subnet\_id = subnet-1497024d

vpc\_security\_group\_ids.# = 1

vpc\_security\_group\_ids.3348721628 = sg-67652003

## Changing infrastructure

We can add another ec2 instance without troubling any running ones by adding a new block:

resource "aws\_instance" "another" {

ami **=** "ami-b374d5a5"

instance\_type **=** "t2.micro"

}

This new instance does not depend on any other resource, and so can be created in parallel with the other resources.

If we were to make a change to an existing resource, e.g. to change the AMI of a running instance, Terraform would carry out the change by completely removing the resource and creating a new one.

Terraform configurations are meant to be changed like this. You can also completely remove resources from your file and Terraform will know to destroy the old one.

Create a file **ec2.tf** containing the following:

provider "aws" {

access\_key **=** "ACCESS\_KEY\_HERE"

secret\_key **=** "SECRET\_KEY\_HERE"

region **=** "us-east-1"

}

resource "aws\_instance" "example" {

ami **=** "ami-2757f631"

instance\_type **=** "t2.micro"

}

Initiate, plan and apply your configuration using **terraform init 🡪 plan 🡪 apply**

Wait for your instance to be created.

While it is running, we will change its AMI from Ubuntu 16.04 LTS to being Ubuntu 16.10.

Edit the aws\_instance.example resource in **ec2.tf** and change it to the following:

resource "aws\_instance" "example" {

ami **=** "ami-b374d5a5"

instance\_type **=** "t2.micro"

}

Use **terraform plan** to view the changes that will be made

$ terraform plan

# ...

-/+ aws\_instance.example

ami: "ami-2757f631" => "ami-b374d5a5" (forces new resource)

availability\_zone: "us-east-1a" => "<computed>"

[...]

The prefix "-/+" means that Terraform will destroy and recreate the resource, versus purely updating it in-place.

The plan output also shows that the AMI change is what necessitated the creation of a new resource. Using this information, you can tweak your changes to possibly avoid destroy/create updates if you didn't want to do them at this time.

Apply your changes with **terraform apply**

You can use **terraform show** to see the new properties associated with the instance.

## Destroying infrastructure

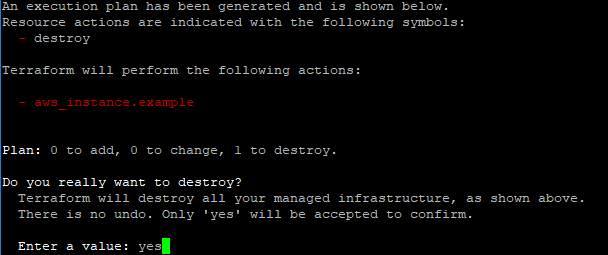
If you're using Terraform to spin up multiple environments such as development, test, QA environments, then destroying a resource is a useful action.

Use the command **terraform plan -destroy** to verify exactly what resources Terraform is managing and will destroy.

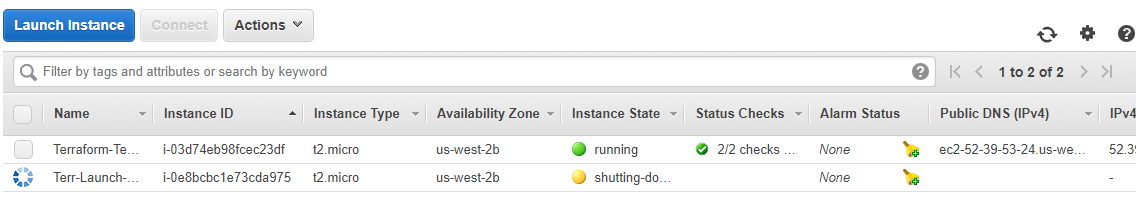
If satisfied with the output, you would then run **terraform destroy**

You will be asked to enter your access key, secret key (and instance tag name if your variables file requests this)

You will then be asked to verify that you really want to destroy the infrastructure:



The console should show the instance is now in the process of being terminated:



# Building infrastructure – Provisioning

This following section will use the following **variables.tf** file contents:

variable "access\_key" {}

variable "secret\_key" {}

variable "region" {

default **=** "us-west-2"

}

variable "amis" {

type **=** "map"

default **=** {

"us-east-1" **=** "ami-6057e21a"

"us-west-2" **=** "ami-32d8124a"

}

}

We can write configuration in the file example.tf that will spin up and provision various ec2 instances. We can use provisioners to do some basic bootstrapping of our instance.

Previously our code has resembled:

resource "aws\_instance" "example" {

ami **=** "ami-b374d5a5"

instance\_type **=** "t2.micro"

}

We will now add a **provisioner** block within the resource block such as:

provisioner "local-exec" {

command **=** "echo ${aws\_instance.example.public\_ip} > ip\_address.txt"

}

Terraform supports multiple provisioners.

Multiple **provisioner** blocks can be added to define multiple provisioning steps.

## Running commands on your Terraform machine (using local-exec)

Modify the **aws\_instance** resource as shown below to include a provisioner block:

resource "aws\_instance" "example" {

ami **=** "ami-b374d5a5"

instance\_type **=** "t2.micro"

provisioner "local-exec" {

command **=** "echo ${aws\_instance.example.public\_ip} > ip\_address.txt"

}

}

The **local-exec** provisioner executes a command locally on the machine running Terraform. In order to do this, no connection needs to be specified.

## Running commands on a created instance (using remote-exec)

The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc. The remote-exec provisioner supports both ssh and winrm type connections.

Now update your **aws\_instance** resource with the text below.

Include the name of a key to associate with your new instance.

Place this key in the same directory as your configuration files.

resource "aws\_instance" "example" {

ami **=** "ami-b374d5a5"

instance\_type **=** "t2.micro"

vpc\_security\_group\_ids = [

"sg-5db83627"

]

key\_name = "filename"

provisioner "remote-exec" {

inline **=** [

"echo hello this is remote > remote1.txt",

]

}

}

Initialise Terraform **terraform init**

View the proposed changes **terraform plan**

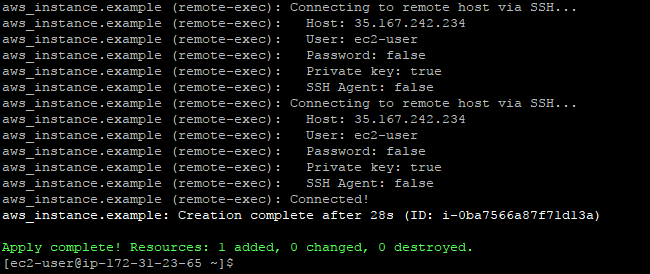
Run your configuration **terraform apply**

## Troubleshooting communication during provisioning

The remote-exec provisioner supports both ssh and winrm type connections.

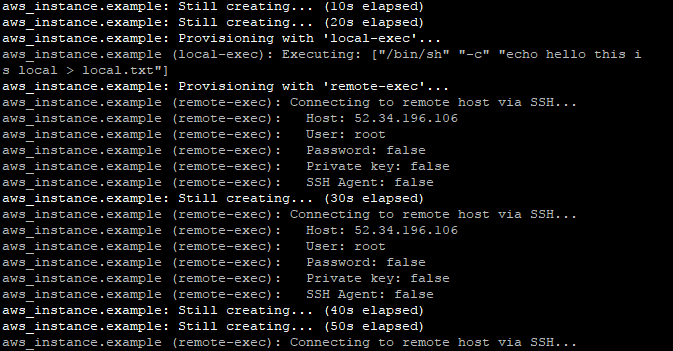
This guide deals with provisioning conducted using SSH. SSH requires TCP port 22.

Successful SSH connection should resemble:



### Key pairs

You may encounter the following error, where instance creation is taking too long/terraform instance cannot SSH into the newly created one, and the process times out.



The default connection variables shown above are used unless you supply others in your configuration file.

Terraform must also have access to the private key that you define under **connection**. This key should match the key that you assigned in the provider block for your ec2 instance.

Keys can be transferred to your Terraform instance from your desktop using scp.

By default, Terraform is trying to access our account over SSH, but is using the wrong **User**.

Declare the desired connection type as **SSH**, and now provide Terraform with the username and the path to the key file that will be required for SSH connection:

resource "aws\_instance" "example" {

ami **=** "ami-b374d5a5"

instance\_type **=** "t2.micro"

vpc\_security\_group\_ids = [

"sg-5db83627"

]

key\_name = "file name"

provisioner "remote-exec" {

inline **=** [

"echo hello this is remote > remote1.txt",

]

}

connection {

type = "ssh"

user = "ec2-user"

**private\_key = "${file("filename.pem")}"**

timeout = "2m"

agent = false

}

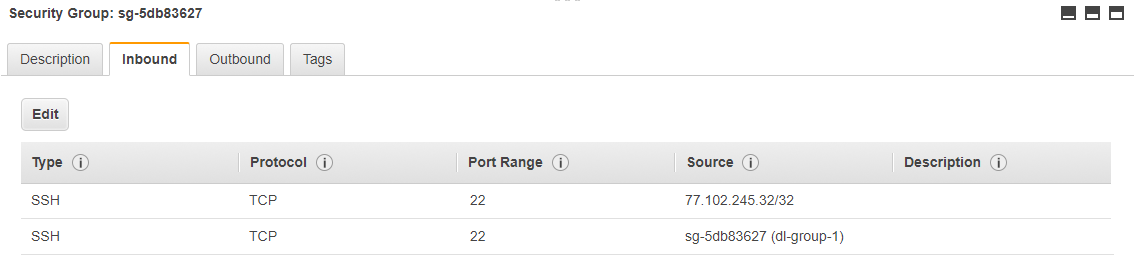
}

Try again to apply your configuration.

### Security Groups

If you encounter IP/port access issues, check the Inbound/Outbound Rules that have been configured for your security group.

From the EC2 management console, we can specify IPs that can access a security group. You should configure your chosen group to allow access from your public IP so that you can reach your instances using SSH.



Ensure you have an **Inbound** rule that allows access on **port 22** from all IPs in the same security group. This will allow all IPs in this group to communicate with each other via SSH.

Add a temporary IP rule to allow all IP access to your instance.

Otherwise, each time you startup your main ec2 instance, ensure that its IP is added to accepted inbound IPs for your security group.

Reboot your main instance, and try again to apply your configuration.

## Terraform vs Configuration Management

If you need to do some initial setup on your instances, then provisioners let you upload files, run shell scripts, or install and trigger other software like configuration management tools, etc.

Provisioners are only run when a resource is created. They are not a replacement for configuration management and changing the software of an already-running server, and are instead just meant as a way to bootstrap a server.

For configuration management, you should use Terraform provisioning to invoke a real configuration management solution.

## Example configuration file – Running commands on a new instance

Simple example 🡪 running a command to create a text file on the new instance

Copy this code into **example.tf**, and include your variables in another file e.g. **variables.tf**

provider "aws" {

access\_key **=** "${var.access\_key}"

secret\_key **=** "${var.secret\_key}"

region **=** "${var.region}"

}

resource "aws\_instance" "keyexample" {

ami **=** "${lookup(var.amis, var.region)}"

instance\_type **=** "t2.micro"

vpc\_security\_group\_ids = [

"sg-5db83627"

]

key\_name = "file name"

provisioner "remote-exec" {

inline **=** [

"echo hello this is remote > remote1.txt",

]

}

connection {

type = "ssh"

user = "ec2-user"

**private\_key = "${file("filename.pem")}"**

timeout = "2m"

agent = false

}

}

## Example configuration file – Installing docker on your EC2 instance

The below code will install Docker on your new instance and start the service.

Copy this code into **example.tf**, and include your variables in another file e.g. **variables.tf**

provider "aws" {

access\_key **=** "${var.access\_key}"

secret\_key **=** "${var.secret\_key}"

region **=** "${var.region}"

}

resource "aws\_instance" "withdocker" {

ami **=** "${lookup(var.amis, var.region)}"

instance\_type **=** "t2.micro"

vpc\_security\_group\_ids = [

"sg-5db83627"

]

key\_name = "filename"

provisioner "remote-exec" {

inline **=** [

"wget -qO- https://get.docker.com/ | sh",

"sudo service docker start",

"sudo usermod -aG docker ec2-user",

]

}

connection {

type = "ssh"

user = "ec2-user"

**private\_key = "${file("filename.pem")}"**

timeout = "10m"

agent = false

}

}

Depending on your instance type, your Docker installation commands may vary.

E.g. the below code will install docker on an Amazon Linux instance.

provisioner "remote-exec" {

inline **=** [

"sudo yum install -y docker",

"sudo service docker start",

"sudo usermod -a -G docker ec2-user",

]

}

See here for more information on AWS and Docker.

<http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html>

Miscellaneous/Issues

Example **variables.tf** file

variable "access\_key" {}

variable "secret\_key" {}

variable "region" {

default **=** "us-west-2"

}

variable "amis" {

type **=** "map"

default **=** {

"us-east-1" **=** "ami-6057e21a"

"us-west-2" **=** "ami-32d8124a"

}

}

Correct syntax?

private\_key = "filename.pem"

private\_key = "${file(var.~/filename.pem)}"

scp help

<http://www.hypexr.org/linux_scp_help.php>

Online resources to troubleshoot your installation e.g.

<https://linuxacademy.com/community/posts/show/topic/18181-can-somebody-explain-how-to-install-terraform>

path

Guide to terraform configuration

<https://www.terraform.io/docs/configuration/index.html>