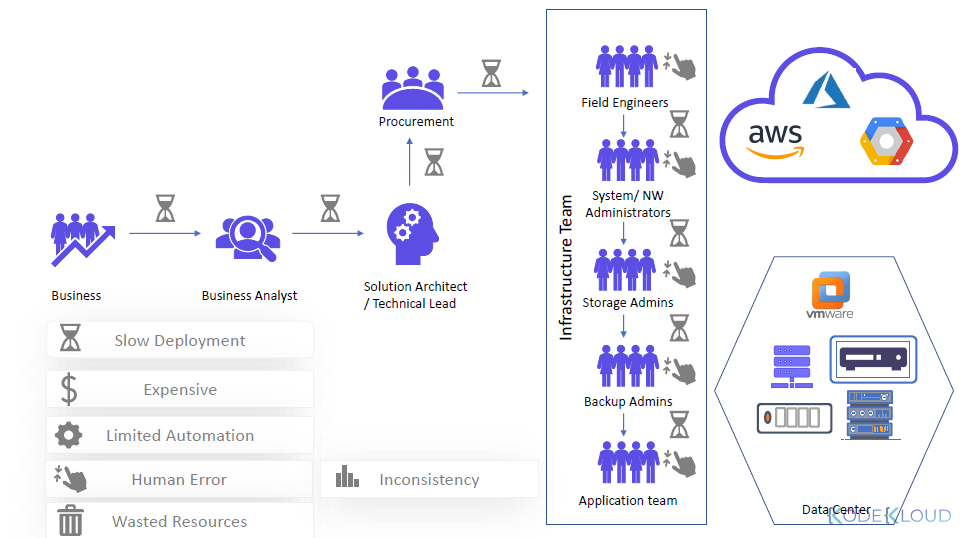
**Terraform**



what is terraform ?

Terraform allows you to automate and manage your infrastructure and your platform and services that run on that infrastructure.

it's open source

and it uses the declarative language meaning you don't have to define every step of how these automation and management is done,

you just clear what you want the final result or end result and terraform will figure out how to execute it.

( versus imperative style where you specify how to execute each step )

**Terraform is a tool for infrastructure provisioning** so what does it mean exactly

let's say you just started a project where you create some application and you want to set up an

infrastructure from scratch where this application will run.

there are two separate steps of creating the whole setup

* one is provisioning the infrastructure preparing everything so the application can be deployed
* the second one is actually deploying the applications on it

so where does terraform come into this whole thing.

Terraform is used for the **first** part where you provision the infrastructure to prepare it for the application deployment

creating the VPC, spinning up the servers, creating the security, the AWS user with its permissions, may be installing docker specific version on servers, etc and obviously all of this needs to be done in a correct order because one task may be depends on the other

**what is a difference between ansible and terraform** because they seem to be doing the same thing especially if you read the official definitions or official documentation they're sound like the same tools.

so the question is pretty logical what is the difference between them and which one should I use for my project

so let's say the similarities and differences between these two using our example setup

* first of all terraform and ansible are both infrastructure as a code meaning they're both used to automate provisioning configuring and managing the infrastructure
* however terraform is mainly infrastructure provisioning tool that's where its main power lies but it also has possibilities to deploy applications in other tools on that infrastructure
* ansible on the other hand is mainly a configuration tool so once the infrastructure is provision and is it's there, ansible can now be used to configure it and deploy applications install and update software on that infrastructure etc
* so as you see there overlaps of what each tool does and this creates the confusion
* other differences to consider in terms of those overlaps are ansible is more mature and terraform is relatively new and because of that is also changing dynamically and
* terraform is more much more advanced in orchestration
* so to summarize the difference terraform is a better tool for provisioning infrastructure and in ansible is a better tool for configuring that infrastructure deploying installing applications and services on them
* so it's a common practice where DevOps engineers use the combination of these tools to cover the whole set up and using both for their own strengths instead of just using one tool

so now let's go back to our use case where we created the infrastructure using terraform and on AWS provision successfully for your project and you deployed the application on it

now we decide that you want to add five more servers to the existing infrastructure to deploy more micro services because your team develops some more features and they need to be deployed and you also want to add some security configuration or maybe remove some stuff that you configure at the beginning

so now we are in the phase of managing the existing infrastructure adding some stuff reconfiguring removing some stuff etc

and using terraform you can make such adjustments to infrastructure pretty easily

and this task of managing the infrastructure is just as important because once you've created the initial infrastructure for your project you will be continually adjusting and changing it

and because of that you also need some automation tool that will do most of the heavy lifting for you so that you don't have to manually configure and do same stuff

so once you are set up with terraform to create and change or maintain your infrastructure another useful thing or a common use case could be **replicating that infrastructure**

**Scene:**

let's say after you have tested this setup and everything works fine you decide now you want to release your application in production environments

so you want to create a production environment that replicates this exact setup and keep the first as a development environment where you can test new features new micro services and updates before you launch it into production

again you can use terraform here to automate that process you can easily spin up an identical infrastructure and setup using the same terraform run code that you use for the first setup the development environment setup

and you can do the same to speed up at identical staging server as well so that makes these tasks also very easy

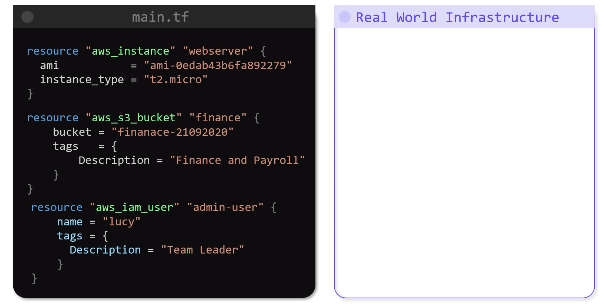
So we said that the code is declarative.

But what is declarative mean?

The code we defined is the state that we want our infrastructure to be in.

That's the desired state.

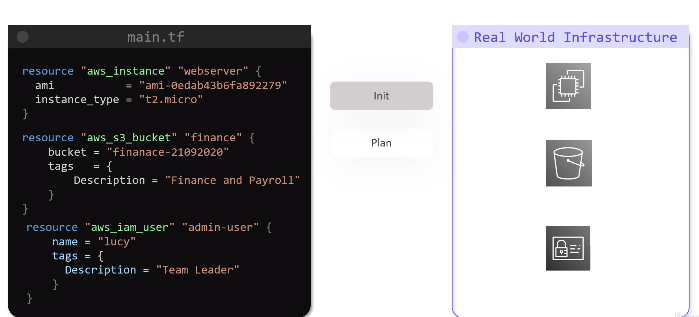
And this on the right is the current state where there's nothing.

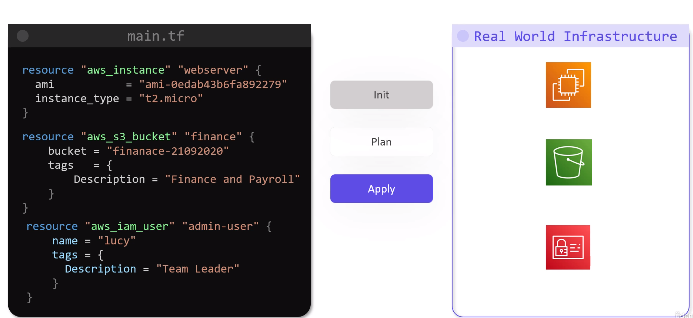


terraform will take care of what is required to go from the current state to the desired state without us having to worry about how to get there.

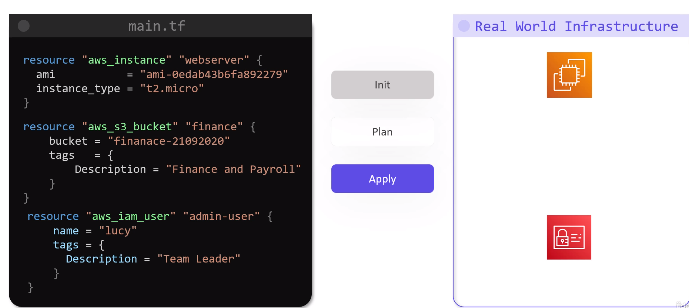
So how does Terraform do that?

Terraform works in three phases **init, plan** and **apply**

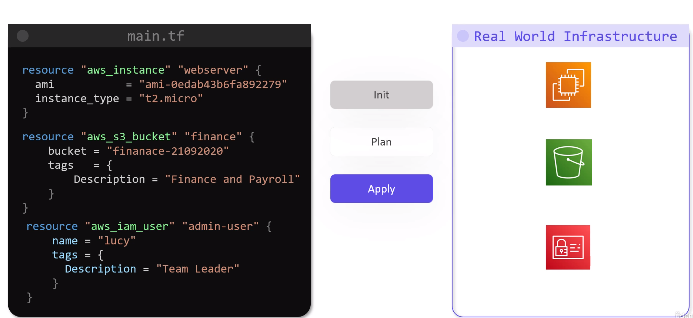
* during the init phase terraform initializes the the project and identifies the providers to be used for the target environment
* during the plan phase terraform draws a plan to get to the target state, 
* and then in the apply phase terraform makes necessary changes required on the target environment to bring it to the desired state.



* If for some reason the environment was to shift from the desired state, then a subsequent terraform apply will bring it back to the desired state by only fixing the missing component.



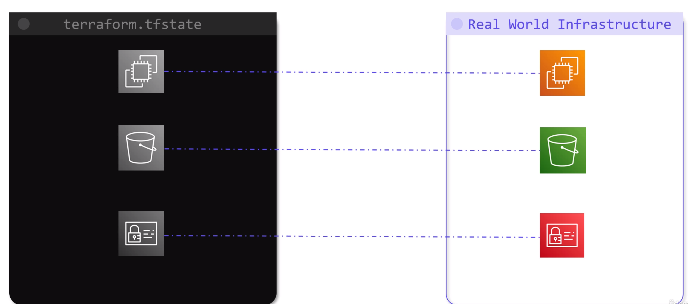
---- terraform apply



Every object that Terraform manages is called a resource, a resource can be a computer, or a database server in the cloud or in a physical server on premise that terraform manages.

It Manages the lifecycle of resources from its provisioning to configuration to decommissioning.

Terrform records the state of the infrastructure as it is seen in the real world, and based on this, it can determine what actions to take when updating resources for a particular platform.



Terraform can ensure that the entire infrastructure is always in the defined state at all times.

The state is a blueprint of the infrastructure deployed by Terraform.

Terraform can read attributes of existing infrastructure components by configuring data sources.

This can lead to be used for configuring other resources within Terraform.

Terraformed can also input other resources outside of the terraform that were either created manually or by the means of other IAC tools and bring it under its control so that it can manage those resources

Terraform cloud editor from Enterprise provide additional features that allow simplified collaboration between teams managing infrastructure, improved security and a centralized UI to manage terraform deployments.

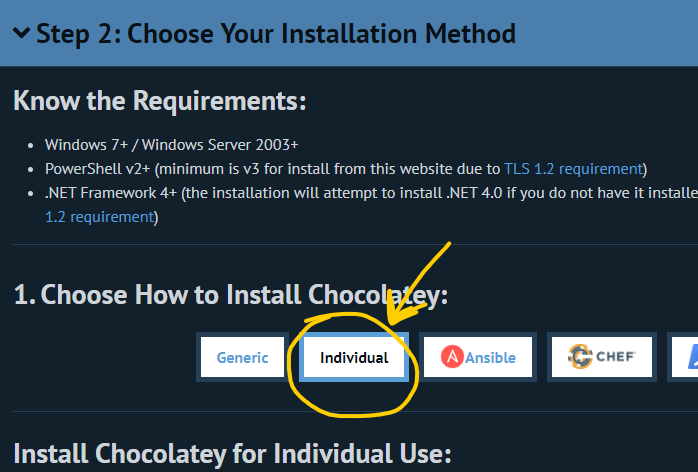
All these features make Terraform an excellent enterprise grade infrastructure provisioning tool.

**Terraform install:**

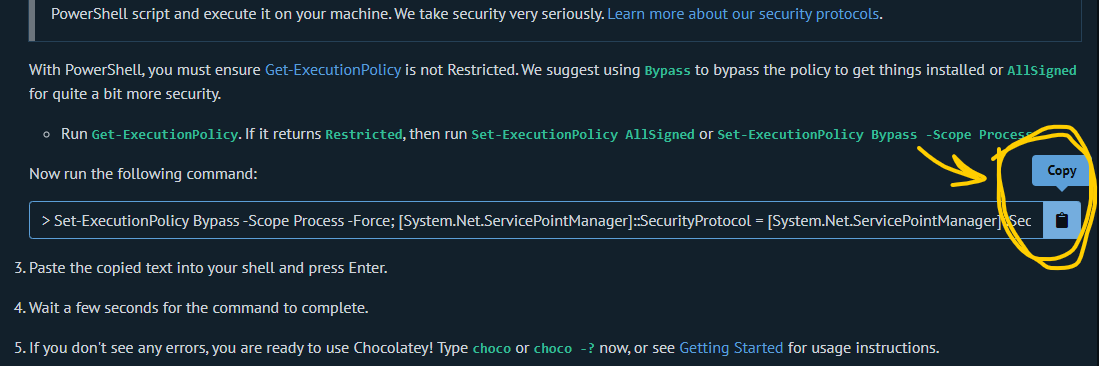
Two ways:

First : Via chocolaty.

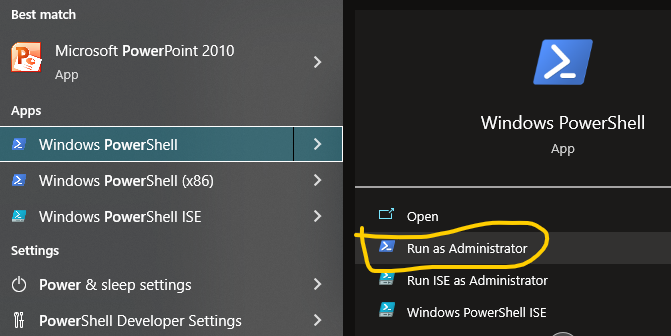
* visit <https://chocolatey.org/install>
* go to step 2, select “individual”



* scroll down and copy the command provided in step 3.



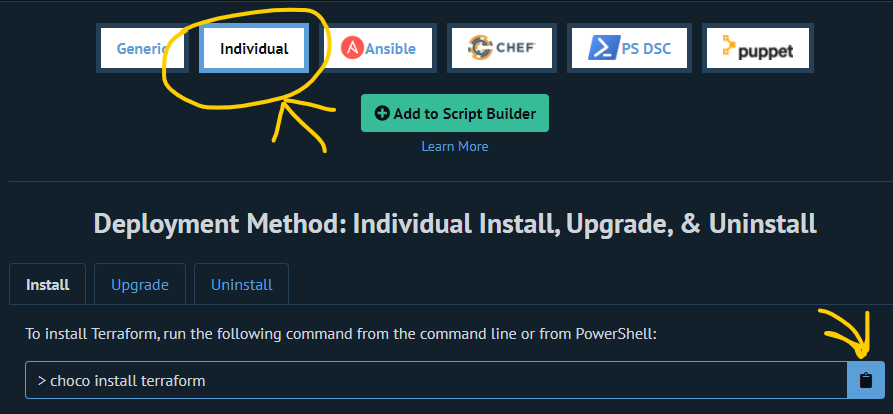
* Search for powershell in windows start, Run powershell as administrator on your system,



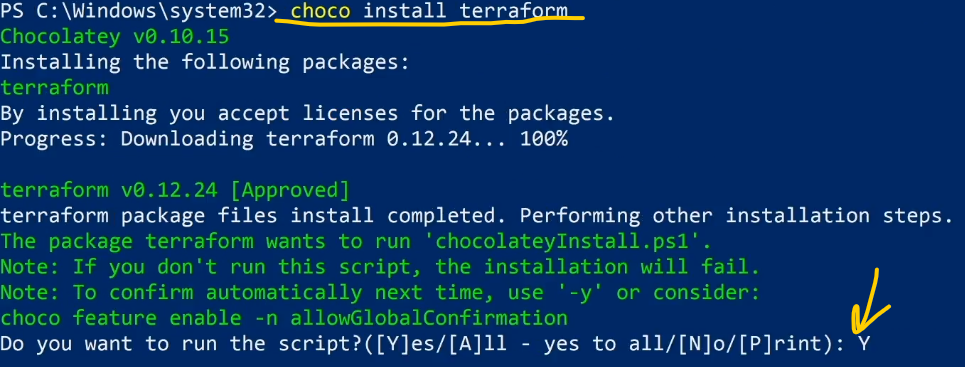
* Paste the command in powershell and press Enter. Now chocolaty will be installed.
* To check chocolaty installation, execute command “choco” , it will print its version.



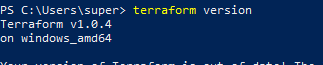
* To install terraform go to <https://community.chocolatey.org/packages/terraform> and copy the command from there



* Again come back to powershell, and paste the command and press enter.
* It will install terraform, and ask “Do you want to run the scripts?”, type y and press enter.



* Now terraform is installed, to check, execute command “terraform version”, it will print the version of terraform installed.



Another way to intall Terraform is through downloading terraform and setting its path.  
You can follow the steps from this video : <https://youtu.be/bFgymnZk9O4>



After creating our first Resource,

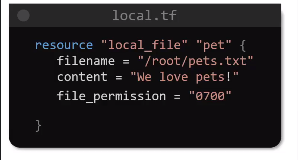
In here, we will update infrastructure using Terraform.

previously, we saw how to create a local file.

Now let us see how we can update and destroy this resource using Terraform.

First, let us try to update this resource.

Let us add in a file permission argument to obtain the permission of the file to 0700 instead of the default value of 0777.



This will remove any permission for everyone else except the owner of the file.

Now, if you run **terraform plan**, we will see an output.

From the output, we can see that the resource will be replaced. The minus plus symbol in the beginning of the resource name in the plan implies that it will be deleted and then recreated the line with the comment that reads **“Force replacement”** is responsible for the deletion and recreation.

And in this example, this is caused by the file permission argument that we added to the configuration file.

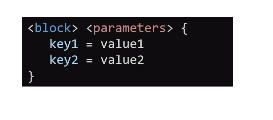
**Even though the change we made was simple, terraform will delete the old file and then create a new file with the updated permissions. This type of infrastructure is called an immutable infrastructure.**

Now we will use “**terraform destroy**”.

**HCL : HashiCorp Configuration Language** (Basics)

Let us understand the HCL syntax, the HCL file consists of blocks and arguments.

A block is defined within curly braces, and it contains a set of arguments in key value pair format representing the configuration data.



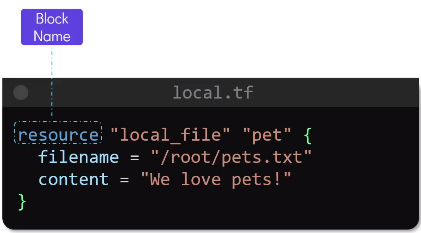
But what is a block and what arguments does it contain?

In its simplest form, a block in terraform contains information about the infrastructure platform and a set of resources within that platform that we want to create.

We created a local file with the help of this below tf file, to understand what each line means.

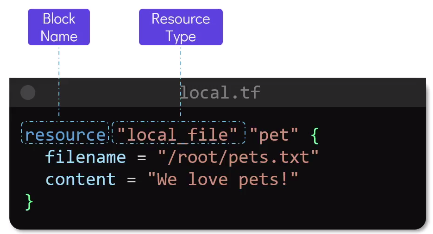


The first element in this file is a **block** of this can be identified by the curly braces.



The type of block we see here is called the **resource block**, and this can be identified by the keyword called **resource** in the beginning of the block.

Following the keyword called resource, we have the declaration of the **resource type** that we want to create.



This is a fixed value and depends on the provider where we want to create the resource. In this case, we have the resource type called local file.

A resource type provides two bits of information.



-First is the **Provider**, which is represented by the word before the underscore in the resource type. Here we are making use of the **local provider**.

The word, following the underscore, which is **file** in this case, represents the **type of resource**.

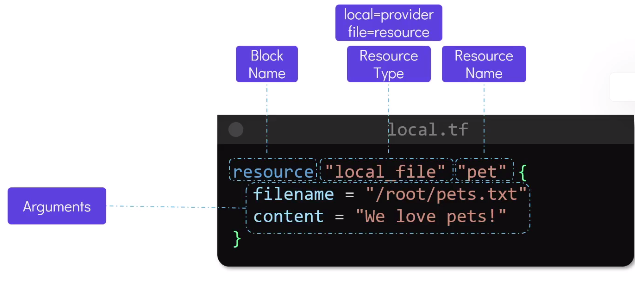
**Provider**- A provider is a Terraform plugin that allows users to manage an external API. Provider plugins like the AWS provider or the cloud-init provider act as a translation layer that allows Terraform to communicate with many different cloud providers, databases, and services.

Each provider adds a set of resource types and/or data sources that Terraform can manage. Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure.

The next and final declaration in this resource block is the **resource name**. This is the logical name used to identify the resource, and it can be named anything. But in this case, we have called it Pet as the file we are creating contains information about pets.



And within this block and inside the curly braces, we defined the **arguments** for resource, which are written in key value pair format.



These arguments are specific to the type of resource we are creating, which in this case is the local file.

The first argument is the filename to this we assign the absolute path to the file.

Now we can also add some content to this file by making use of the content argument to this let us add the value “we love pets”

The words **filename** and **content** are specific to the local file resource we want to create and they cannot be changed.

In other words, the resource type of local file expects that we provide the argument or filename and content.

Each resource type has specific arguments that they expect.

How do we know what resource type other than local file are available under the provider call local and finally, how do we know what arguments are expected by the local file resource.

For that we have terraform documentation.

**Providers**:

we run Terraform init within a directory containing the configuration files, terraform downloads and installs plugins for the provider's used within the configuration.

These can be plugins for cloud providers such as AWS, GCP, Azure, or something as simple as the local provider that we use to create a local file type resource.

Terraform uses a plugin based architecture to work with hundreds of such infrastructure platforms. Different providers are distributed by HashiCorp and are publicly available in the Terraform Registry.

There are **three tiers of providers**.

* The first one is the **official provider**. These are owned and maintained by Hashi Corp. and include the major cloud providers such as GCP and Azure. The local provider that we have used so far is also an official provider.
* The second type of provider is a **verified provider**. A verify provider is owned and maintained by a third party technology company that has gone through a partner provider process with HashiCorp.

Some of the examples are the big IP, a provider from F.I. networks, Heroku, Digital Ocean, etc.

* And finally, we have the **community providers** that are published and maintained by individual contributors of the community.

The plugins are downloaded into a hidden directory called terraform/plugins in the working directory containing the configuration files.

**Configutation Directory**

Terraform will consider any file with the “**.tf**” extension within the configuration directory.



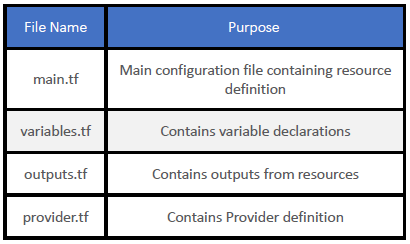
Another common practice is to have one single configuration file that contains all the resource blocks required to provision the infrastructure.

A single configuration file can have as any number of configuration blocks that you need.

A common naming convention used for such a configuration file is to call it the main.tf



There are other configuration files that can be created within the directory, such as the **variables.tf**, **Outputs.tf**, and **Providers.tf**.



**Multiple Providers**



If we add different kind of provider after first “**terraform init**” command, then we need to use the “init” command again, because now terraform needs to download plugin for the new provider as well.

**Variables** in Terraform



We have you several arguments in our resource Block so far for the local file.

Since these values are directly defined within the main configuration files, they are considered to be hard coded values.

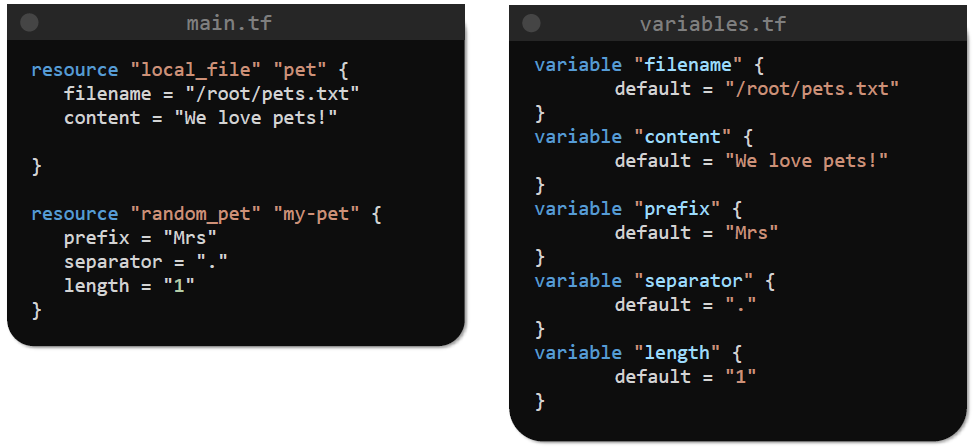
Hard coded values is not a good idea.

This limits the reusability of the code, which defeats the purpose of using IAC. We want to make sure that the same code can be used again and again to deploy resources based on a set of input variables that can be provided during the execution.

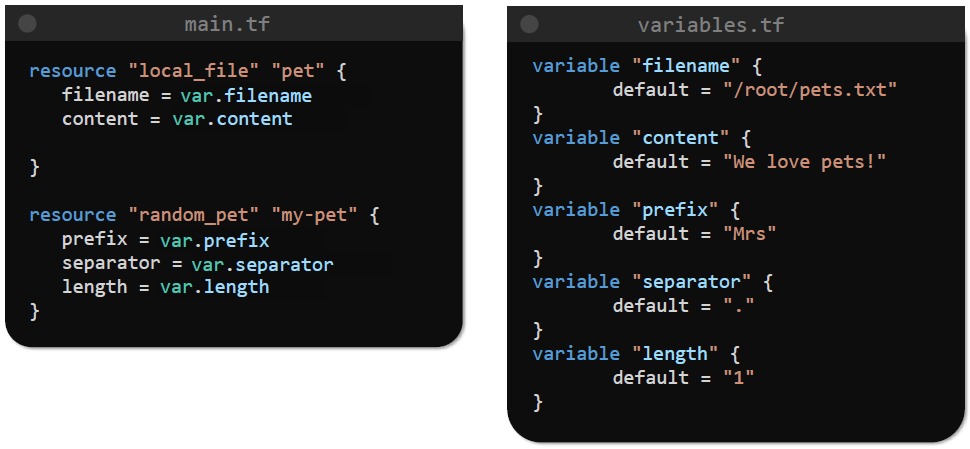
And that is where input variables come into the picture.

To assign variables let us create a new configuration file called **variables.tf**

The variables.tf file, just like the main.tf file, consists of blocks and arguments to create a variable.



and define the values like this.



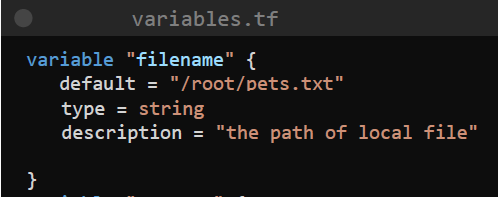
Now, if you want to make an update to the resources by making changes to the existing arguments, we can do that by just updating the variables.tf file.

The main.tf file need not be modified.

Example: making EC2 instance



Let us look at the different arguments that our variable block uses.



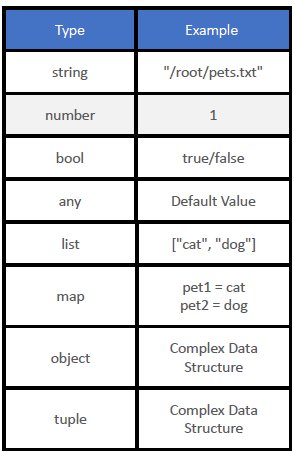
The **default** parameter, this is where we specify the default value for a variable.

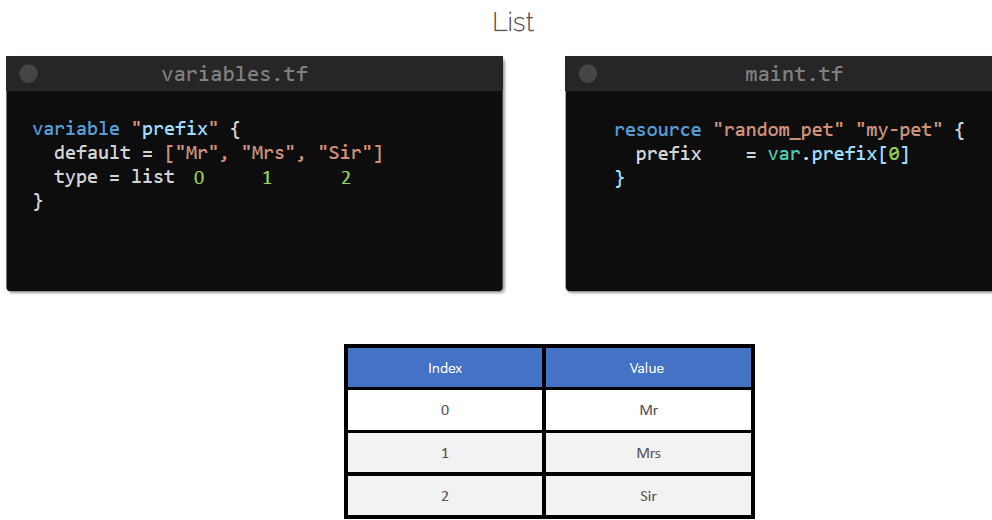
The **description** is optional, but it is a good practice to use this argument to describe what the variable is used for.

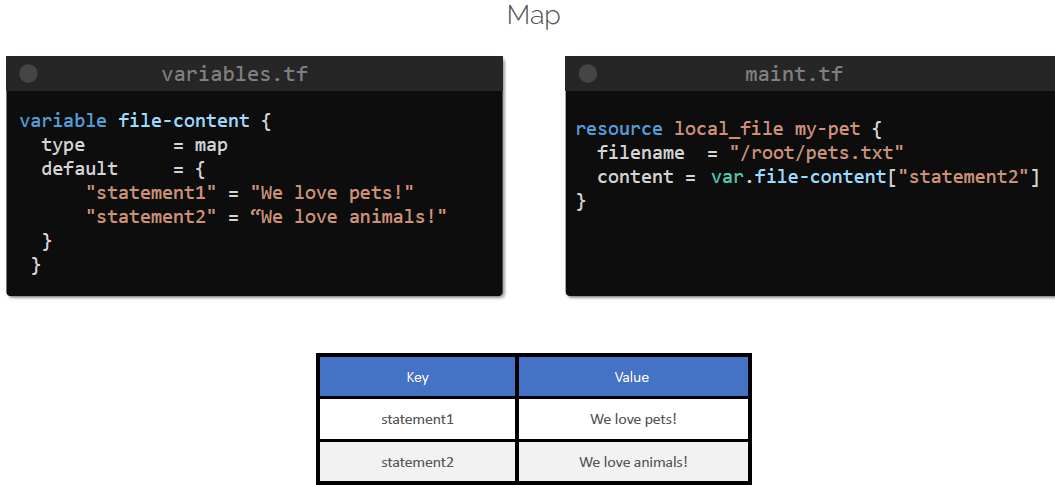
The **type** argument is optional as well, but when used it enforces the type of variable being used.

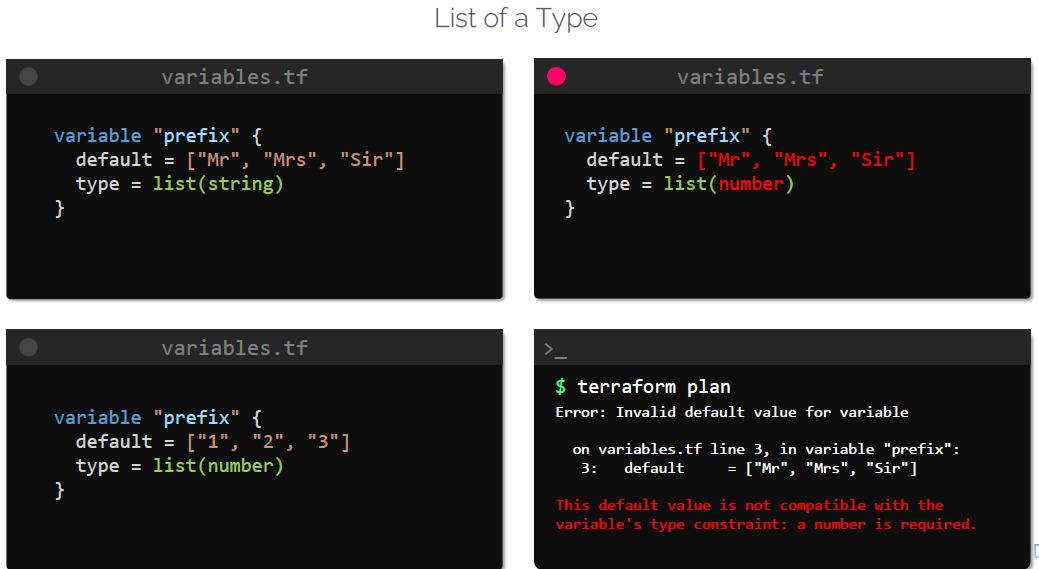
The type parameter is optional but if it is not specified in the variable block then it is said to be type **any** by default.

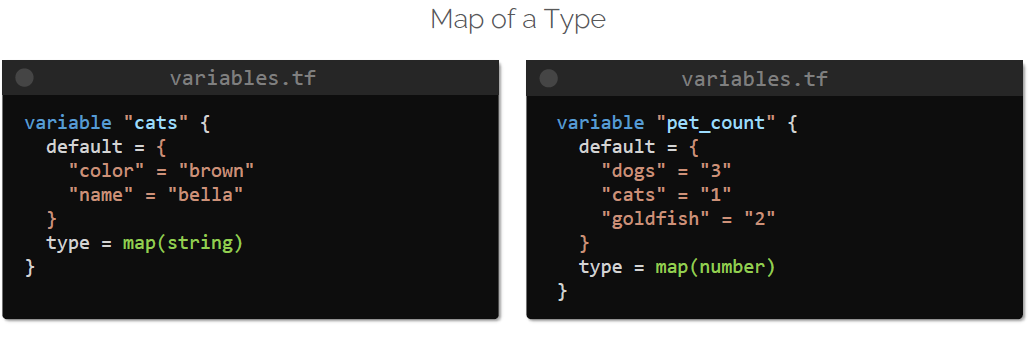
Terraform supports additional types such as



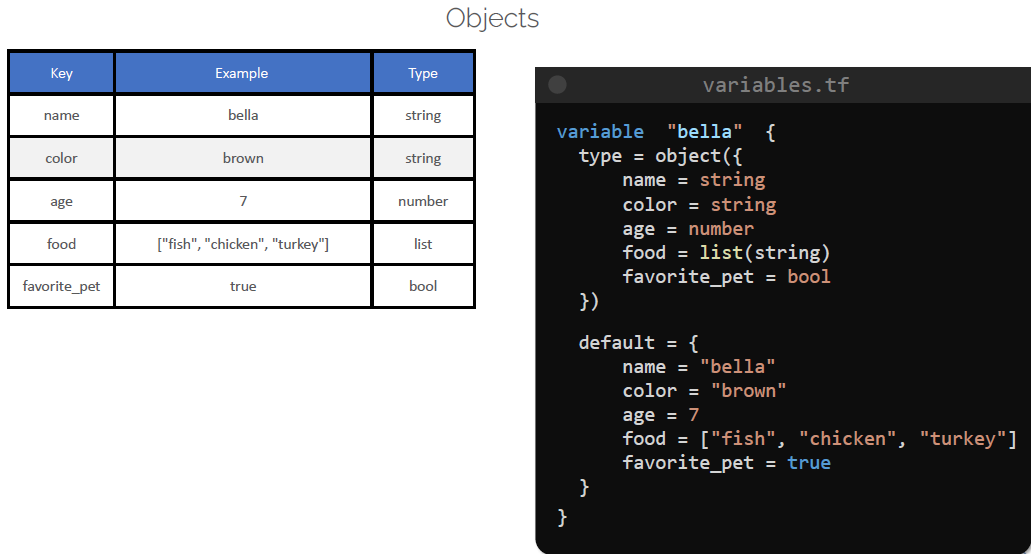


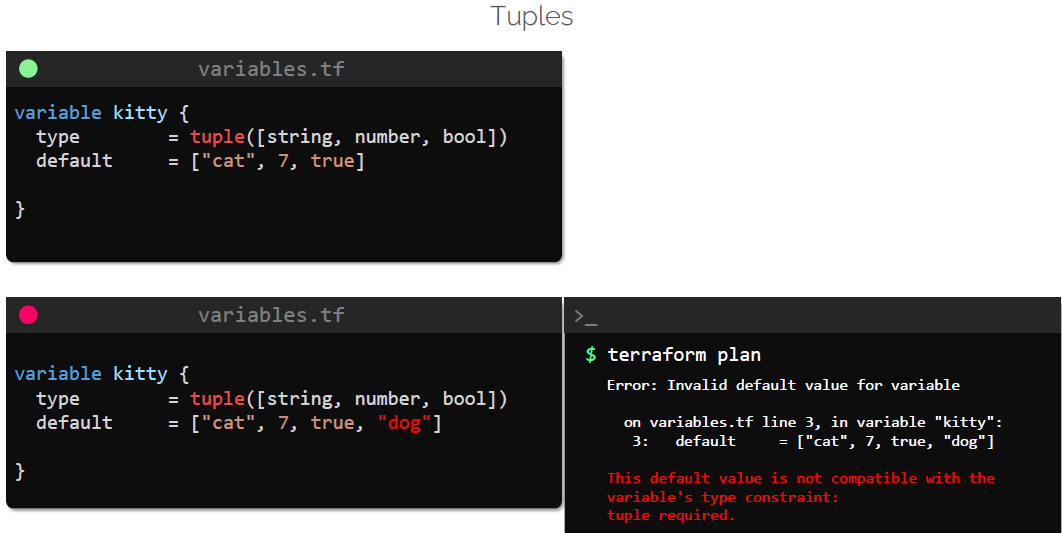












Tuples is similar to a list and consists of a sequence of elements.

The difference between a tuple and a list as that list uses elements of the same variable type, such as string or number. But in case of tuple, we can make use of elements of different variable types.

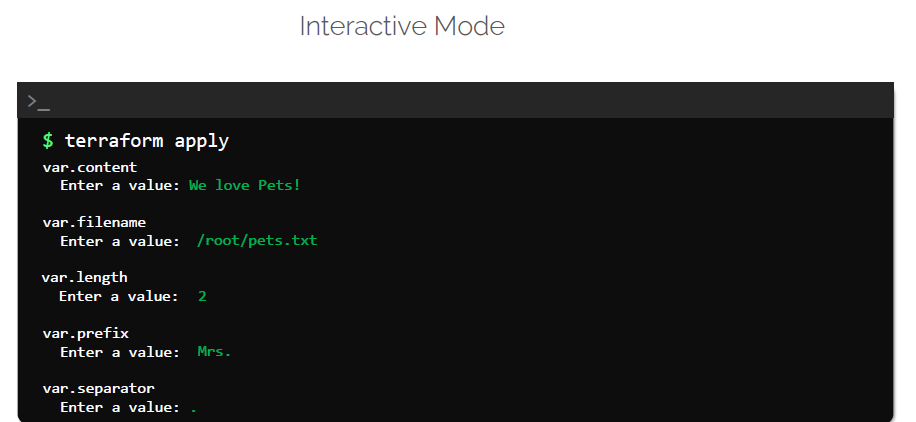
The type of variables to be used in a tuple is defined within square brackets.

**Input variables**

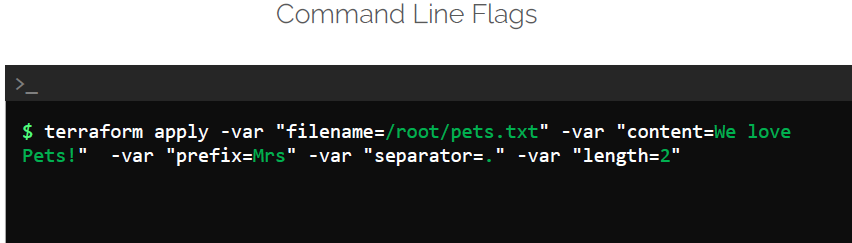
So far, we have created input variables in Terraform and assigned default values to it based on a variable type. This is just one of the ways to pass on values to the variable.

Earlier, we learned that the default parameter in a variable block is optional. This means that we can we will have our variable block look like this.

Then we will be prompted to enter values for each variable used, in an interactive mode.



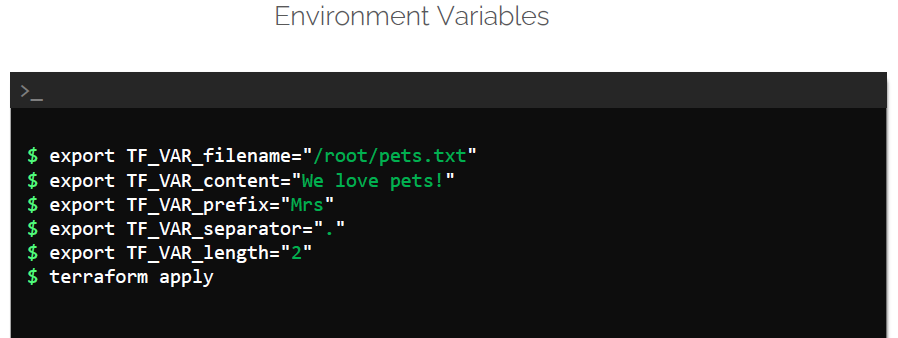
If you do not want to supply values in an interactive mode, we can also make use of command line flags like this.



When we run the terraform comand we can make use of the “**–var**” option with the variable name equals to the value format.

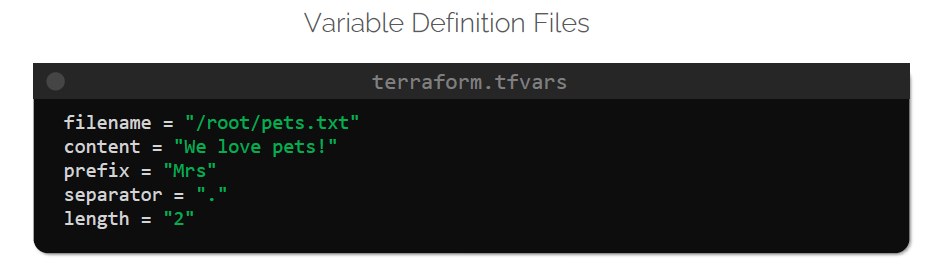
We can pass on as many variables as we want with this method by making use of the “–var” flag multiple times.

We can also make use of environmental variables with TF\_VAR\_<NAME\_OF\_VARIABLE>



Note: In windows you can use set varname=value

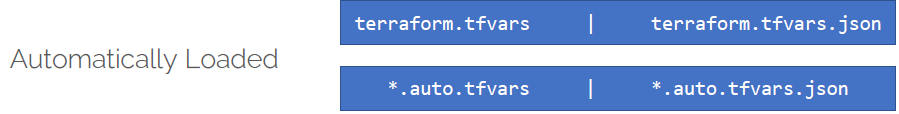
And finally, when we're dealing with a lot of variables, we can load values by making use of variable definition files like below. These variable definition files can be named anything, but should always end either “**.tfvars**” or “**.tfvar.json”**



Here we have declared variables and the values in a file called **terraform.tfvars**.

If you look at the syntax used to create this file, you'll observe that this is using the same syntax of an HCL file, but it only consists of variable assignments.

The variable definition file, if called “**terraform.tfvars**” or “**terraform.tfvars.json**” or by any other name ending with “**.auto.tfvars**” or “**.auto.tfvars.json**” will be automatically loaded by Terraform.



If we use any other finally, such as variables**.tfvars**, for example, you will have to pass it along with a command line flag called **–var-file** like below.



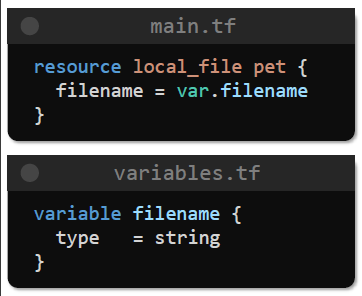
Finally, it is important to note that we can use any of the options that we have seen so far to assign values to the variables. But if we use multiple ways to assign values for the same variable,

that often follows a variable definition precedence to understand which value it should accept.

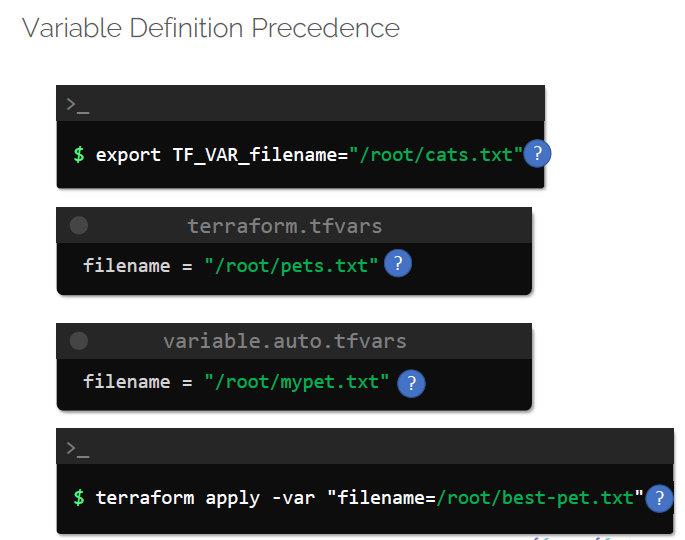
To illustrate this, let us make use of a simple example.

In this case, we have a main configuration file with a single resource, a local file which will create file at a path declared in a variable called “filename” in the “variables.tf” file.

We have not specified a default value for this variable.



And now we have assigned different values to this variable in multiple ways.

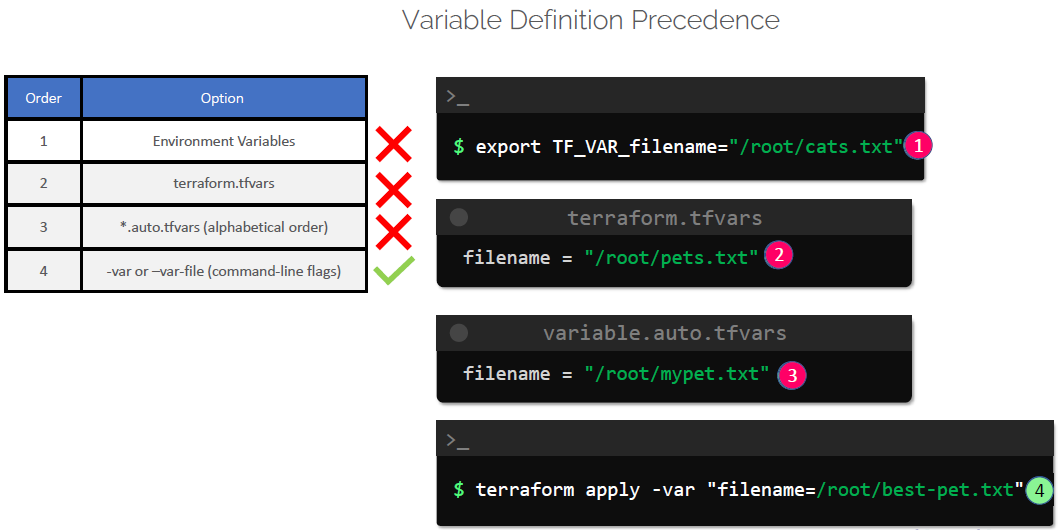


So in this case, which one of these values would be accepted?

Terraform follows a variable definition, precedence order to determine this.

* First, it loads the **Environment variables**.
* the value in **terraform.tfvars** file.
* This is followed by any file that ends with the **.auto.tfvars** or **.auto.tfvars.json** in an alphabetical order.
* And finally, terraform considers the **command line flag** of **–var** or **–var-file**, which take the highest priority and will override any of the previous values.

In this case, the variable filename will be assigned the value of “/root/best-pet.txt”.



**Resource Attributes**

In a realworld infrastructure provisioning process we have multiple resources that are dependent on each other.

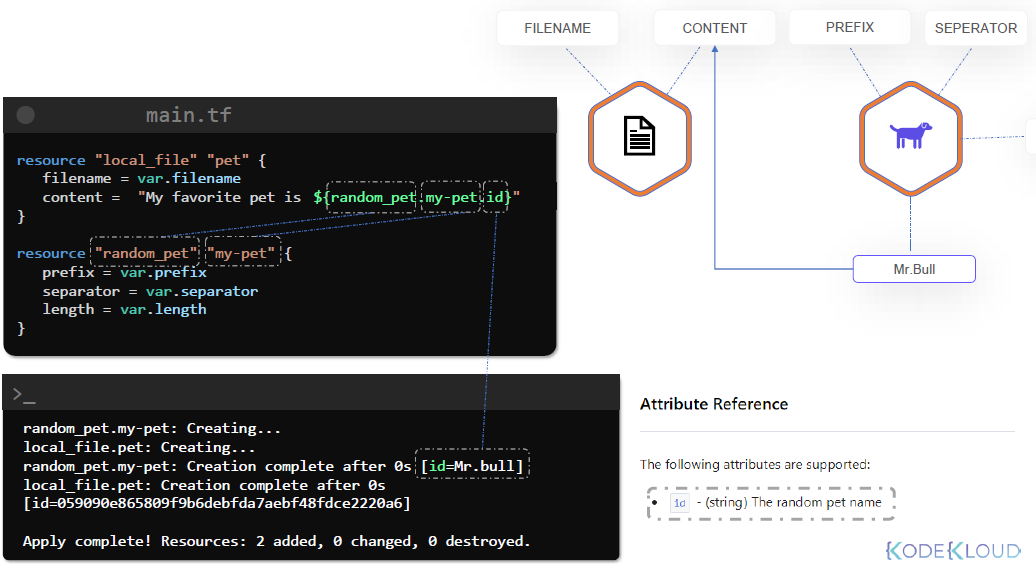
So what if you want to make use of the output of one resource and use it as an input for another one?

What if we want the content of the file to use the name generated by the random pet resource?

On terraform registry there is a section called Attribute Reference, which provides the list of attributes return back from the resource after we run terraform apply.

In our case, the random\_pet resource returns, just one attribute called id, which is of type string.

The ID is the pet name generated after we run terraform apply, as we have seen before. Our goal is to make use of the attribute called ID and to make use of it as the content of the local file resource.



**Resource Dependencies**

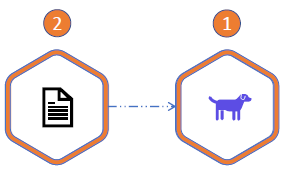
Let’s have a look at the different types of resource dependencies in terraform.

Previously we saw how to link one resource to another using reference attributes by making use of reference, expression and interpolation.

We were able to make use of the output of this random\_pet resource as an input for the local\_file resource.

Now, when terraform creates these resources, it knows about the dependency, since the local file resource depends on the output of the random\_pet resource.

As a result, it uses the following order to provision them first.

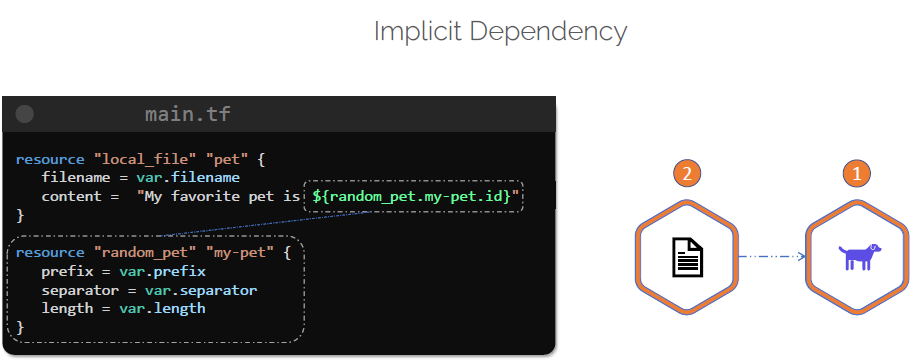


Therefore, it creates the random\_pet resource, and then it creates the local\_file resource.

When resources are deleted, Terraform deletes it in the reverse order, the local file first and then the random pet.

This type of dependency is called the **implicit dependency**.

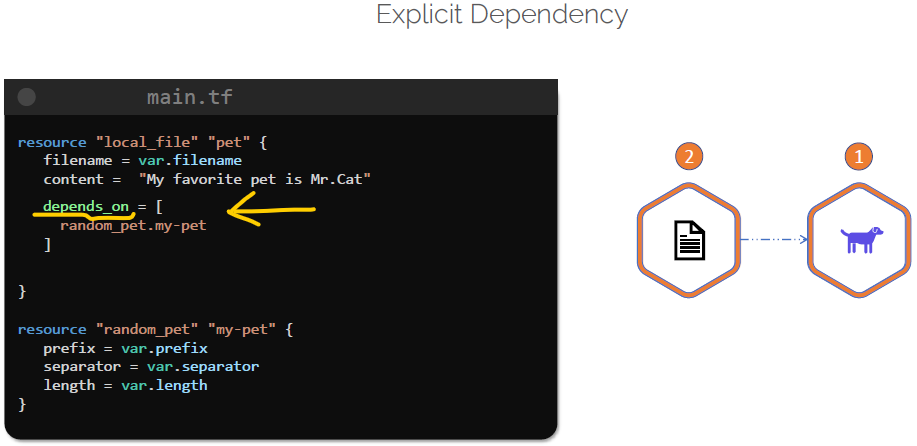
We are not explicitly specifying which resource is dependent on which other resource to terraform figures it out by itself.



However, there is another way to specify dependency within the configuration file.

For example, let us make use of the all the configuration file without making use of the reference expression for the file content.

If you still want to make sure that the local fire resource is created after the random\_pet, we can do this by using the “**depends\_on”** argument like this.



Here we have added a depends on argument inside the resource block for the local file, and we are provided a list of dependencies that include the random\_pet resource called by “my-pet”.

This will ensure that the local\_file is only created after the random\_pet resources created.

This type of dependency is called an **explicit dependency**.

Explicitly specifying dependency is only necessary when a resource relies on some other resource indirectly, and it does not make use of a reference expression as seen in the previous case.

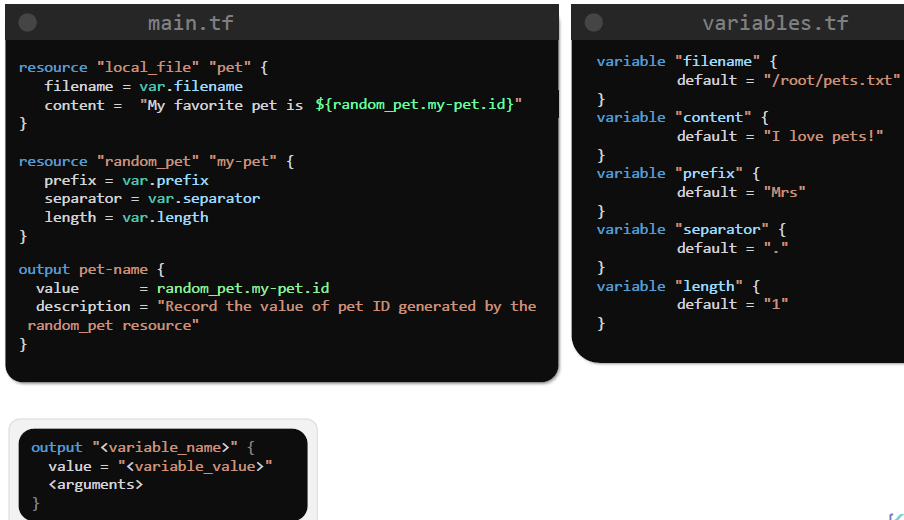
**Output variables**

Terraform also supports output variables. These variables can be used to store the value of an expression into a form.

For example, let us go back to the configuration file that we used in the previous lecture.

We already know that the random\_pet resource will generate a random pet name using the attribute called ID.

When we apply the configuration to save this “id” in an output variable called pet\_name, we can create an output block like this.

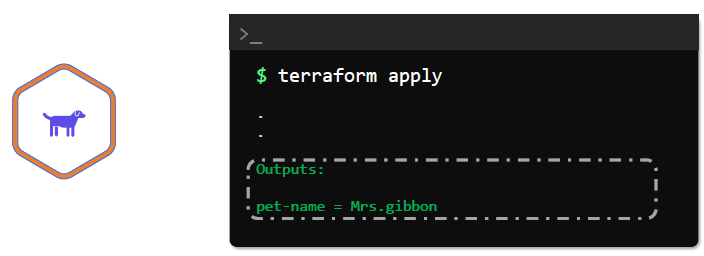


The syntax used to create this output block is the keyword called output, followed by the name that we want to call this variable inside this block.

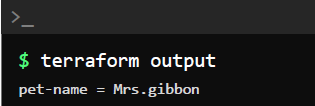
The mandatory argument for value is the reference expression.

We can also add a description, which is an optional argument to describe what this output variable will be used for.

Once this block has been created, when we run terraform apply, like we can see that the output variable is printed on the screen.

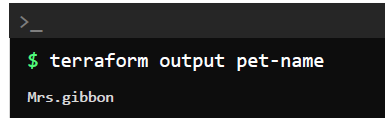


Once the resource has been created, we can also make use of the “**terraform output**” command to print the value of the output variables.



The command to “terraform output” by itself will print all the output variables define in all the files in the current configuration dedicatory.

We can also use this command specifically to print the value of an existing output variable like this.



The best use of terraform output variables is when you want to quickly display details about a provisioned resource on the screen or to feed the output variables to other IAC tools, such as Ansible playbook for configuration management and testing.

**Terraform State**

Let us take a look at the contents of the configuration directory again. After using “terraform apply” command, we can now see that there is an additional file called “**Terraform.tf.state**” created in the directory.

This file is called Terraform State File, which was created as a consequence of “**terraform apply**” command that created the resource in the first place.

This file is not created until “terraform apply” is run at least once. The state file is a JSON data structure that maps the real world infrastructure resources to the resource definition in the configuration files.

If we inspect the contents of the state file, we can see that it has the complete record of the infrastructure created by Terraform.

The details, such as the Resource ID, Provider information and all the resource attributes are stored within this file.

It contains every little detail pertaining to the infrastructure that was created by Terraform, and it uses it as a single source of truth when using command such as “plan” and “apply”.

If you make a change in the configuration file now and rerun the “plan” or “apply” command terraform by default refreshes the state again and compares it against the configuration file.

It now knows that the resource arguments have different values in the configuration file as compared to the terraform state and the real world.

And as a result, Terraform knows that the resource must be recreated.

And when we run “terraform apply”, it updates the state file as well from the output of the apply command and we can see that the older resource with the older ID is deleted and replaced with the new resource with a completely different ID with the updated contents.

The same can be seen in detail from state file as well. It no longer has a reference for the older resource it and now has the details recorded for the replaced resource.

At this point in time, the configuration file and the state file are in sync, and so they do not have any differences. There will be no changes to apply subsequently.

Terraform configuration may contain numerous resources belonging to several different providers, irrespective of the size of the infrastructure.

Terraform will always create a State file and use it to store information about the state of the infrastructure in the real world. It is non optional.

Besides the mapping between resources and the configuration and the real world, the state file also tracks metadata details such as resource dependencies.

**Consideration**

First one is that the state file contains sensitive information within it, it contains every little detail about our infrastructure. We need to make sure that the state file is always stored in a single storage.

When working as a team, it is considered a best practice to store terraform configuration files in a distributed version control systems such as GitHub, GitLab or Bitbucket.

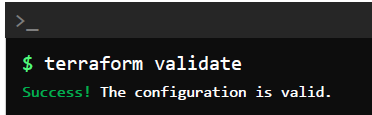
However, owing to the sensitive nature of the state file, it is not recommended to store them in git repositories. Instead, store the state in remote backend systems such as AWS-S3, Google Cloud Storage, Azure Storage, Terraform Cloud, etc.

**Terraform Commands**

**“terraform validate”**

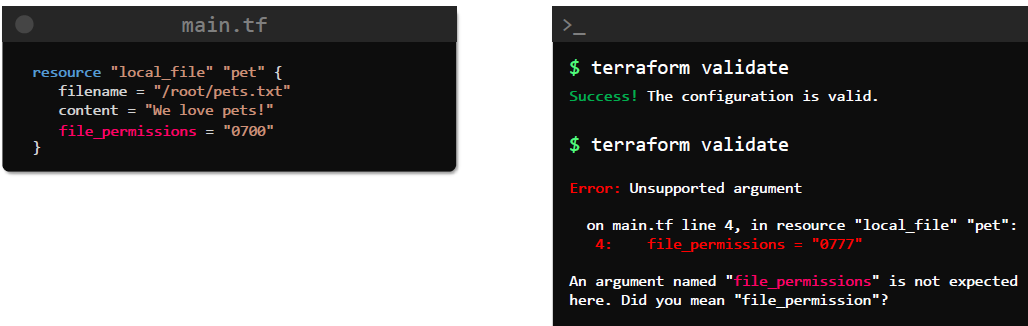
Once we write our configuration file, it's not necessary to run terraform “plan” or “apply” to check if the syntax used is correct. Instead, we can make use of “terraform validate” command like this.

And if everything is correct with the file, we should see a successful validation message like this.



If there's an error in the configuration file, the validate command will show you the line in the file that is causing the error with the hints to fix it.

In this example, we have used an incorrect argument for the local fire resource. It should be file\_permission and not file\_permissions.

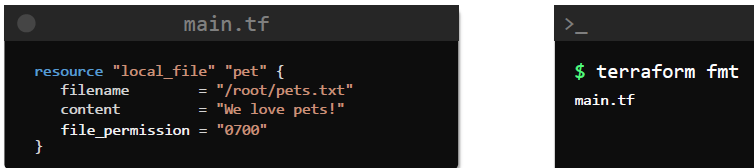


“**terraform fmt**”

This command scans the configuration files and the current working directory and formats the code into a canonical format.

This is a useful command to improve the readability of the Terraform configuration file.

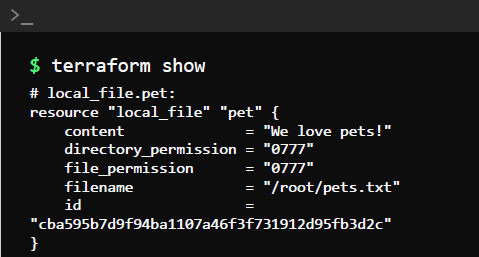
When we run this command, the files that are changed in the configuration directory are displayed on the screen.



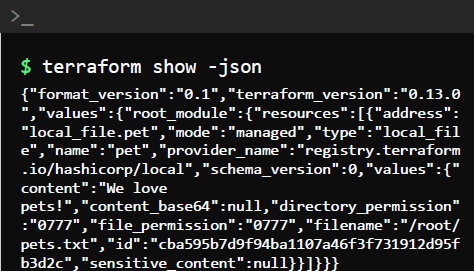
“**terraform show**”

It prints out the current state of the infrastructure, as seen by Terraform.

In this example, we have already created the local file resource and when we run the show command, it displays the current state of the resource, including all the attributes created by Terraform for that resource, such as the file name, file and directory permissions, content and ID of the resource.

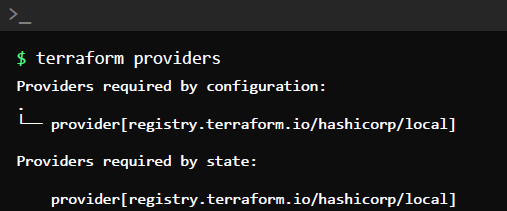


Additionally, we can make use of the **“-json”** flag to print the contents in the JSON format.

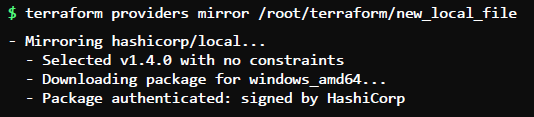


“**terraform providers”**

It is used to see a list of all providers used in the configuration directory.

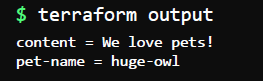


You can also make use of the “**mirror**” sub command to copy provider plugins needed for the current configuration to another directory like below. This command will mirror the provider configuration in another directory.



“**terraform output**”

If you want to print all output variables in the configuration directory use the command terraform output.



You can also print the value of a specific variable by appending the name of the variable to the end of the output command like this.



“**terraform refresh**”

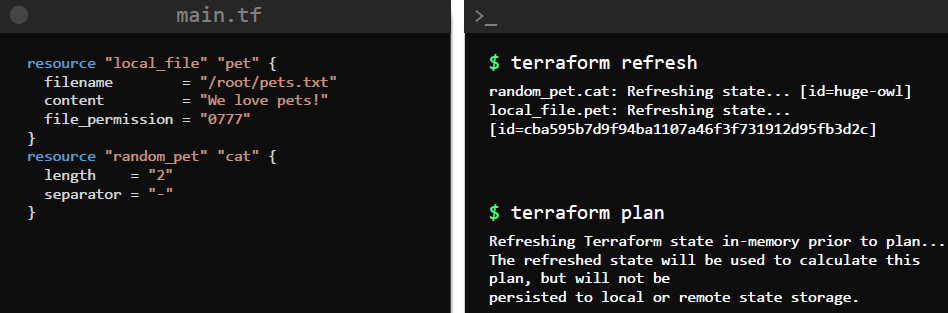
The Terraform Refresh command is used to sync terraform with the real world infrastructure.

For example, if there are any changes made to a resource created by Terraform outside its control, such as a manual update, Terraform Refresh Command will pick it up and update the state file. This reconciliation is useful to determine what action to take during the next apply.

This command will not modify any infrastructure resource, but it will modify the state file.

Terraform Refresh is also run automatically by command, such as the terraform plan and terraform apply, and this was done prior to terraform generating an execution plan.

This can, however, be bypassed by using the “**-refresh=false**” option with the commands.

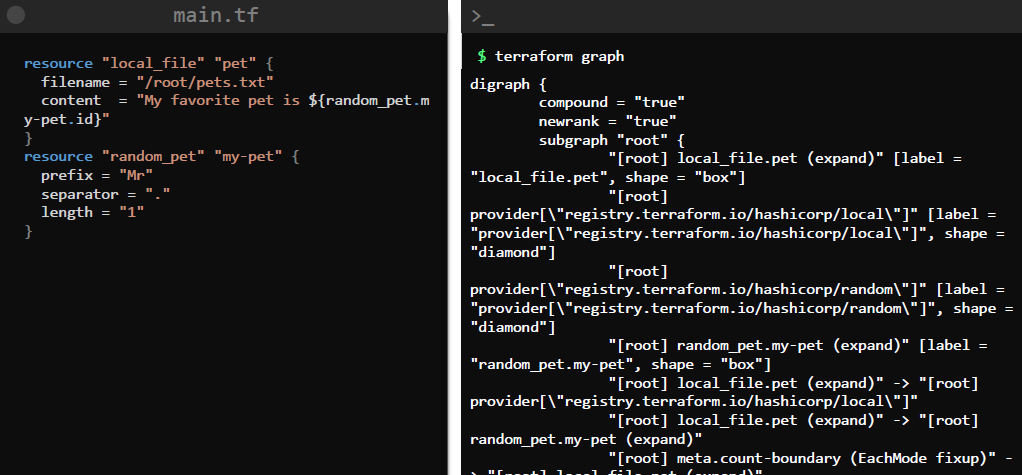


“**terraform graph**”

This Command is used to create a visual representation of the dependencies in a terraform configuration or an execution plan.

In this example, the local file in our main.tf file has a dependency on the random\_pet resource. This command can be run as soon as you have the configuration file ready, even before you have initialize the configuration directory with terraform init.

Upon running terraform graph command you should see an output like this.

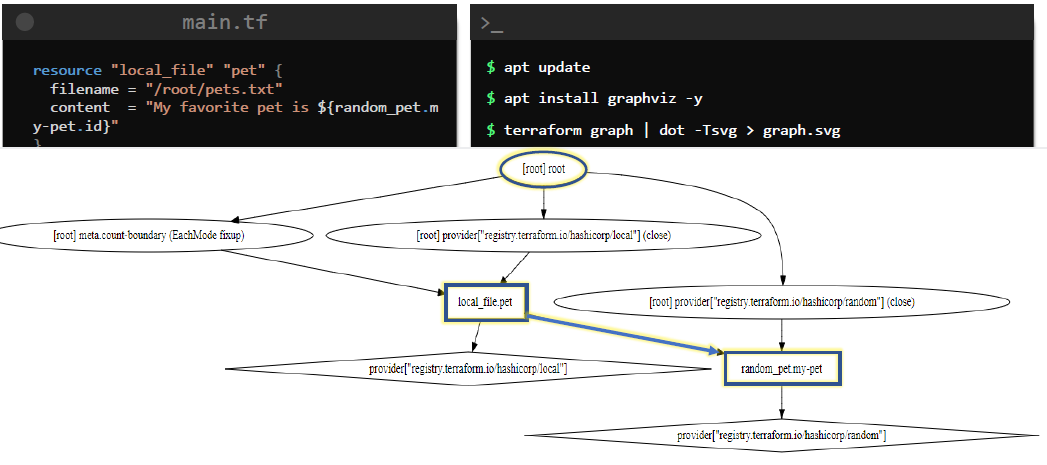


This text generated is hard to comprehend as it is, but it is a graph generated in a format called DOT.

To make more sense of this graph, we can pass it through a Graph visualization software such as Graphviz and we can install it in Ubuntu using APT.

Once installed, we can parse the output of the data from graph to the document, which we installed using the graphics package and generate a graphic like below. We can now open this file, with a browser and it should show our dependency graph like this.

The root is the configuration directory, where the configuration for this graph is located.



**Mutable and Immutable infrastructure**

<https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure>

using mutable infrastructure, can lead to “configuration drift”. Also when dealing with large infrastructure chances of configuration drift increases. Thus immutable infrastructure is preferred in DevOps.

**Configuration drift** is caused by inconsistent configuration items (CIs) across computers or devices. Configuration drift occurs naturally in data centre environments when changes to software and hardware are made ad hoc and are not recorded or tracked in a comprehensive and systematic fashion.

**Lifecycle rules**

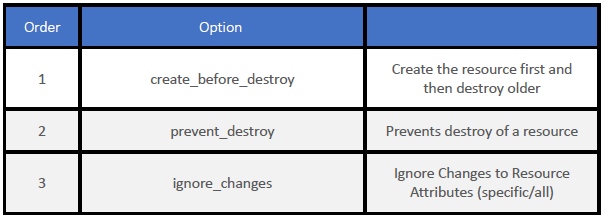
Previously, we saw that when terraform updates a resource, it treats the infrastructure to be immutable and first deletes the resource before creating a new one with the updated configuration.

For example, if we update the file permissions on our local\_file resource from 777 to 700 and then run terraform apply, you will see that the older file is deleted first and then the new file is created.

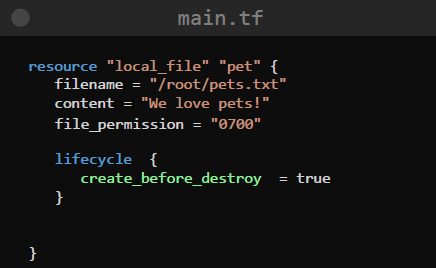
Now, this may not be a desirable approach in all cases, and sometimes you may want the updated version of the resource to be created first before the older one is deleted.

Or you may not want the resource to be deleted at all, even if there was a change made in its local configuration.

This can be achieved in TerraForm by making use of Life-Cycle rules.



The syntax of a resource block with the lifecycle rules looks like this.



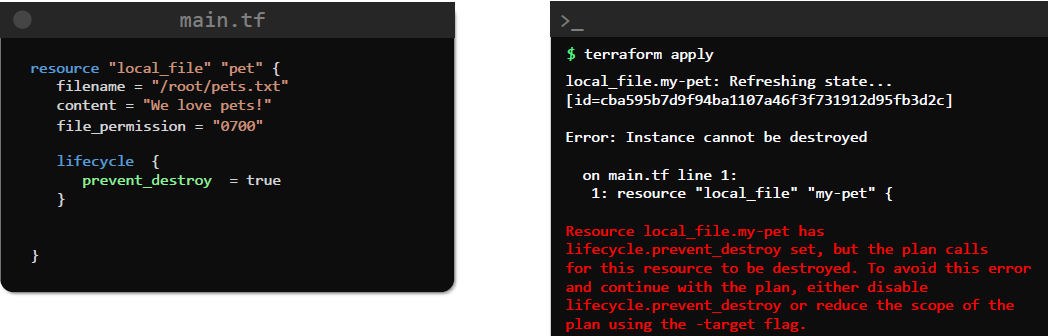
Inside the lifecycle block, we add the rule which we want terraform to adhere to while updating resources.

* create\_before\_destroy

This rule ensures that when a change in configuration forces the resource to be re-created a new resource created first before deleting the old one.

* prevent\_destroy

If we do not want a resource to be destroyed for any reason, for this we can make use of this prevent\_destroy option. When it is set to true terraform will reject any changes that will result in the resource getting destroyed and display an error message.



This is especially useful to prevent your resources from getting accidentally deleted.

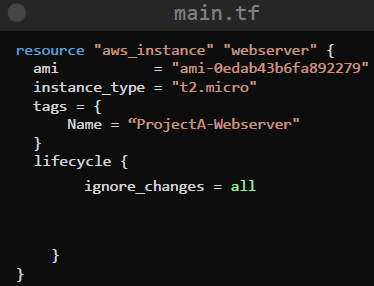
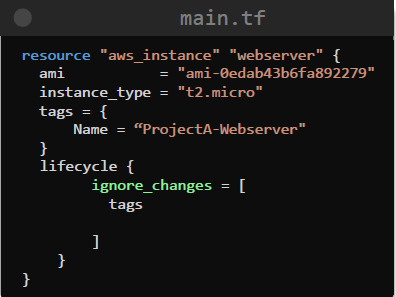
One important thing to note here is that the results can still be destroyed if you make use of the “terraform destroy” command.

This rule will only prevent resource deletion from changes that are made to the configuration and a subsequent apply.

* ignore\_changes

This Life-Cycle rule, when applied, will prevent a resource from being updated based on a list of attributes that be defined within the lifecycle block.

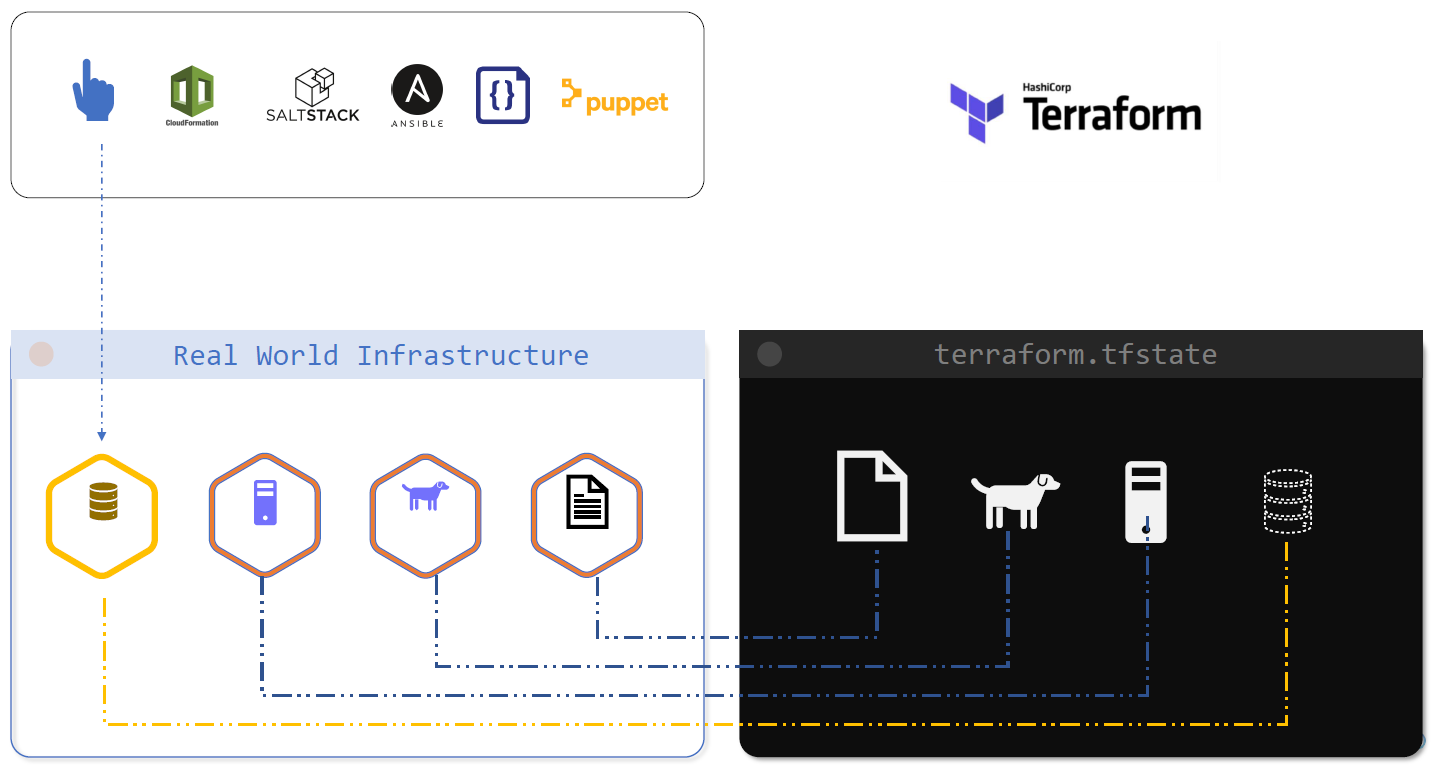
The ignore changes argument accepts a list as indicated by the square brackets, and it will accept any valid resource attribute.

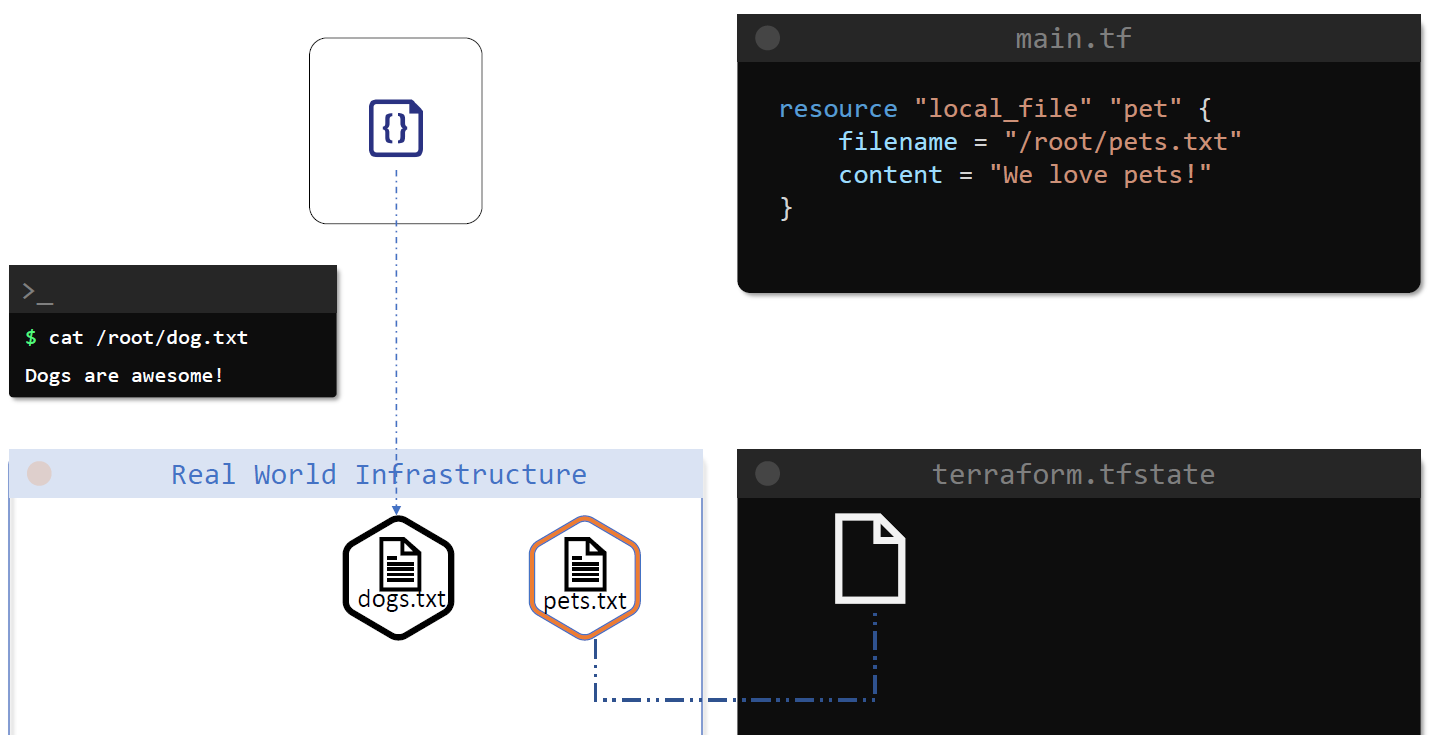
**DataSources**

Data sources allow terraformed to read attributes from resources are provision outside its control.

For instance, let us assume that a database instance was provisioned manually in the AWS cloud, although terraform does not manage this resource, it can read attributes such as the database name, host address or the DB user and use it to provision an application resource that is managed by TerraForm.



Let's take a simpler example.



We have a local file resource called pets.txt with the contents “We love pets”. Once those resources provision, the file is created in the /root directory and the information about this file has also stored in terraform state file.

Now let's create a new file using a simple shell script like this. Quite evidently, this file is outside the control and management of terraform at this point in time.

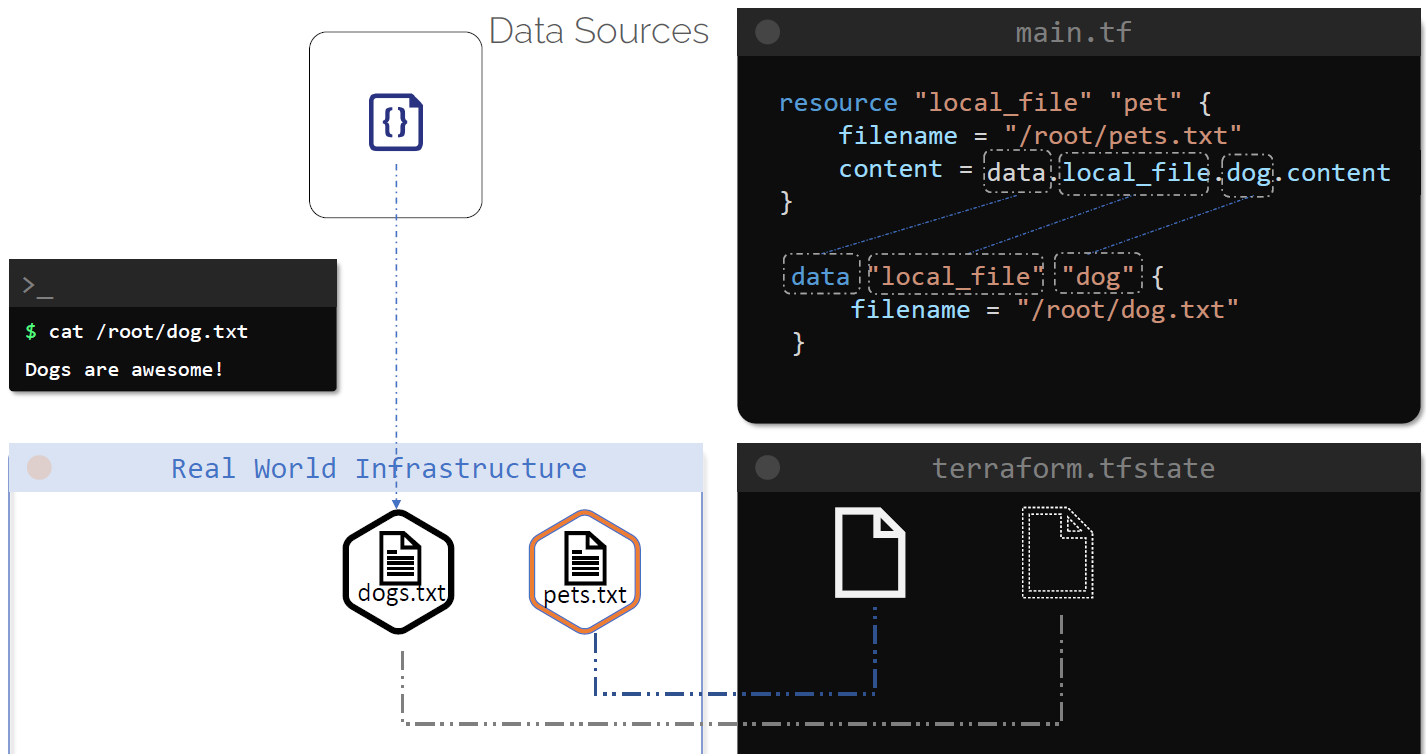
The dogs.txt has a single line that says dogs are awesome.

We would like TerraForm to use this file as a data source and use its data as contents of our existing file called pets.txt.

If we want to make use of the attributes of this new file that is created by the shell script, we can make use of datasources.

Data sources allow terraformed to read attributes from resources are provision outside its control.

For example, to read the attributes from the local file called dogs.txt we can define a data block within the configuration file like this.



As you may have noticed, the data block is quite similar to the resource block instead of the key called resource.

We define a data source block with the keyword called **data**, this is followed by the type of **resource** which we are trying to read, in this example it's a local file, this can be any valid resource type for any provider supported by TerraForm.

Next comes the **logical** **resource name** into which the attributes for a resource will be a read.

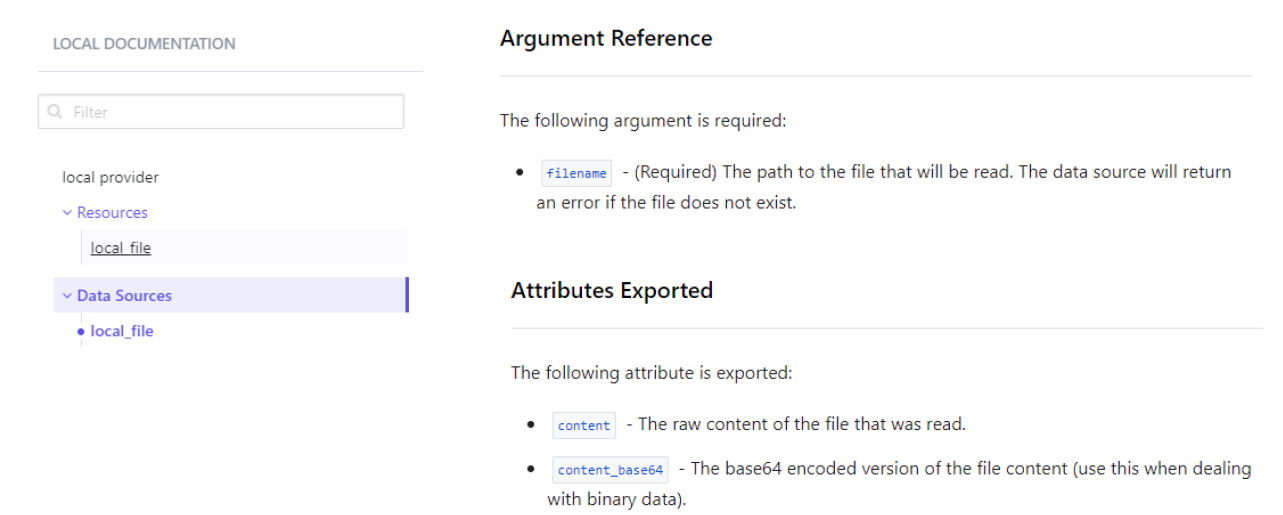
Within the block, we have arguments just like we have in a normal resource block.

The DataSource Block consists of specific arguments for a data source and to know argument is expected we can look up the provided documentation in terraform registry.

In case of our example, for the local file data source we just have one argument that should be used, which is the file name to be read.

The data load from a data source is then available under the data object in terraform, so to use this data in the resource called pet, we could simply use **“data.local\_file.dog.content”**

These details are, of course, available in terraform documentation under data sources, within the documentation and under the attributes exported

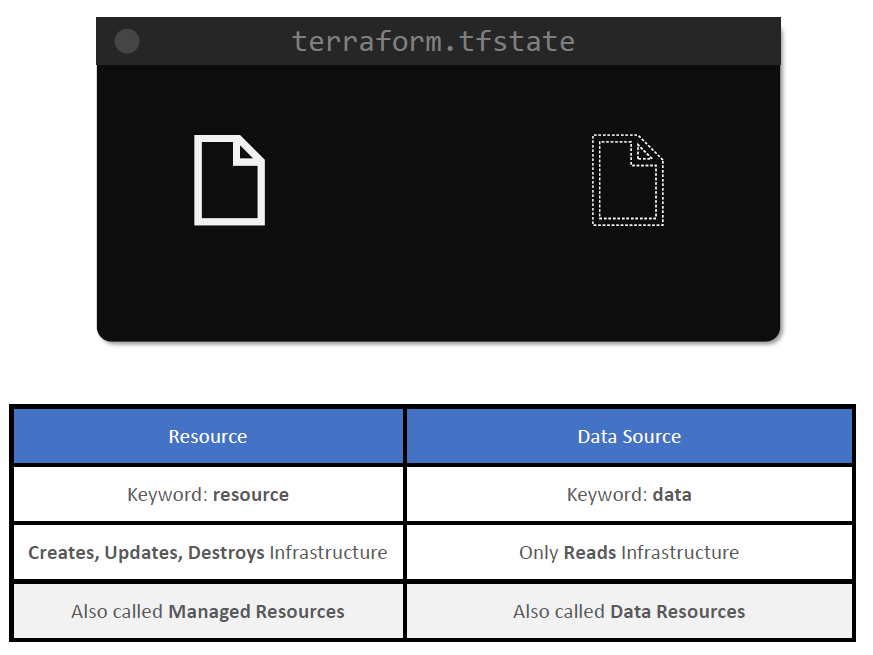


To distinguish between a resource and data sources, let's do a quick comparison.

Resources are created with the resource block and data sources are created with the data block

Resources in terraform are used to create, update and destroy infrastructure, whereas a data source is used to read information from a specific resource.

Regular resources are also called **managed resources**, as it's an extension which is managed by TerraForm, datasources are also called data resources.



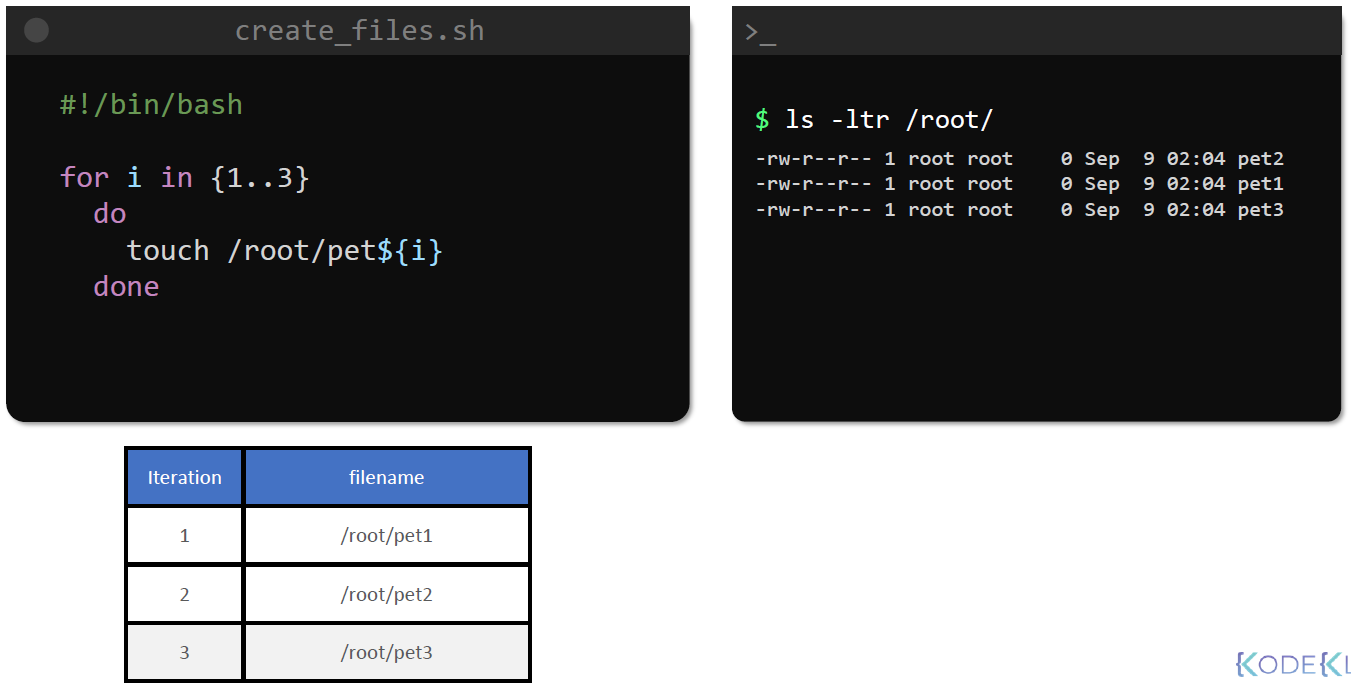
**Meta arguments**

Meta arguments can be used within any resource block to change the behaviour of resources.

Until now, we have been able to create single resources, such as a local file and a random\_pet resource using TerraForm but what if you want to create multiple instances of the same resource, say, three local files?

For example, if you were using a Shell script or some other programming language, we could create multiple files like below, in this example, we have created a shell script called create\_files.sh, which uses a for loop to create empty files inside the root directory.

The files will be called pet, followed by the range from one to three.



While we cannot use the same script as is within the resource block, TerraForm offers several alternatives to achieve the same goal.

Now, these can be done by making use of specific meta arguments in TerraForm.

Meta arguments can be used within any resource block to change the behaviour of resources.

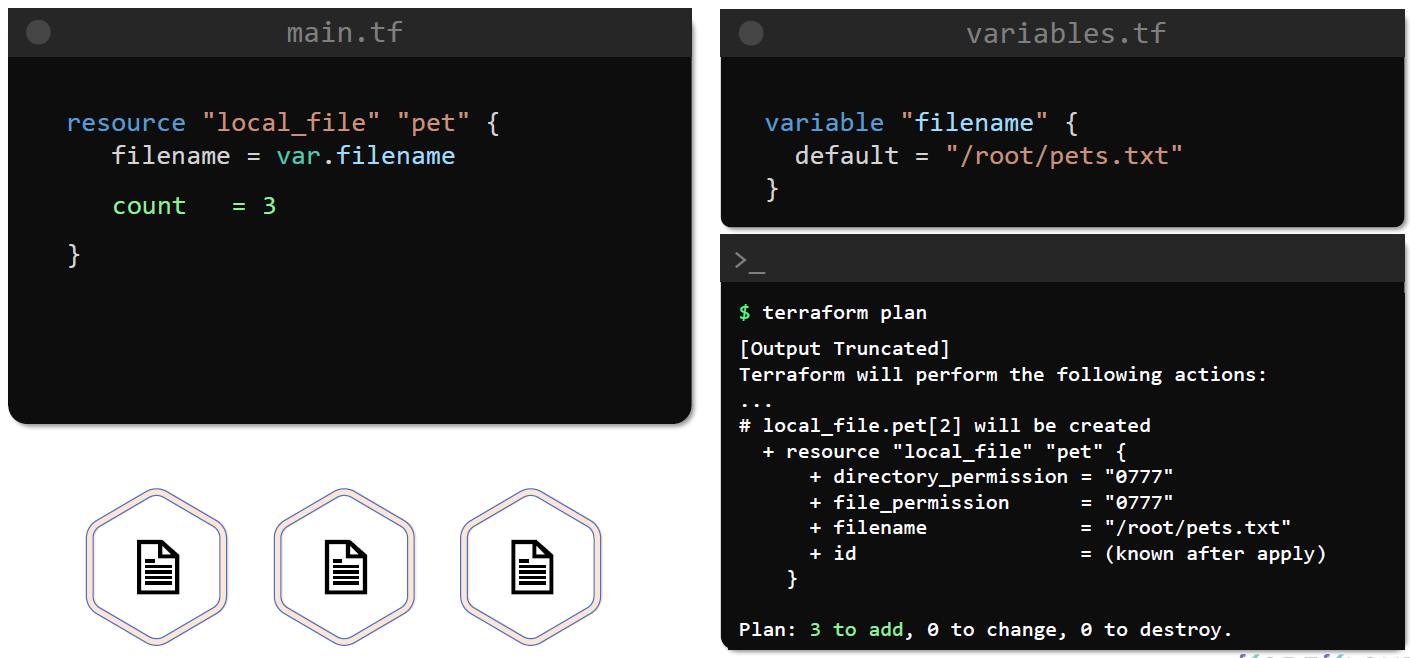
We have already seen two types of meta arguments, they are **“depends\_on”** for defining explicit dependency between resources and the **lifecycle rules** which define how the resources should be created, updated and destroyed within TerraForm.

Now let us look at some more meta arguments, Specifically related to Loop's In terraform.

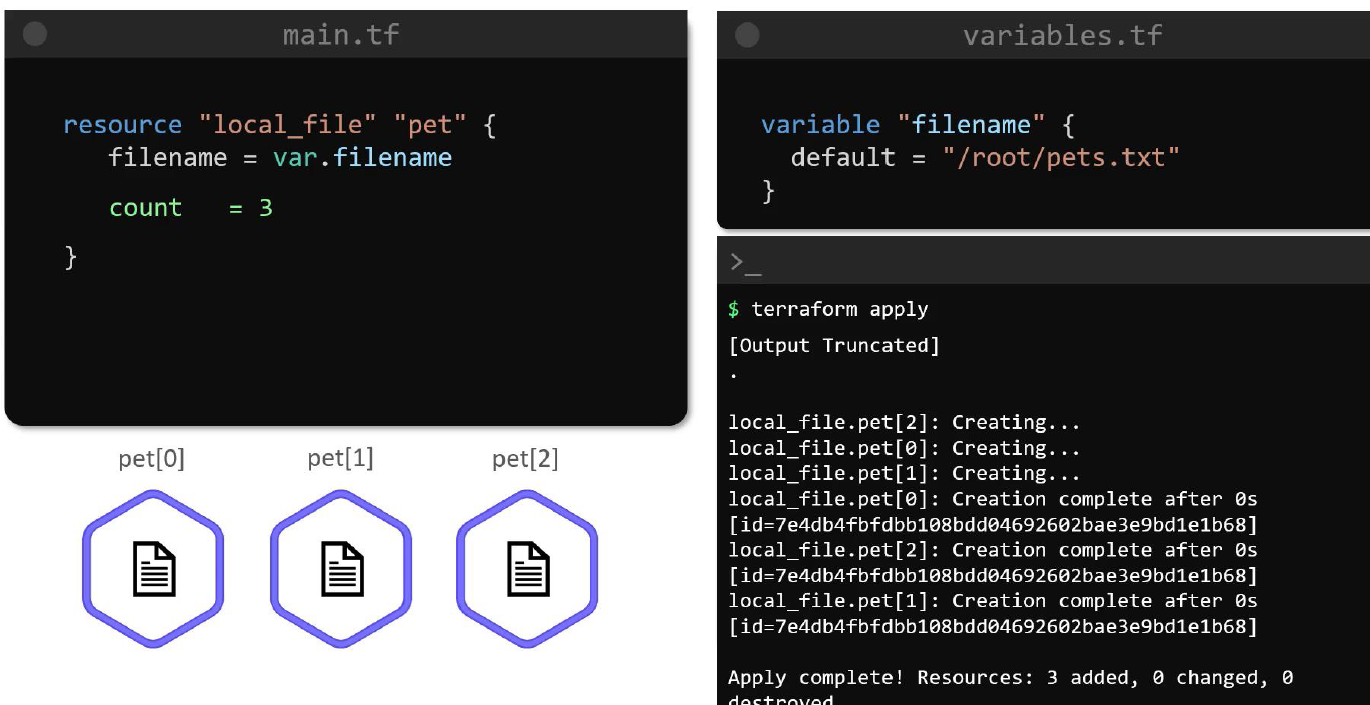
**Count: Meta Argument**

One of the easiest ways to create multiple instances of the local file is to make use of the count meta argument, to do this simply add an argument called count with a value greater than one.

Here we have used count is equal to three, when we try to run TerraForm plan now, we can see terraform tries to create three resources instead of one.



In the output of the terraform apply command, we can see that three resources are created.

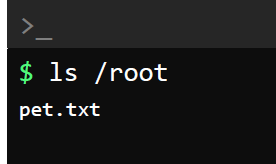


The resources are identified by pet[0], pet[1] and pet[2].

The resource is now considered to be a list of resources with elements at index zero, one and two.

However, there's one problem with this approach.

Since we have only specified the count, terraform will tried to create the same resource three times



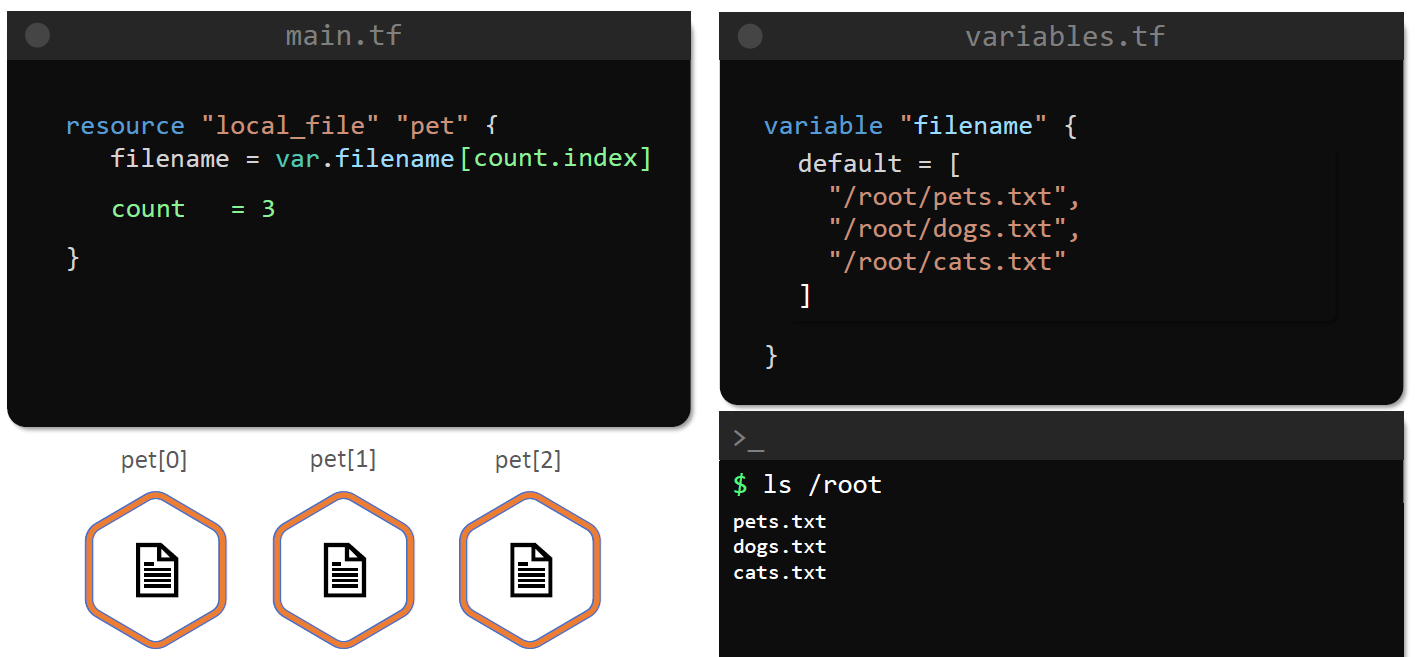
Since the file name is not unique terraform recreate the same file three times, rather than creating three separate files, which defeats the purpose of this task.

A better way to do this and make sure that all three resources have unique names is to make use of a list type variable for filename. And to do this we have used default values with three elements, each corresponding to the name of the file that we want to create.

Next, we want terraform to make use of each element of this list as the value of the filename argument.

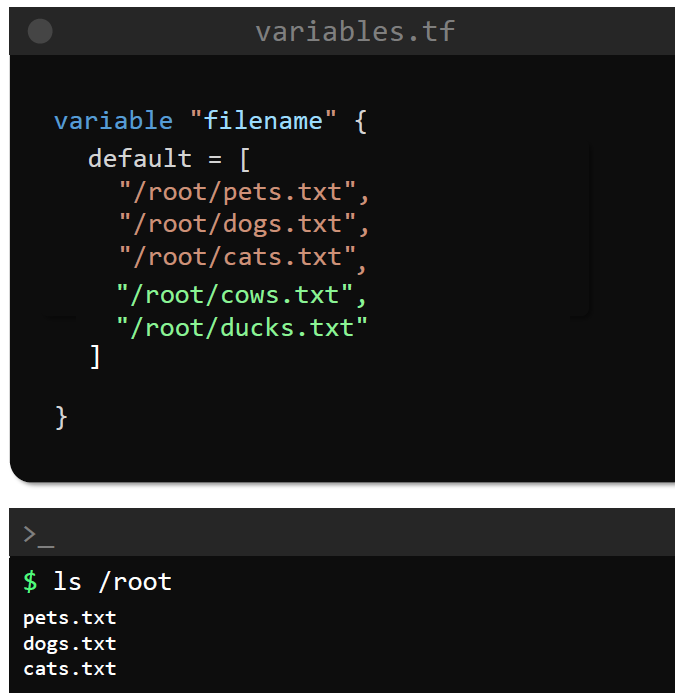
In this example, terraform should make three iterations, as the count has a value of three.

To use this within the configuration file, we can make use of **count.index** in the expression for the file name like this.



But what if we were to add a few more elements to the list in the future, say we wanted to add a few more files by the name of /root/cow.txt and /root/ducks.txt

If you were to apply this configuration now, we would see that it will still create only three files because we have set the count to a static value of three.



We want the count to automatically pick up the number of items that are defined within the filename variable and to do this, we can set the value of count to use a Built-In function that will return the length of the list.

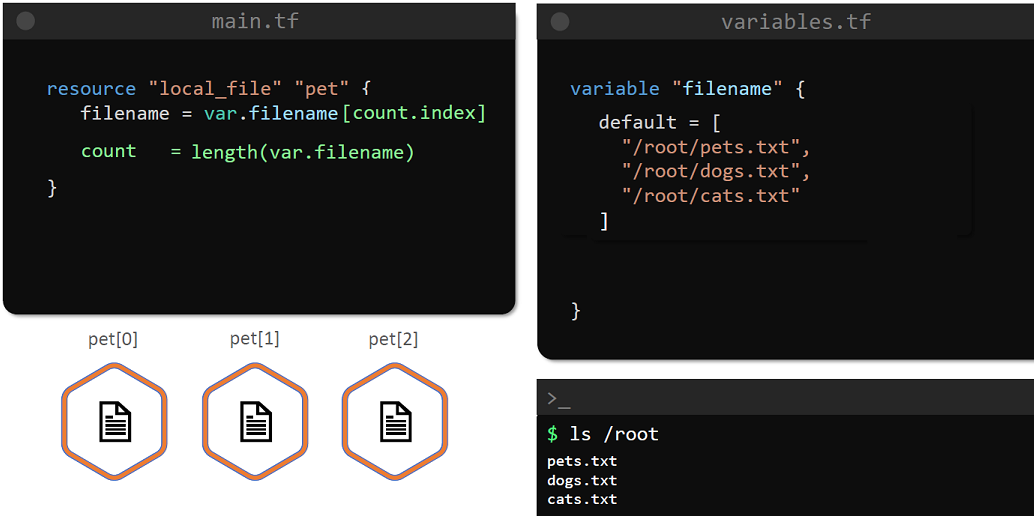
The length function is used to calculate the size of a list, and we can use this function in the count meta argument to dynamically determine the size of the variable. It will set the value of count to five.



Terraform has several built in functions that allows us to manipulate values within expressions.

But before we do that, let's change the default value for the filename variable back to three elements. And that's it.

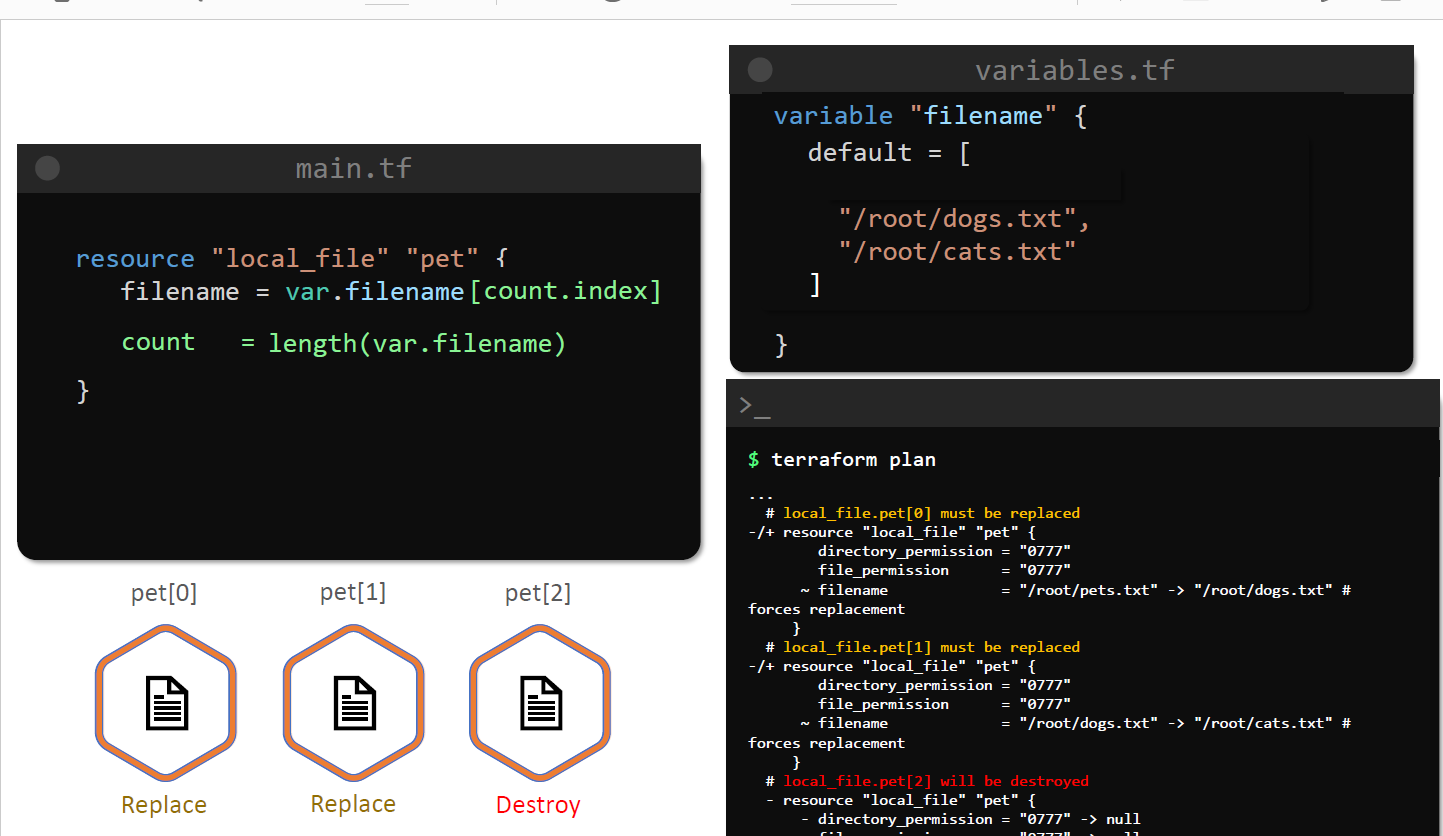
If you run terraform plan and apply now, we should see that three resources with distinct file names have been created.



There is, however, another significant drawback when we use the count meta argument to local variables this way

to illustrate this, let us use the same example, but this time, let us remove the element “/root/pets.txt” from the list.

If we apply terraform plan now, we can see that instead of deleting just one resource with the filename /root/pets.txt, terraform is replacing two resources and deleting one resource.



Why does it do that?

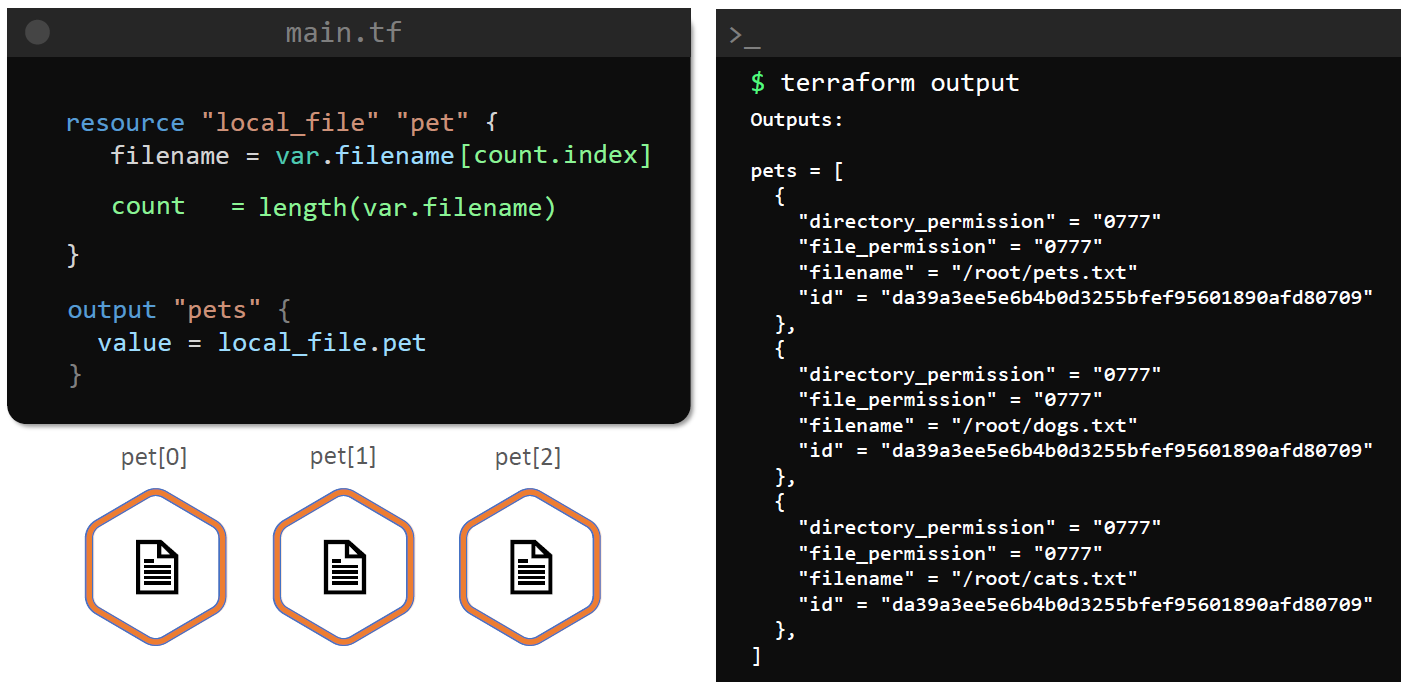
We only want to remove the first element in this list.

And it looks like all elements are going to be replaced by this operation.

To see why this is happening, let us first understand how it works.

As we saw before, when we use count, the resources become a list of resources.

To see this using terraform, let us add an output variable to main.tf file to print all details of the resource using the TerraForm output command.



**We can see that the resource is now in the format of a list**. Originally, the resource called pets is a list with three resources, each identified by its index.

The first resource in the list creates the file by the name pets.txt, and it is identified by pet[0]

The second resource element pet[1], which creates a file called dogs.txt.

And the third resource is pet[2], which creates a file called cats.txt.

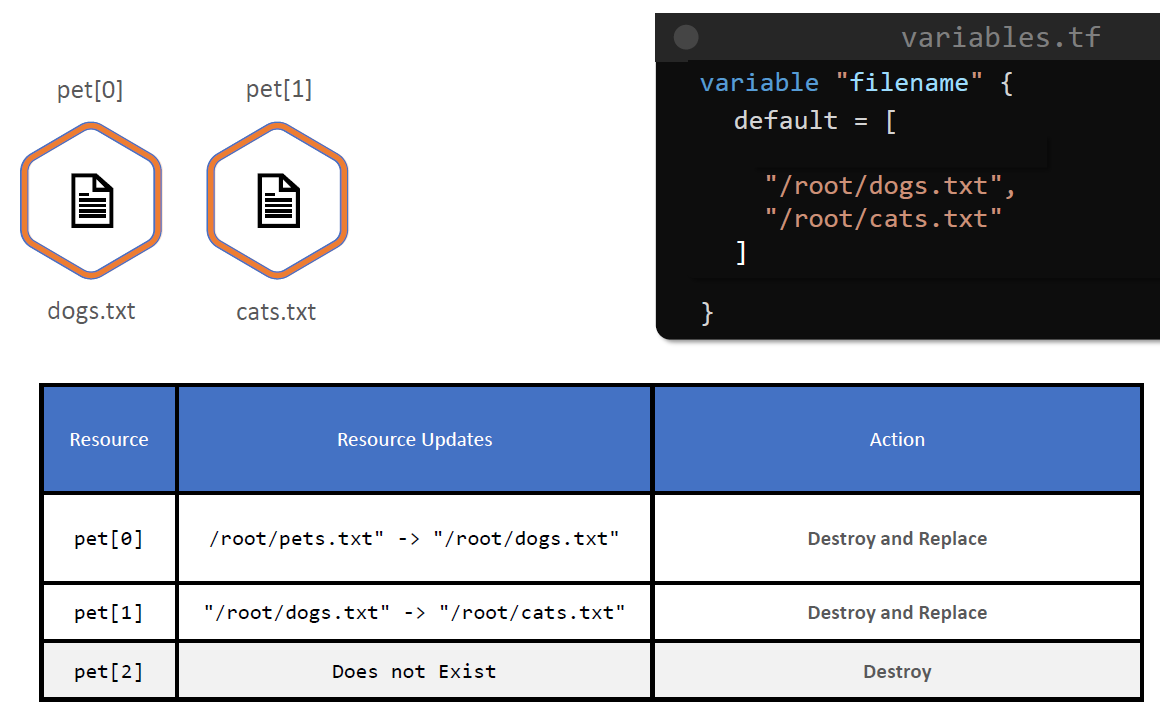
And as a result, when we deleted the element called /root/pets.txt, which was at index zero, the element with the value /root/dogs.txt shifts up and takes its place at index zero.

Likewise /root/cats.txt becomes the element at index one.

And the list now only has two elements in it, when we run terraform plan now, terraform can see that the resources at pet[0] and pet[1] have to be destroyed and replaced.

This is owing to the change in their file names, and there is no resource pets.txt, so it will delete that resource entirely.





Although after the apply operation, we will have the resources created, as per our intended instead, this is not an ideal approach.

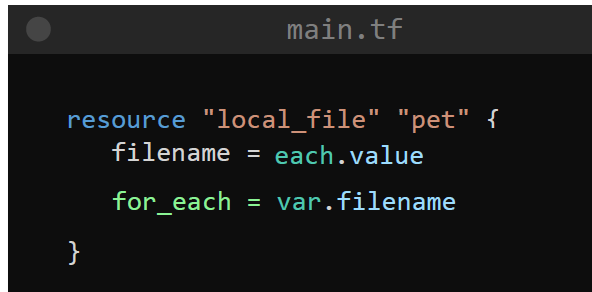
We may not want the resources to be destroyed and recreated just because we removed an unrelated element from the list.

And we will now see how to fix that.

**for\_each: Meta Argument**

Previously, we saw that when we use count, the resources are created as a list and this can have undesirable results when updating them.

One way to overcome this is to make use of the **for\_each** argument instead of count and to set the value of filename to each element in the list, we can make use of the **each.value**

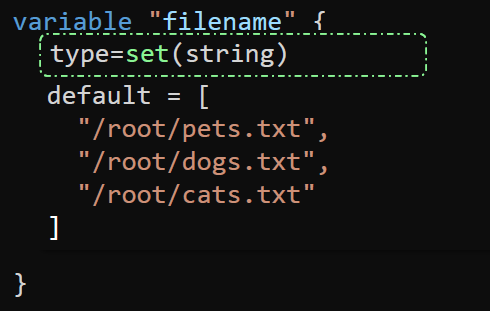


However, there's a catch.

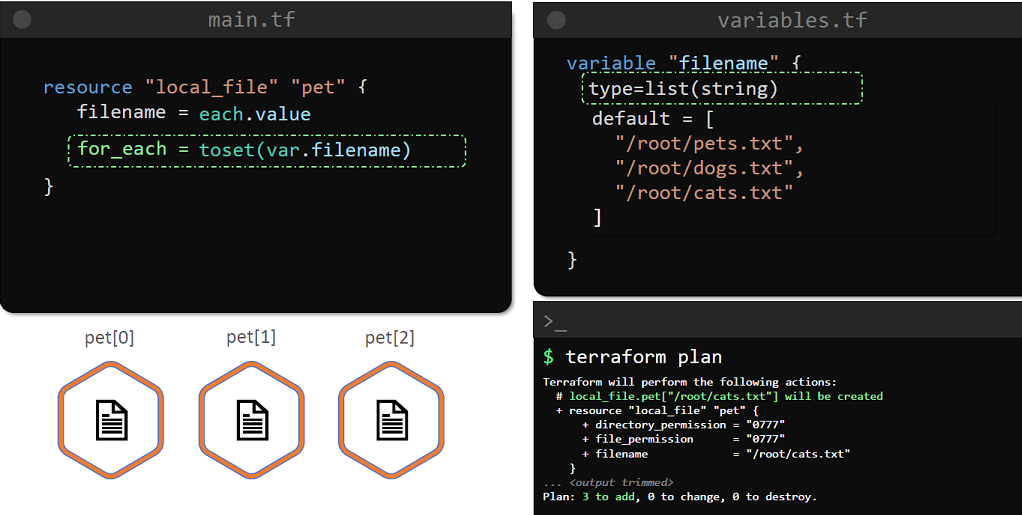
If we're on the reform plan now, we will see an error, **the for\_each argument only works with a map or a set,** in the variables.tf file we are currently making use of a list containing string elements.

There are a couple of ways to fix this.

* Either change the variable called filename to be type **set**,



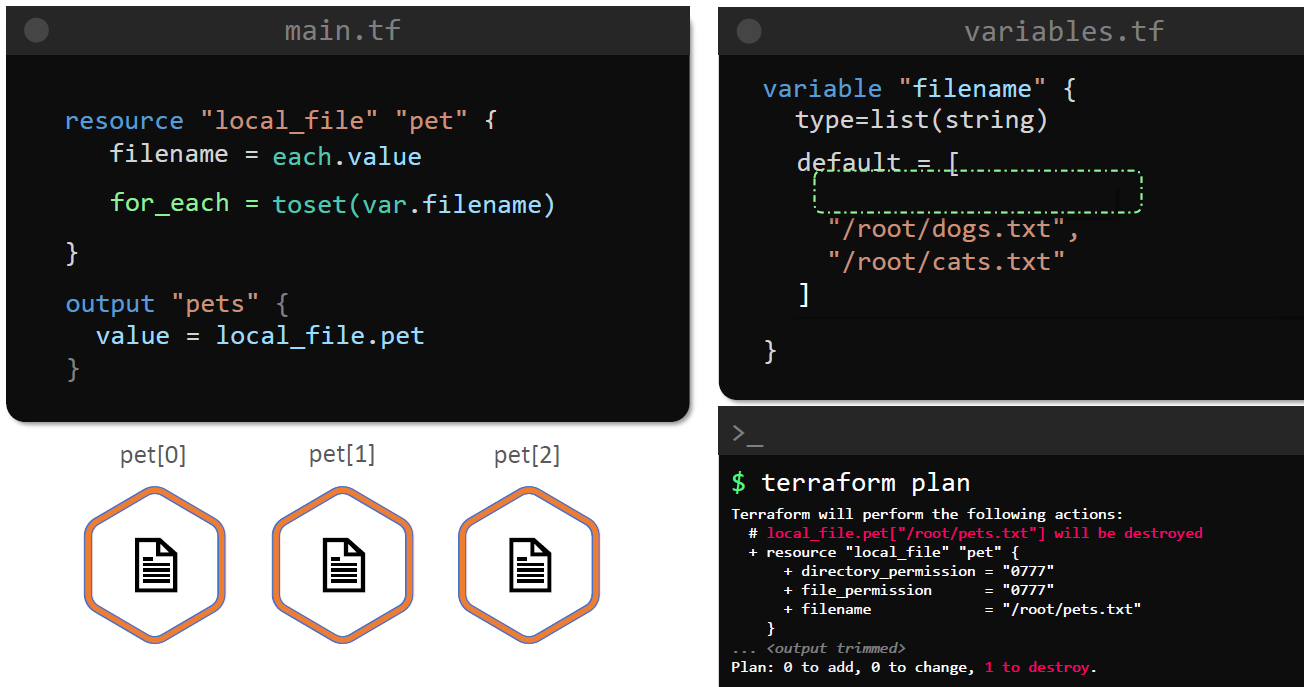
* Another way to fix this error while retaining the variable type as a list is to make use of another built in function, this time, we'll make use of the **toset** **function**, which will convert the variables from a list to a set.



Once this is done, terraform plan command should now work as expected.

Now let us replicate the same scenario as the one we did earlier with the count meta argument and delete the first element with the value /root/pets.txt from the list.

When we are on terraform plan now, we can see that only one resource is set to be destroyed.



The file with the name pets.txt, the other resources will be untouched. To see how this is working, let us create an output variable called pets to print the resource details like we did with the example using count.

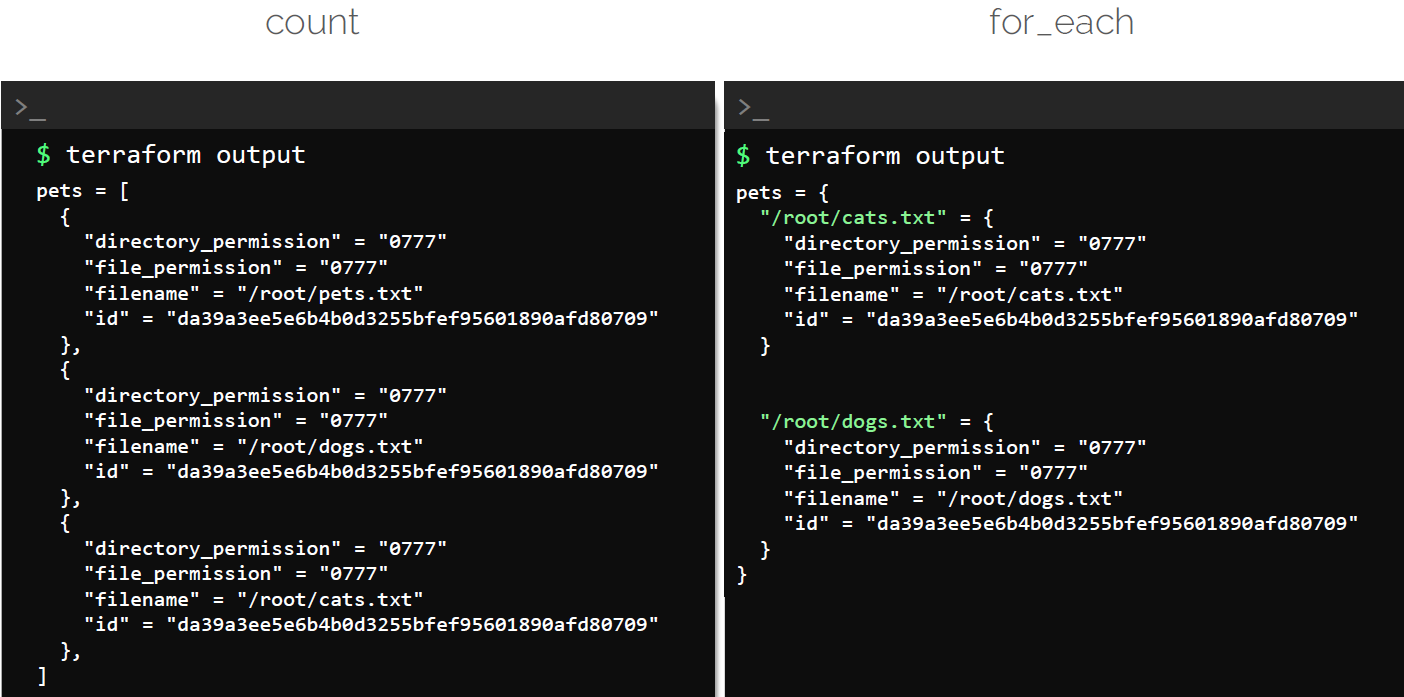


From the TerraForm output command, we can now see that the **resources are stored as a map and not a list.**

When we use for\_each instead of count, the resources are created as a map and not a list.

This means that the resources are no longer identified by the index, thereby bypassing the issues that we saw when we use count.

They are now identified by the keys, which are filenames, /root/cats.txt, /root/dogs.txt which are used by the for each argument in the configuration file.



**Version Constraints**

We will now see how to make use of specific provider version in terraform.

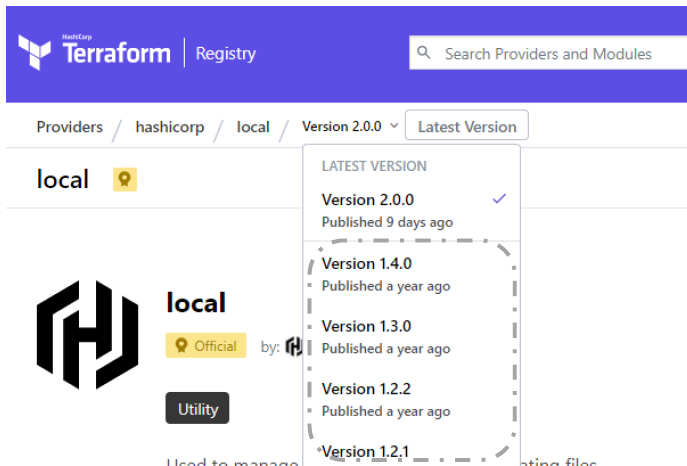
We saw earlier that providers use a plugin based architecture and that most of the popular ones are available in the public terraform registry.

By default “INIT” command downloads the latest version of the provider plugins that are needed by the configuration files.

However, this is not something that we may decide every time, the functionality of a provider plugin may vary drastically from one version to another, our terraform configuration may not work as expected when using a version different than the one it was written in.

Fortunately, we can make sure that a specific version of the provider is used by terraform when we run the terraform init command.

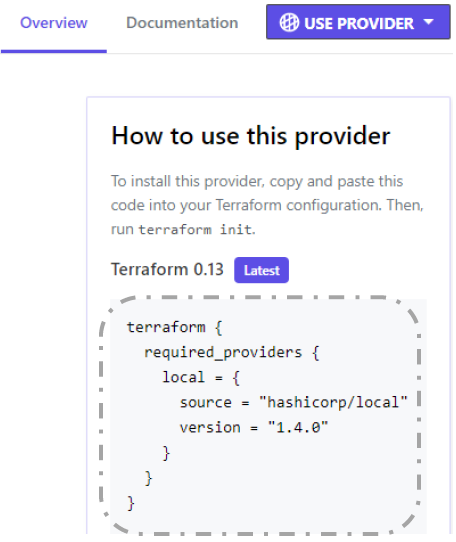
The instructions to use a specific version of a provider is available in the provided documentation in the registry.



Here we are making use of a **new block called terraform**, which is used to configure settings related to terraform itself.

To make use of a specific version of the provider, we need to make use of another **block called required\_providers** inside the terraform block.

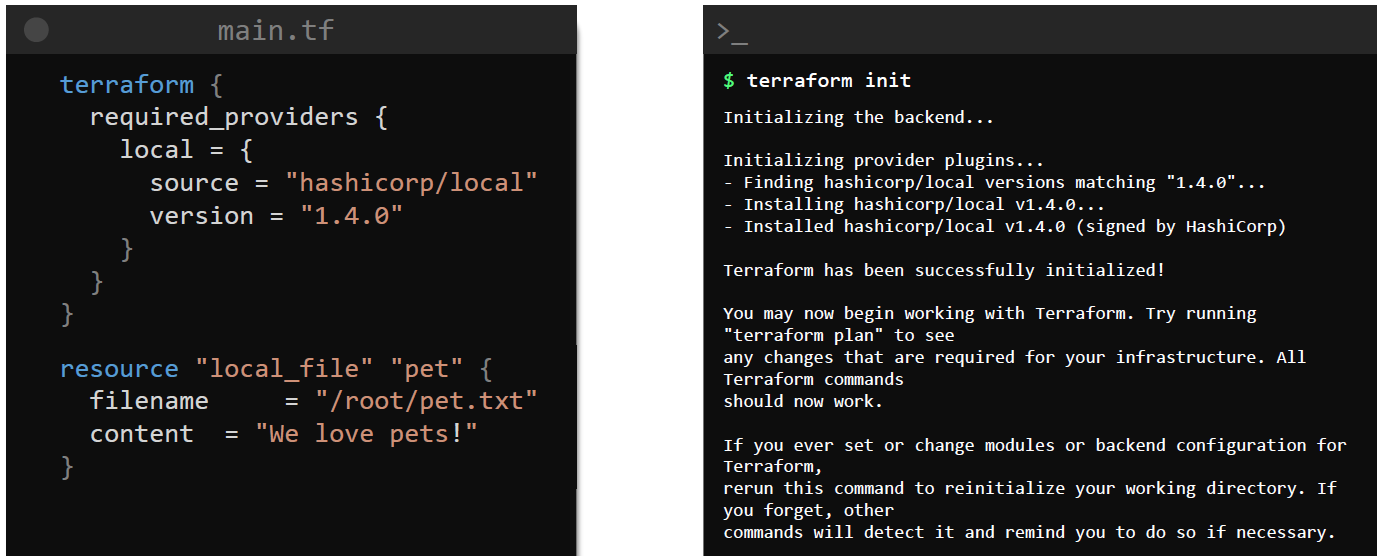
And inside the required\_providers block, we can have multiple arguments for every provider that we want to use.



In this example, we have one argument with the key call local for the local provider.

The value for this argument is an object with the source address of the provider and the exact version that we want to install, which in this case is 1.4.0.

When we run terraform init, we should see a message like this.



In the configuration for the local provider, we have specified version equal to “1.4.0”, this allows terraform to find and download this exact version of the local provider.

However, there are other ways to use the version constraint.

**version = “!=2.0.0”** not equal to symbol, terraform will ensure that this specific version is not downloaded

**version = “> 1.1.0”** greater than,make use of a version greater than a specific version, we can make use of the greater than operator

**version = “< 2.1.0”** smaller than,make use of a version smaller than a specific version, we can make use of the smaller than operator

**version = “> 1.2.0, < 2.0.0, != 1.4.0”** We can also combine the comparison operators like this to make use of a specific vision within our range, in this example, we want to make use of any version greater than “1.2.0” but lesser than ”2.0.0”, but also not the version “1.4.0” specifically.

And finally, we can also make use of pessimistic constraint operators.

This is defined by making use of the tilde greater than symbol like this.

**version = “~> 1.2”** This operator allows terraform to download the specific version or any available incremental version based on the value we provided. For example, here we have given the value “1.2” to following the tilde greater than symbol,this means that terraform can either download the version 1.2 or incremental version, such as 1.3, 1.4, 1.5 all the way up until 1.9.

**version = “~> 1.2.0”** This time, terraform can download the version 1.2.0, or 1.2.1, or 1.2.2 all the way up until 1.2.9.

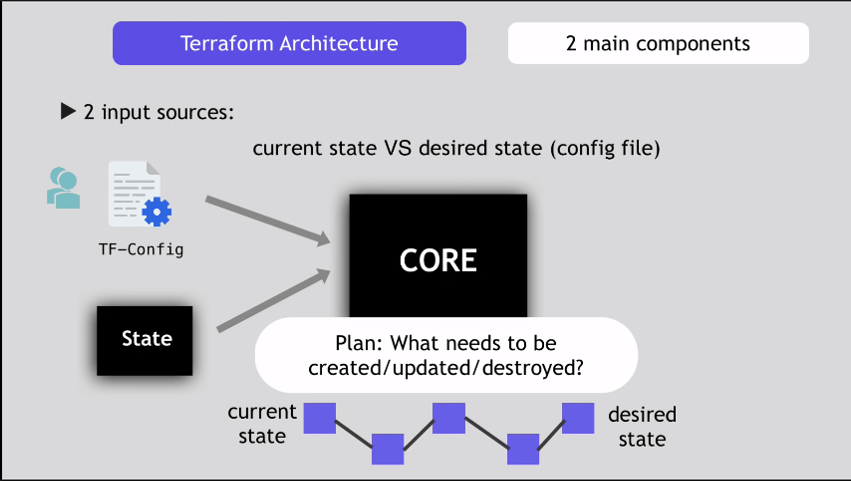
so **how does terraform do all this**. How does terraform actually connect to this infrastructure provider platforms and use all these technologies to provision stuff so for example how does terraform connect to AWS to create virtual space start ec2 instances configure networking etc

in order to do the job terraform has two main components that make up its architecture the

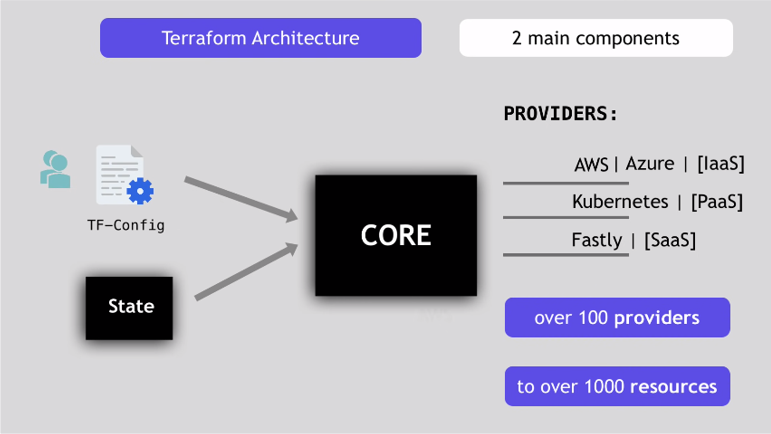
* first one is **terraform core** and the core uses two input sources in order to do its job so we take terraform configuration that you as a user write and where you defined what needs to be created or provisioned

* and the second one is **terraform state** where Terraform keeps the up-to-date state of how the current set up of the infrastructure looks like

so what core then does is it takes this input and it figures out the plan of what needs to be done so it compares the state what is the current state, what is the configuration that you desire the end result as I mentioned at the beginning and compares that and when it sees there is a difference so you want something else than what the current state is it figures out what needs to be done to get to that desired state in the configuration file so what needs to be created what needs to be updated deleted in which order on that infrastructure setup



and the second component or the second part of the architecture are providers for specific technologies this could be cloud providers like AWS measurer or other infrastructure as a service platforms so for the infrastructure level tasks but terraform



as I mentioned is also providers for

more high-level components like kubernetes or other platform as-a-service tools even some software is a self-service tool so it gives you possibility to create stuff on different levels like create a AWS infrastructure then deploy or create kubernetes on top of it and then create services inside that or components inside that kubernetes cluster so it gives you all these possibilities and it does that through those providers terraform has over hundred providers for these different technologies and each provider then gives terraform user access to its resources

so once the core creates an execution plan based on the input from config file and state it then uses providers for specific technologies to execute the plane to connect to those platforms and to actually carry out those execution steps

