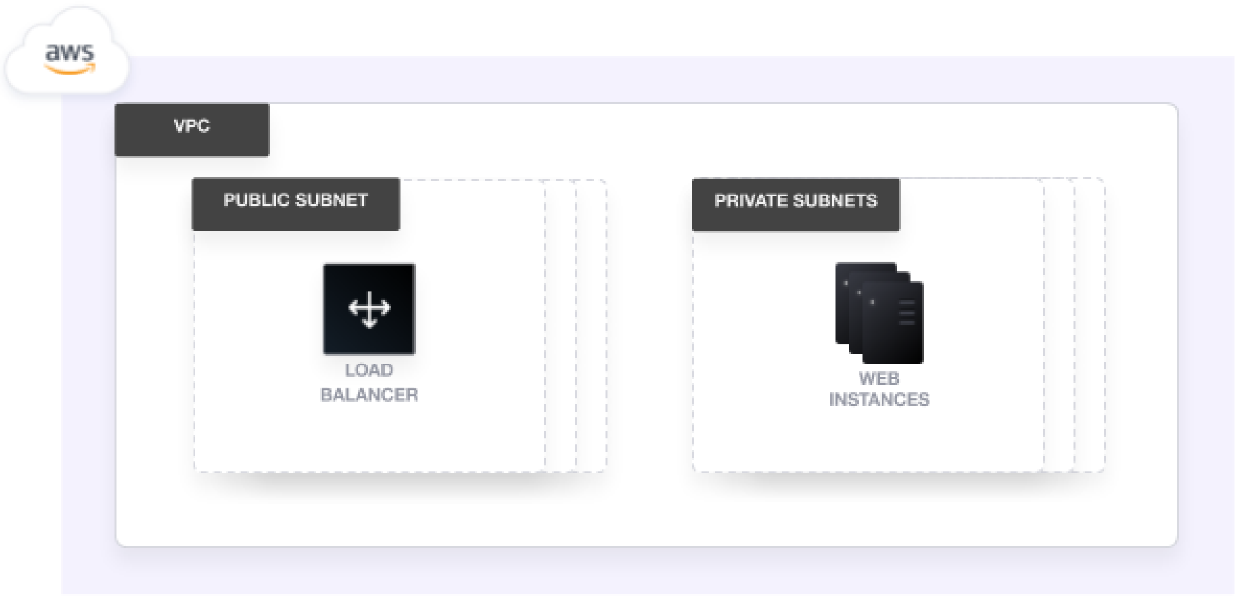
Customize Terraform Configuration with Variables

developer.hashicorp.com/terraform/tutorials/certification-associate-tutorials/variables

Input variables make your Terraform configuration more flexible by defining values that your end users can assign to customize the configuration. They provide a consistent interface to change how a given configuration behaves.

Unlike variables found in programming languages, Terraform's input variables don't change values during a Terraform run such as plan, apply, or destroy. Instead, they allow users to more safely customize their infrastructure by assigning different values to the variables before execution begins, rather than editing configuration files manually.

In this tutorial, you will use Terraform to deploy a web application on AWS. The supporting infrastructure includes a VPC, load balancer, and EC2 instances. You will parameterize this configuration with Terraform input variables. Finally, you will interpolate variables into strings, use variables with functions, and use variable validation.



Prerequisites

In order to follow this tutorial, you will need the following:

- The Terraform CLI, version 0.13 or later.
- AWS Credentials configured for use with Terraform.
- The git CLI.

Note: Some of the infrastructure in this tutorial may not qualify for the AWS free tier. Destroy the infrastructure at the end of the guide to avoid unnecessary charges. We are not responsible for any charges that you incur.

Create infrastructure

Clone the Learn Terraform variables GitHub repository for this tutorial.

```
$ git clone https://github.com/hashicorp/learn-terraform-variables.git
```

Change to the repository directory.

```
$ cd learn-terraform-variables
```

The configuration in `main.tf` defines a web application, including a VPC, load balancer, and EC2 instances.

Initialize this configuration.

```
$ terraform init
```

Now apply the configuration.

```
$ terraform apply
```

Respond to the confirmation prompt with a `yes` to create the example infrastructure.

Parameterize your configuration

Variable declarations can appear anywhere in your configuration files. However, we recommend putting them into a separate file called `variables.tf` to make it easier for users to understand how the configuration is meant to be customized.

To parameterize an argument with an input variable, you will first define the variable in `variables.tf`, then replace the hardcoded value with a reference to that variable in your configuration.

Add a block declaring a variable named `aws_region` to `variables.tf`.

```
variable "aws_region" {
  description = "AWS region"
  type        = string
  default     = "us-west-2"
}
```

Variable blocks have three optional arguments.

- **Description**: A short description to document the purpose of the variable.
- **Type**: The type of data contained in the variable.
- **Default**: The default value.

We recommend setting a description and type for all variables, and setting a default value when practical.

If you do not set a default value for a variable, you must assign a value before Terraform can apply the configuration. Terraform does not support unassigned variables. You will see some of the ways to assign values to variables later in this tutorial.

Variable values must be literal values, and cannot use computed values like resource attributes, expressions, or other variables.

You can refer to variables in your configuration with `var.<variable_name>`.

Edit the provider block in `main.tf` to use the new `aws_region` variable.

```
 provider "aws" {
-  region  = "us-west-2"
+  region  = var.aws_region
 }
```

Add a declaration for the `vpc_cidr_block` variable to `variables.tf`.

```
variable "vpc_cidr_block" {
  description = "CIDR block for VPC"
  type        = string
  default     = "10.0.0.0/16"
}
```

Now, replace the hard-coded value for the VPC's CIDR block with a variable in `main.tf`.

```
 module "vpc" {
   source  = "terraform-aws-modules/vpc/aws"
   version = "2.66.0"

-  cidr = "10.0.0.0/16"
+  cidr = var.vpc_cidr_block
   ## ...
 }
```

Apply the updated configuration. Since the default values of these variables are the same as the hard-coded values they replaced, Terraform will perform no changes.

```
$ terraform apply
```

Since the apply will make no changes to your resources, you can either respond to the confirmation prompt with a `yes`, or cancel the apply.

Set the number of instances

Terraform supports several variable types in addition to `string`.

Use a `number` type to define the number of instances supported by this configuration. Add the following to `variables.tf`.

```
variable "instance_count" {
  description = "Number of instances to provision."
  type        = number
  default     = 2
}
```

Update EC2 instances to use the `instance_count` variable in `main.tf`.

```
 module "ec2_instances" {
   source = "./modules/aws-instance"

-  instance_count = 2
+  instance_count = var.instance_count
   ## ...
 }
```

When Terraform interprets values, either hard-coded or from variables, it will convert them into the correct type if possible. So the `instance_count` variable would also work using a string (`"2"`) instead of a number (`2`). We recommend using the most appropriate type in variable definitions to helps users of your configuration know the appropriate data type to use, as well as to catch configuration errors early.

Toggle VPN gateway support

In addition to strings and numbers, Terraform supports several other variable types. A variable with type `bool` represents true/false values.

Use a `bool` type variable to control whether your VPC is configured with a VPN gateway. Add the following to `variables.tf`.

```
variable "enable_vpn_gateway" {
  description = "Enable a VPN gateway in your VPC."
  type        = bool
  default     = false
}
```

Use this new variable in your VPC configuration by editing `main.tf` as follows.

```
 module "vpc" {
   source  = "terraform-aws-modules/vpc/aws"
   version = "2.44.0"

   ## ...

   enable_nat_gateway = true
-  enable_vpn_gateway = false
+  enable_vpn_gateway = var.enable_vpn_gateway
   ## ...
 }
```

Leave the value for `enable_nat_gateway` hard-coded. In any configuration, there may be some values that you want to allow users to configure with variables and others you don't.

When you write Terraform modules you intend to re-use, you will usually want to make as many attributes configurable with variables as possible, to make your module more flexible for use in more situations.

When you write Terraform configuration for a specific project, you may choose to leave some attributes with hard-coded values when it doesn't make sense to allow users to configure them.

List public and private subnets

The variables you have used so far have all been single values. Terraform calls these types of variables *simple*. Terraform also supports *collection* variable types that contain more than one value. Terraform supports several collection variable types.

- **List:** A sequence of values of the same type.
- **Map:** A lookup table, matching keys to values, all of the same type.
- **Set:** An unordered collection of unique values, all of the same type.

In this tutorial, you will use lists and a map, which are the most commonly used of these types. Sets are useful when a unique collection of values is needed, and the order of the items in the collection does not matter.

A likely place to use list variables is when setting the `private_subnets` and `public_subnets` arguments for the VPC. Make this configuration easier to use while still being customizable by using lists along with the `slice()` function.

Add the following variable declarations to `variables.tf`.

```
variable "public_subnet_count" {
  description = "Number of public subnets."
  type        = number
  default     = 2
}

variable "private_subnet_count" {
  description = "Number of private subnets."
  type        = number
  default     = 2
}

variable "public_subnet_cidr_blocks" {
  description = "Available cidr blocks for public subnets."
  type        = list(string)
  default     = [
    "10.0.1.0/24",
    "10.0.2.0/24",
    "10.0.3.0/24",
    "10.0.4.0/24",
    "10.0.5.0/24",
    "10.0.6.0/24",
    "10.0.7.0/24",
    "10.0.8.0/24",
  ]
}

variable "private_subnet_cidr_blocks" {
  description = "Available cidr blocks for private subnets."
  type        = list(string)
  default     = [
    "10.0.101.0/24",
    "10.0.102.0/24",
    "10.0.103.0/24",
    "10.0.104.0/24",
    "10.0.105.0/24",
    "10.0.106.0/24",
    "10.0.107.0/24",
    "10.0.108.0/24",
  ]
}
```

Notice that the type for the list variables is `list(string)`. Each element in these lists must be a string. List elements must all be the same type, but can be any type, including complex types like `list(list)` and `list(map)`.

Like lists and arrays used in most programming languages, you can refer to individual items in a list by index, starting with 0. Terraform also includes several functions that allow you to manipulate lists and other variable types.

Use the `slice()` function to get a subset of these lists.

The Terraform `console` command opens an interactive console that you can use to evaluate expressions in the context of your configuration. This can be very useful when working with and troubleshooting variable definitions.

Open a console with the `terraform console` command.

```
$ terraform console
>
```

Now use the Terraform console to inspect the list of private subnet blocks.

Refer to the variable by name to return the entire list.

```
> var.private_subnet_cidr_blocks
tolist([
  "10.0.101.0/24",
  "10.0.102.0/24",
  "10.0.103.0/24",
  "10.0.104.0/24",
  "10.0.105.0/24",
  "10.0.106.0/24",
  "10.0.107.0/24",
  "10.0.108.0/24",
])
```

Retrieve the second element from the list by index with square brackets.

```
> var.private_subnet_cidr_blocks[1]
"10.0.102.0/24"
```

Now use the `slice()` function to return the first three elements from the list.

```
> slice(var.private_subnet_cidr_blocks, 0, 3)
tolist([
  "10.0.101.0/24",
  "10.0.102.0/24",
  "10.0.103.0/24",
])
```

The `slice()` function takes three arguments: the list to slice, the start index, and the end index (exclusive). It returns a new list with the specified elements copied ("sliced") from the original list.

Leave the console by typing `exit` or pressing `Control-D`.

```
> exit
```

Now use the slice function to extract a subset of the cidr block lists in `main.tf` when defining your VPC's public and private subnet configuration.

```
 module "vpc" {
   source  = "terraform-aws-modules/vpc/aws"
   version = "2.44.0"

   cidr = "10.0.0.0/16"

   azs             = data.aws_availability_zones.available.names
-  private_subnets = ["10.0.101.0/24", "10.0.102.0/24"]
-  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24"]
+  private_subnets = slice(var.private_subnet_cidr_blocks, 0,
var.private_subnet_count)
+  public_subnets  = slice(var.public_subnet_cidr_blocks, 0,
var.public_subnet_count)
   ## ...
 }
```

This way, users of this configuration can specify the number of public and private subnets they want without worrying about defining CIDR blocks.

Map resource tags

Each of the resources and modules declared in `main.tf` includes two tags: `project_name` and `environment`. Assign these tags with a `map` variable type.

Declare a new `map` variable for resource tags in `variables.tf`.

```
variable "resource_tags" {
  description = "Tags to set for all resources"
  type        = map(string)
  default     = {
    project     = "project-alpha",
    environment = "dev"
  }
}
```

Setting the type to `map(string)` tells Terraform to expect strings for the values in the map. Map keys are always strings. Like dictionaries or maps from programming languages, you can retrieve values from a map with the corresponding key. See how this works with the Terraform console.

Start the console.

```
$ terraform console
>
```

Retrieve the value of the `environment` key from the `resource_tags` map.

```
> var.resource_tags["environment"]
"dev"
```

Leave the console by typing `exit` or pressing `Control-D`.

```
> exit
```

Note: The `terraform console` command loads your Terraform configuration only when it starts. Be sure to exit and restart the console to pick up your most recent changes.

Now, replace the hard coded tags in `main.tf` with references to the new variable.

```
-  tags = {
-    project     = "project-alpha",
-    environment = "dev"
-  }
+  tags = var.resource_tags

## ... replace all five occurrences of `tags = {...}`
```

The hard-coded tags are used five times in this configuration, be sure to replace them all.

Apply these changes.

```
$ terraform apply
```

Since the value of the `project` tag has changed, there will be changes to apply. Respond to the prompt with `yes` to apply them.

Lists and maps are *collection* types. Terraform also supports two *structural* types. Structural types have a fixed number of values that can be of different types.

- **Tuple:** A fixed-length sequence of values of specified types.
- **Object:** A lookup table, matching a fixed set of keys to values of specified types.

Assign values to variables

Terraform requires that every variable be assigned a value. Terraform supports several ways to assign variable values.

Assign values when prompted

In the examples so far, all of the variable definitions have included a default value. Add a new variable without a default value to `variables.tf`.

```
variable "ec2_instance_type" {
  description = "AWS EC2 instance type."
  type        = string
}
```

Replace the reference to the EC2 instance type in `main.tf`.

```
 module "ec2_instances" {
   source = "./modules/aws-instance"

   instance_count = var.instance_count
-  instance_type  = "t2.micro"
+  instance_type  = var.ec2_instance_type
   ## ...
 }
```

Apply this configuration now.

```
$ terraform apply
var.ec2_instance_type
  AWS EC2 instance type.

  Enter a value: t2.micro

random_string.lb_id: Refreshing state... [id=3Bn]
module.vpc.aws_vpc.this[0]: Refreshing state... [id=vpc-06083988d1c86b823]

## ...

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

Outputs:

public_dns_name = "lb-3Bn-project-alpha-dev-542453763.us-west-2.elb.amazonaws.com"
```

Terraform will prompt you for a value for the new variable. Enter `t2.micro`.

Since the value you entered is the same as the old value, there will be no changes to apply.

Assign values with a `terraform.tfvars` file

Whenever you execute a plan, destroy, or apply with any variable unassigned, Terraform will prompt you for a value. Entering variable values manually is time consuming and error prone, so Terraform provides several other ways to assign values to variables.

Create a file named `terraform.tfvars` with the following contents.

```
resource_tags = {
  project     = "new-project",
  environment = "test",
  owner       = "me@example.com"
}

ec2_instance_type = "t3.micro"

instance_count = 3
```

Terraform automatically loads all files in the current directory with the exact name `terraform.tfvars` or matching `*.auto.tfvars`. You can also use the `-var-file` flag to specify other files by name.

These files use syntax similar to Terraform configuration files (HCL), but they cannot contain configuration such as resource definitions. Like Terraform configuration files, these files can also contain JSON.

Apply the configuration with these new values.

```
$ terraform apply
```

Respond to the confirmation prompt with `yes` to apply these changes.

You can also set input variables via environment variables and command line flags. Check out the documentation for more details. If there are different values assigned for a variable through these methods, Terraform will use the last value it finds, in order of precedence.

Interpolate variables in strings

Terraform configuration supports string interpolation — inserting the output of an expression into a string. This allows you to use variables, local values, and the output of functions to create strings in your configuration.

Update the names of the security groups to use the project and environment values from the `resource_tags` map.

```
 module "app_security_group" {
   source  = "terraform-aws-modules/security-group/aws//modules/web"
   version = "3.12.0"

-   name        = "web-sg-project-alpha-dev"
+   name        = "web-
sg-${var.resource_tags["project"]}-${var.resource_tags["environment"]}"

   ## ...
 }

 module "lb_security_group" {
   source  = "terraform-aws-modules/security-group/aws//modules/web"
   version = "3.12.0"

-   name        = "lb-sg-project-alpha-dev"
+   name        = "lb-
sg-${var.resource_tags["project"]}-${var.resource_tags["environment"]}"

   ## ...
 }

 module "elb_http" {
   source  = "terraform-aws-modules/elb/aws"
   version = "2.4.0"

   # Ensure load balancer name is unique
-   name = "lb-${random_string.lb_id.result}-project-alpha-dev"
+   name =
"lb-${random_string.lb_id.result}-${var.resource_tags["project"]}-${var.resource_t
ags["environment"]}"
   ## ...
 }
```

Validate variables

This configuration has a potential problem. AWS load balancers have naming restrictions. They must be no more than 32 characters long, and can only contain a limited set of characters.

One way to deal with this is to use variable validation to restrict the possible values for the project and environment tags.

Replace your existing `resource tags` variable in `variables.tf` with the below code snippet, which includes validation blocks to enforce character limits and character sets on both `project` and `environment` values.

```
variable "resource_tags" {
  description = "Tags to set for all resources"
  type        = map(string)
  default     = {
    project     = "my-project",
    environment = "dev"
  }

  validation {
    condition     = length(var.resource_tags["project"]) <= 16 &&
length(regexall("[^a-zA-Z0-9-]", var.resource_tags["project"])) == 0
    error_message = "The project tag must be no more than 16 characters, and only
contain letters, numbers, and hyphens."
  }

  validation {
    condition     = length(var.resource_tags["environment"]) <= 8 &&
length(regexall("[^a-zA-Z0-9-]", var.resource_tags["environment"])) == 0
    error_message = "The environment tag must be no more than 8 characters, and
only contain letters, numbers, and hyphens."
  }
}
```

The `regexall()` function takes a regular expression and a string to test it against, and returns a list of matches found in the string. In this case, the regular expression will match a string that contains anything other than a letter, number, or hyphen.

This way the maximum length of the load balancer name will never exceed 32, and it will not contain invalid characters. Using variable validation can be a good way to catch configuration errors early.

Apply this change to add validation to these two variables. There will be no changes to apply, since your infrastructure configuration has not changed.

```
$ terraform apply
```

Now test the validation rules by specifying an environment tag that is too long. Notice that the command will fail and return the error message specified in the validation block.

```
$ terraform apply -var='resource_tags={project="my-
project",environment="development"}'

Error: Invalid value for variable

  on variables.tf line 69:
  69: variable "resource_tags" {

The environment tag must be no more than 8 characters, and only contain
letters, numbers, and hyphens.

This was checked by the validation rule at variables.tf:82,3-13.
```

Clean up your infrastructure

Now you have defined and used Terraform variables. By adding variables to your configuration, you have made it easier to make changes to your resources, such as adding more subnets or EC2 instances. Variables also make it easier for you to re-use this configuration for other projects, or turn it into a module that can be consumed by other Terraform configuration. You have also made this configuration more generic, allowing you to save project-specific configuration into the `terraform.tfvars` file.

Before moving on, destroy the infrastructure you created by running the `terraform destroy` command. Remember to confirm your destroy with a `yes`.

```
$ terraform destroy
```

Next steps

Now that you have seen how to define and use variables, check out the following resources for more information.

- Read the Input variables documentation.
- Read the Local variables documentation.
- Learn how to create and use Terraform modules.
- Learn how to validate modules with custom conditions.
- Read more about structural types in the documentation.