

Lab 4. VPC deployment with variables and iterations

Contents

Overview	1
Solution	1
Start Lab	2
Task 1. Load existing code	2
Task 2. Introducing variables	2
Task 3. Variables deep dive	4
Task4. Working with multiple tfvar files	8
Test 1: A single tfvars file declaring all variable values	8
Test 2. Using a single tfvars file to declare some but not all variable values	9
Test 3. Using a tfvars file other than terraform.tfvars to declare all values	9
Test 4. Using a tfvars file other than terraform.tfvars to declare some but not all values	9
Test 5. Using multiple optional tfvars files and terraform.tfvars (1)	10
Test 6. Using multiple optional tfvars files and terraform.tfvars (2)	10
Test 7. Using multiple tfvars files combined with -var options	11
Task 5. Working with auto-tfvars files	11
Task 6. Simple Iterations	13
Challenge	14
Clean-Up	14

Overview

This lab will modify existing code that has been used successfully to deploy a 'hard-coded' basic VPC architecture.

Solution

The solution to this lab can be found in [awslabs/solutions/04](#). Try to use this only as a last resort if you are struggling to complete the step-by-step processes.

Start Lab

1. Ensure you have completed Lab0 before attempting this lab.
2. In the IDE terminal pane, enter the following commands...
`cd ~/environment/awslabs/04`
3. This shifts your current working directory to awslabs/labs/04. Ensure all commands are executed in this directory
4. Close any open files and use the Explorer pane to navigate to and open the pre-configured main.tf file in awslabs/04.

Task 1. Load existing code

1. Run `terraform init`
2. Run `terraform apply`, entering `yes` when prompted. A total of 12 resources should be created.

Note: If you receive an error regarding the chosen availability zone then use the replace option (Ctrl+f) to change all occurrences of 'us-west-2a' to a zone available to your account as shown in the error message. Save and repeat the apply command.

If you have not completed the lab in which you created this deployment file then familiarize yourself with the components deployed using the console VPC and EC2 dashboards, selecting the US West2 (Oregon) region.

If you have completed the lab in which you created this deployment file, then you may have noticed that only 12 and not 14 resources have been created. The creation of an EIP and NAT gateway has been omitted here in order to speed up the modify/apply/destroy cycle we will be going through as we work through this lab.

3. Switch to the IDE and run `terraform destroy` and type `yes` to clear the hard-coded deployment

Task 2. Introducing variables

Notice how main.tf has all argument values explicitly declared; the region, the availability-zone, the subnet names etc. This is a 'hard-coded' file which limits its re-usability. We could duplicate, edit, and deploy a new version, but this is not optimal. Ideally, we should be able to re-use the same code-base but introduce variations as needed. This is where variables come into use.

Take a moment to review developer information regarding variables from Hashicorp at

<https://developer.hashicorp.com/terraform/language/values/variables>

1. Whilst variables can be declared in main.tf, best practice is to use dedicated tf files. Create a file to hold our variables...

`touch variable.tf`

2. Paste the following boiler-plate code into variable.tf

```
variable "" {  
  description = ""  
  type        =  
  default     = ""  
}
```

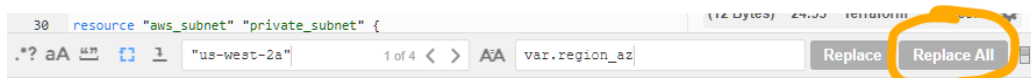
3. We will create a variable for the region we are going to deploy to. Modify variable.tf to set us-west-1 as our new region

```
variable "deployment_region" {  
  description = "Region of deployment"  
  type        = string  
  default     = "us-west-1"  
}
```

4. Save **variable.tf**
5. Navigate to the provider configuration block in main.tf (Lines 10-12)
6. Replace **"us-west-2"** (*removing the quotes too*) with **var.deployment_region** (no quotes). Note how variables are referenced by **var.** followed by the name of the variable, as defined in variable.tf
7. Save main.tf, run **terraform apply** and type **yes**
8. **Note the errors** regarding the availability-zones we are attempting to deploy resources to. The code specifies an availability zone that is not in the newly selected region. This shows how changing one argument may have a knock-on effect! Expect similar errors as we progress. Make note of the first availability zone available to your account in the us-west-1 region.
9. Switch to variable.tf and paste in the new block below. Populate the default value with the az noted in the previous step (for example "us-west-1a").

```
variable "region_az" {  
  description = "Availability Zone"  
  type        = string  
  default     = "{YOUR AZ HERE}"  
}
```

10. Run **terraform destroy** and type **yes** as terraform will not roll-back the partial deployment.
11. In **main.tf**, change the hard-coded value for the 4 availability_zone arguments from their current value to **var.region_az** (no quotes). The IDE has a find/replace facility to assist in this. Press Ctrl+F to open up Find and replace....



12. Save both files, run **terraform apply** and type **yes**. We again have errors, this time relating to the image id.

13. Run **terraform destroy** and type **yes**

14. Switch to variable.tf and paste in the new block below...

```
variable "instance_ami" {  
  description = "EC2 ami"  
  type        = string  
  default     = "ami-0036b4598ccd42565"  
}
```

15. In main.tf, change the hard-coded value for the 2 ami arguments from the current value "**ami-06e85d4c3149db26a**" to **var.instance_ami** The IDE has a find/replace facility.

16. Repeat this process for the `instance_type` argument value. Create a variable block named **ec2_instance_size** with a description of "**EC2 size**" and set the default value to "**t3.micro**". Don't forget to reference this in main.tf, there are 2 replacements needed.

17. Ensure any changes are saved, then run **terraform apply** and type **yes**

18. Use the console to verify that the deployment has succeeded in us-west-1 (N.California) and that your EC2 instances are size t2.micro

19. Run **terraform destroy** and type **yes** to remove the deployment

20. A copy of the modified main.tf and variable.tf are shown in the solution folder /awslabs/solutions/04 section at the end of this document should you have any issues. (Your default az may differ in variable.tf if us-west-1a was unavailable to your account so adjust accordingly)...

Task 3. Variables deep dive

We have removed some of the hard-coded entries from the main.tf file by replacing explicit argument values with variables, the values of which are now in a variables file, and we have configured these values for deploying to us-west-1. We would still have to modify the content of this variables file whenever we want to make changes to the values passed into main.tf Whilst more efficient than modifying the main.tf, this approach is still sub-optimal.

Any .tf file can contain variable declarations, and there may be multiple such files, provided they do not contain duplications.

1. To demonstrate this, create an empty file variable1.tf

```
touch variable1.tf
```

2. Paste in the following...

```
variable "deployment_region" {  
  description = "Region of deployment"  
  type        = string  
  default     = "us-west-2"
```

```

}
variable "region_az" {
    description = "Availability Zone"
    type        = string
    default     = "us-west-2a"
}

```

3. Save the file and run **terraform plan**
4. There will be errors relating to duplicate variable declaration.
5. Delete variable1.tf using **rm variable1.tf**
6. If a variable is referenced in a .tf file then that variable **must** be declared, but it need not necessarily have a default value assigned, as the next steps will demonstrate.
7. Comment out the first 5 lines of code in variable.tf using # and save the file...

```

1  #variable "deployment_region" {
2  #   description = "Region of deployment"
3  #   type        = string
4  #   default     = "us-west-1"
5  #}
6  |

```

8. Run **terraform plan** and note the error. A variable is being referenced that hasn't been declared.
9. Now uncomment the lines, except for line 4, the default value.
10. Save the file and run **terraform plan** again
11. You are now prompted for the region. Notice that the prompt script is the value of the description argument `Region of deployment`
12. Escape using **Ctrl+c** to escape without entering a region
13. Uncomment line 4 and save
14. One option for passing values into declared variables is to enter the values at the command line. This would suppress prompting and would also override the default value if one had been set.
15. In variable.tf, comment out the **ec2_instance_size** default line (line 22) and save the changes.
16. Run **terraform plan** to validate we get the prompt for ec2_instance_size.
17. Escape and run... **terraform plan -var "ec2_instance_size=t3.micro"**
18. The planning should now be completed without errors.
19. So, if a variable is referenced then it must be declared otherwise an error will occur. A variable value can be declared at the command line otherwise the default value of the variable is assumed. If there is no default value, then we are prompted for the value.
20. What if there is a default value and we declare a different value at the command line? Which takes precedence ? Let's see...

Running `terraform plan` gives a visual output of the proposed deployment. This output can also be piped to a file using the `-out` option. The file created is in binary format but can be converted to a json file using the `terraform show` command.

21. Run terraform plan with `-var` and `-out` options...

```
terraform plan -var "ec2_instance_size=t1.micro" -var "instance_ami=ami-0d50e5e845c552faf" -out=out.tfplan
```

22. Run terraform show with json options

```
terraform show -json out.tfplan > output.json  
nano output.json
```

23. Focus on the `ec2_instance_size` and `instance_ami` values as compared with those in `variable.tf`. (Reducing your browser zoom setting may help here) We supplied the instance size rather than being prompted, which would have otherwise happened given that a default size has not been declared in the variable file. We supplied the ami, overriding the default ami declared in the variable file.

24. Another method of providing argument values to variables is by using `.tfvar` files, which will be demonstrated shortly. We will also look at how precedence deals with potential variable value conflicts.

25. Prompting for the value of a variable is the lowest order of precedence and will only happen if the value is undeclared elsewhere....

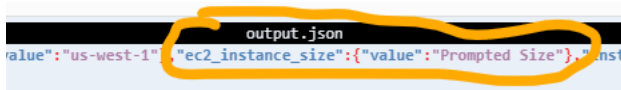
26. `ctrl+x` to exit nano and then run terraform plan `-out` without providing a `-var` option using `terraform plan -out=out.tfplan`

27. No default value exists, for instance size, so you are prompted. Enter **Prompted Size** Note this is not a valid instance size and is being used for demonstration purposes only.

28. Run terraform show with json options

```
terraform show -json out.tfplan > output.json  
nano output.json
```

29. Examine the `output.json` file for the `ec2_instance_value`. Here it shows that the deployment would use the size entered when prompted...

A screenshot of a terminal window showing the contents of the output.json file. The file is open in nano. The text visible is: "value": "us-west-1", "ec2_instance_size": {"value": "Prompted Size"}, "inst. The "ec2_instance_size" key and its value are circled in orange.

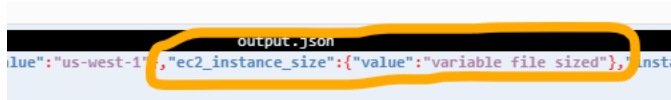
30. A variable can be assigned with a value as a local environment variable. This option will not be applied in this task and therefore the default value assigned to a variable in the variable file is next in the order of precedence and will be used if there are no higher precedent overrides....

31. In `variable.tf`, uncomment the default line for `ec2_instance_size` and enter a value of **"variable file sized"** Save the file, `ctrl+x` out of nano and then and run the following commands (you can copy and paste in all 3 lines at the same time) ...

```
terraform plan -out=out.tfplan
```

```
terraform show -json out.tfplan > output.json
nano output.json
```

32. Examine the output.json file for the ec2_instance_size value. Here it shows that the deployment would use the size entered in the variable file...

A screenshot of a terminal window showing the contents of the output.json file. The file is titled "output.json". The JSON content is partially visible, showing "value": "us-west-1", "ec2_instance_size": {"value": "variable file sized"}, and "instan". A yellow circle highlights the "ec2_instance_size" entry, indicating that the deployment would use the size entered in the variable file.

33. The value assigned to variables in .tfvar files is next in order of precedence. tfvar values will override default values declared in variable files
34. Ctrl+x out of nano and then create a file called terraform.tfvars...

```
touch terraform.tfvars
```

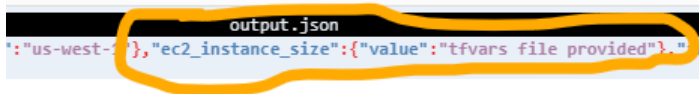
35. Open terraform.tfvars, paste the following code and save

```
ec2_instance_size = "tfvars file provided"
```

36. Run...

```
terraform plan -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

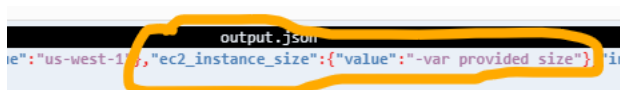
37. Examine the output.json file for the ec2_instance_size value. Here it shows that the deployment would use the size entered in the tfvars file....

A screenshot of a terminal window showing the contents of the output.json file. The file is titled "output.json". The JSON content is partially visible, showing "value": "us-west-1", "ec2_instance_size": {"value": "tfvars file provided"}, and "instan". A yellow circle highlights the "ec2_instance_size" entry, indicating that the deployment would use the size entered in the tfvars file.

38. The value assigned to variable at runtime using the -var command line option is the highest order of precedence and override all other values
39. Ctrl+x out of nano and then run terraform plan with -var option specifying the ec2_instance_size value...

```
terraform plan -var="ec2_instance_size=-var provided size" -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

40. Examine the output.json file for the ec2_instance_value. Here it shows that the deployment would use the size entered using the -var option at the command line...

A screenshot of a terminal window showing the contents of the output.json file. The file is titled "output.json". The JSON content is partially visible, showing "value": "us-west-1", "ec2_instance_size": {"value": "-var provided size"}, and "fin". A yellow circle highlights the "ec2_instance_size" entry, indicating that the deployment would use the size entered using the -var option at the command line.

41. Ctrl+x out of nano and then modify the default for region_az in variable.tf from "use-west-1a" to "From Variable file". Save the changes

Task4. Working with multiple tfvar files

During this task we will focus on 3 variables: **ec2_instance_size**, **instance_ami** and **region_az**. We will create two new .tfvars files and populate them with values for these variable as well as using the existing terraform.tfvars file. We will look at the precedence order used to determine the final value for the variables.

Note The values assigned to these variables are purely for demonstration purposes.

1. Create 2 new files

```
touch file_1.tfvars
touch file_2.tfvars
```

2. Copy the following into `file_1.tfvars`...

```
ec2_instance_size = "from file_1.tfvars"
instance_ami      = "from file_1.tfvars"
region_az         = "from file_1.tfvars"
```

3. Copy the following into `file_2.tfvars` ...

```
ec2_instance_size = "from file_2.tfvars"
instance_ami      = "from file_2.tfvars"
region_az         = "from file_2.tfvars"
```

4. Copy the following into `terraform.tfvars` overwriting the current content ...

```
ec2_instance_size = "from terraform.tfvars"
instance_ami      = "from terraform.tfvars"
region_az         = "from terraform.tfvars"
```

Test 1: A single tfvars file declaring all variable values

1. Save all three files and run ...

```
terraform plan -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

2. Examine the output.json file to view where the values for the three variables are derived.

Result: If it exists, terraform.tfvars is automatically loaded during plan, apply and destroy operations. Other tfvars files must be explicitly referenced. (auto tfvars will be introduced later) As terraform.tfvars is the only currently active .tfvars file and we have not overridden any values with the -var option, all three variable values are therefore derived from terraform.tfvars

```
output.json
{"ec2_instance_size":{"value":"from terraform.tfvars"},"instance_ami":{"value":"from terraform.tfvars"},"region_az":{"value":"from terraform.tfvars"}}
```


Test 2. Using a single tfvars file to declare some but not all variable values

1. In **terraform.tfvar** comment out line 3, `region_az`, and save
2. Ctrl+x out of nano and the run...

```
terraform plan -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

3. Examine the output.json file to view where the values for the three variables are derived.

Result: Given that the `region_az` variable is not defined in `terraform.tfvars` nor entered as a `-var` option, its value is derived from lower precedence sources, in this case the default value in the variable file. The other values are derived from `terraform.tfvars`

```
output.json
{"ec2_instance_size":{"value":"from terraform.tfvars"},"instance_ami":{"value":"from terraform.tfvars"},"region_az":{"value":"From variable file"}}
```

Test 3. Using a tfvars file other than terraform.tfvars to declare all values

1. In **terraform.tfvar** un-comment line 3 `region_az` and save.
2. Ctrl+x out of nano and then specify the inclusion of `file_1.tfvars` by running...

```
terraform plan --var-file=file_1.tfvars -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

3. Examine the output.json file to view where the values for the three variables are derived.

Result: Additional tfvar files can be specified using `--var-file={filename}`. If values in these files conflict with values in `terraform.tfvars`, or if there is no `terraform.tfvars` file, then these have the higher precedence. All three values are therefore derived from `file_1.tfvars`

```
output.json
{"ec2_instance_size":{"value":"from file_1.tfvars"},"instance_ami":{"value":"from file_1.tfvars"},"region_az":{"value":"from file_1.tfvars"}}
```

Test 4. Using a tfvars file other than terraform.tfvars to declare some but not all values

1. In **file_1.tfvars** comment out `region_az` and save.

Question. Where do you think the `region_az` value will be derived from if we include this file?

2. Ctrl+x out of nano and then specify the inclusion of `file_1.tfvars` by running...

```
terraform plan --var-file=file_1.tfvars -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

3. Examine the output.json file to view where the values for the three variables are derived.

Result: If additional tfvar file values do not override the values specified in terraform.tfvars, then the terraform.tfvars values will apply. Therefore region_az is from terraform.tfvars file whilst ec2_instance_size and instance_ami are derived from file_1.tfvars

```
output.json
{"ec2_instance_size":{"value":"from file_1.tfvars"},"instance_ami":{"value":"from file_1.tfvars"},"region_az":{"value":"from terraform.tfvars"}}
```

Test 5. Using multiple optional tfvars files and terraform.tfvars (1)

Question: We are going to specify the inclusion of both file_1.tfvars and file_2.tfvars Where do you think the 3 variable values will be derived from ?

1. Ctrl+x out of nano and then run...

```
terraform plan --var-file=file_1.tfvars --var-file=file_2.tfvars -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

2. Examine the output.json file to view where the values for the three variables are derived.

Result: When multiple additional tfvar files are loaded in sequence, values in the latter have precedence over those in the former where there are conflicts. Therefore, all values are from file_2.tfvars

```
output.json
{"ec2_instance_size":{"value":"from file_2.tfvars"},"instance_ami":{"value":"from file_2.tfvars"},"region_az":{"value":"from file_2.tfvars"}}
```

Test 6. Using multiple optional tfvars files and terraform.tfvars (2)

1. Comment out region_az and instance_ami in `file_2.tfvars` and save.
2. Verify region_az is commented out in `file_1.tfvars` and save.

Question: Where will the 3 variable values now be derived from ?

3. Ctrl+x out of nano and then run...

```
terraform plan --var-file=file_1.tfvars --var-file=file_2.tfvars -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

3. Examine the output.json file to view where the values for the three variables are derived.

Result: When multiple additional tfvar files are loaded in sequence, values in the latter have precedence over those in the former where there are conflicts. Therefore instance_size is derived from file_2.tfvars, instance_ami from file_1.tfvars and region_az from terraform.tfvars

```
output.json
{"ec2_instance_size":{"value":"from file_2.tfvars"},"instance_ami":{"value":"from file_1.tfvars"},"region_az":{"value":"from terraform.tfvars"}}
```

Test 7. Using multiple tfvars files combined with -var options

1. Finally, we will evaluate the effect of using explicit -var options combined with tfvars files.
2. Evaluate the command below. Where will the variable values be derived from? Ctrl+x out of nano and then run the following to check your assumption...

```
terraform plan --var-file=file_1.tfvars --var-file=file_2.tfvars -var="region_az=from var-input" -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

3. Examine the output.json file to view where the values for the three variables are derived.

Result: Variable values specified explicitly have the highest precedence order. Therefore region_az is from var-input, instance_size is from file_2.tfvars file 2 and instance_ami is from file_1.tfvars file 1

```
output.json
{"ec2_instance_size":{"value":"from file_2.tfvars"},"instance_ami":{"value":"from file_1.tfvars"},"region_az":{"value":"from var-input"}}
```

4. Ctrl+x out of nano, close the file_1.tfvars and file_2.tfvars tabs and then delete file_1.tfvars and file_2.tfvars...

```
rm file_1.tfvars
rm file_2.tfvars
```

Task 5. Working with auto-tfvars files

1. Along with terraform.tfvars we can automatically load other tfvars files by naming them {name}.auto.tfvars
2. Create 2 auto.tfvars files

```
touch file_a.auto.tfvars
touch file_b.auto.tfvars
```

3. Copy the following code into file_a.auto.tfvars

```
ec2_instance_size = "from file_a.auto.tfvars"
instance_ami = "from file_a.auto.tfvars"
region_az = "from file_a.auto.tfvars"
```

4. Copy the following code into file_b.auto.tfvars

```
ec2_instance_size = "from file_b.auto.tfvars"
instance_ami = "from file_b.auto.tfvars"
region_az = "from file_b.auto.tfvars"
```

5. Ensure all lines of terraform.tfvars are uncommented.
6. Save all files and run...

```
terraform plan -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

7. Examine the output.json file to view where the values for the three variables are derived.

Result: Terraform.tfvars and any auto.tfvars files are automatically loaded during plan and apply actions. Terraform.tfvars is loaded first and then subsequent auto.tfvars are loaded in lexical name order, a before b before c etc. If there are conflicts of variable values, then the later loaded file takes precedence over the earlier loaded file. Therefore, in this example all values are derived from file_b.auto.tfvars...

```
output.json
{"ec2_instance_size":{"value":"from file_b.auto.tfvars"},"instance_ami":{"value":"from file_b.auto.tfvars"},"region_az":{"value":"from file_b.auto.tfvars"}},
```

8. Comment out region_az in file_b.auto.tfvars and save
9. Ctrl+x out of nano and then run..

```
terraform plan -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

10. Examine the output.json file to view where the values for the three variables are derived.

Result: Only if there are conflicts of variable values do the later loaded files takes precedence over earlier loaded files. region_az is not declared in file_b.auto.tfvars and is therefore derived from file_a.auto.tfvars whilst ec2_instance_size and instance_ami are still derived from file_b.auto.tfvars

```
output.json
{"ec2_instance_size":{"value":"from file_b.auto.tfvars"},"instance_ami":{"value":"from file_b.auto.tfvars"},"region_az":{"value":"from file_a.auto.tfvars"}},
```

11. Comment out both region_az and instance-ami in file_b.auto.tfvars and save
12. Comment out region_az in file_a.auto.tfvars and save

Question. Where do you think the 3 variable values will be derived in this example ?

13. Ctrl+x out of nano and then run..

```
terraform plan -out=out.tfplan
terraform show -json out.tfplan > output.json
nano output.json
```

Result: Only if there are conflicts of variable values do the later loaded file takes precedence over the earlier loaded file. Therefore, instance_size is derived from file_b.auto.tfvars, instance_ami is derived from file_a.auto.tfvars and region_az is derived from terraform.tfvars

```
output.json
"ec2_instance_size":{"value":"from file_b.auto.tfvars"},"instance_ami":{"value":"from file_a.auto.tfvars"},"region_az":{"value":"from terraform.tfvars"}
```

14. Ctrl+x out of nano and then delete all of your .tfvars files prior to moving onto the next task.

```
rm *.tfvars
```

15. In **variable.tf** reset the default value of the **region_az** variable back to **"us-west-1a"** and the default value of the **"ec2_instance_size"** back to **"t3.micro"**. Save the changes.

Task 6. Simple Iterations

Our code currently deploys a single EC2 instance onto each of our 2 subnets. What if we want more? We could duplicate our `aws_instance` blocks, renaming each to give us as many instances as we need. This is poor coding, a better solution being to iterate through the same block multiple times.

1. Paste the following block the end of **variable.tf** and save the changes...

```
variable "instance_count" {
    description = "Number of EC2 instances in each subnet"
    type        = number
    default     = 2
}
```

2. Open **main.tf** and paste the following argument line into the 2 `aws_instance` blocks, above the `ami` argument.

```
count = var.instance_count
```

3. Save both files and run ``terraform apply` `yes``

4. Review the deployment into **us-west-1 (N.California)** using the console

5. Notice that whilst we have created the correct number of instances, 2 for each subnet, there is name duplication.

6. Run **terraform destroy yes**

7. One option to resolve this problem would be to simply remove the `name` tag as this is not a mandatory parameter. Not naming instances can cause confusion though and often a naming convention is adopted. We need to use the following naming convention ``name-az-#``

Examples:

``pub-us-west-1a-001`` an instance on the public subnet

``priv-us-west-1a-002`` an instance on the private subnet

Challenge

1. Modify and test your code so that the desired naming convention is adopted. This can be achieved with the modification of a single line of code in each of the `aws_instance` resource blocks.

Guidance

<https://developer.hashicorp.com/terraform/language/meta-arguments/count>

<https://developer.hashicorp.com/terraform/language/functions/format>

The solution can be found in `iterations.tf` in the solutions folder for this lab

Clean-Up

1. Destroy all of your resources ahead of the next lab using **`terraform destroy yes`**

***** Congratulations, you have completed this lab *****