

# Terraform In Azure

Automate your cloud deployments  
with Infrastructure-As-Code



Michael Levan

# Terraform in Azure

Michael Levan

This book is for sale at [http://leanpub.com/terraform\\_in\\_azure](http://leanpub.com/terraform_in_azure)

This version was published on 2020-02-26



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 Michael Levan

# Contents

<b>About This Book</b> . . . . .	<b>1</b>
<b>Brief Contents</b> . . . . .	<b>2</b>
<b>Acknowledgements</b> . . . . .	<b>3</b>
<b>Chapter 1</b> . . . . .	<b>4</b>
<b>Getting started with Terraform concepts</b> . . . . .	<b>5</b>
What does Terraform do for us? . . . . .	6
What is a Provider? . . . . .	7
What is a Resource? . . . . .	8
What are Resource Parameters (attribute reference)? . . . . .	8
What are Modules? . . . . .	8
Terraform configuration files. . . . .	8
Summary . . . . .	9
<b>Chapter 2</b> . . . . .	<b>10</b>
<b>The Azure Terraform provider</b> . . . . .	<b>11</b>
Setting up the Azure Terraform provider . . . . .	11
Authentication with the AZ CLI . . . . .	12
Authentication with a service principal and client secret . . . . .	14
Summary . . . . .	16
<b>Chapter 3</b> . . . . .	<b>17</b>
<b>Writing out first piece of Terraform code</b> . . . . .	<b>18</b>
Setting up our environment . . . . .	18
Getting our code ready . . . . .	19
Writing our code . . . . .	21
Run our code. . . . .	24
Summary . . . . .	28
<b>Chapter 4</b> . . . . .	<b>30</b>

## CONTENTS

<b>Creating a network</b>	<b>31</b>
Setting up our storage account for TFSTATE	31
Creating a network security group	33
Creating a network security group rule	38
Creating a vNet	40
Putting it all together	41
Summary	43

# About This Book

## Summary

As time goes on in the tech world, things change. I remember when people used to say “every 2-5 years things are changing”. Now it feels like every year, or less. One of the many huge things we’ve seen in the tech world lately is DevOps. DevOps means a different thing to everyone. For me, DevOps is about continuously delivering value to our end users and our business. To do this properly, we need to be well-rounded professionals in all aspects of technology. Infrastructure, applications, support, troubleshooting, coding, and the list goes on and on. One of the components that makes us deliver value to our business is efficiency. Efficiency comes in many shapes and forms. What we’ll be focusing on in this book is the efficiency of how fast we spin up our infrastructure, how reliable the deployment is, and is it repeatable. What I mean by “repeatable” is simple. Anyone can spin up infrastructure with some code, but can you do it two times? Five times? Ten times? with the same code? Do you have static values that make it less agnostic and specific for one environment? Do you have the ability to spin it up anywhere and everywhere?

## Who should read this book

Whether you’re a sysadmin, infrastructure engineer, developer, or DevOps professional, this book will open your eyes to automating your deployments quickly and reliably. This book does focus on Terraform, but the concepts learned can be used with any infrastructure-as-code platform.

## Author

Michael Levan is an Engineer at heart. Michael started his career like most do, in helpdesk/desktop support. After realizing the different ways to automate his job, he quickly realized the next step to his career, DevOps and Cloud Engineering. After working as a Sysadmin managing Hyper-V, ESXi, and all-around Windows Infrastructure, he wanted to take the next step in his career. This led him to doing a complete 180 and moving into Linux and AWS. After not knowing much about this space, Michael buckled down, studied, and quickly became a Subject Matter Expert. After working with Linux, AWS, Python, Ansible, Containerization, Orchestration, and Web Applications, Michael started to become fascinated with the improvements that Microsoft was implementing in the same space that Linux and Open Source tooling have been in for so long. He took this opportunity to do yet another 180, go back to the Microsoft space, and focus his efforts in practicing DevOps on the other side of the realm. This was what inspired him to write this book. Michael’s primary day-to-day as a DevOps Engineer revolves around Configuration Management, Azure, Azure DevOps, PowerShell, .NET, Docker, Orchestration, Cloud Engineering, Source Control, and helping the organization ship the product faster. He is a public speaker, blogger, all-around technology enthusiast, E-Trainer, and podcaster.

# Brief Contents

Chapter 1 - Getting started with Terraform concepts

Chapter 2 - The Azure Terraform provider and authentication

Chapter 3 - Writing our first piece of Terraform code

Chapter 4 - Creating a network with Terraform

# Acknowledgements

I would like to thank my son Zachary for giving me the strength to keep striving for success. I'm an engineer because I love it of course, but every day I push to be more and more successful for him. Zac gave me the greatest gift in the world which is being a dad. I want to re-pay him by giving him the best life possible. I love you big guy.

I would also like to thank my girlfriend Jessica. Jessica, you're the most understanding person I know. With me starting a business to always being on my laptop to being an entrepreneur, all of those things take a massive amount of time. You sit there for hours with me while I'm on my laptop and encourage me to keep going. As you always say "just don't quit". I love you so much, thank you.

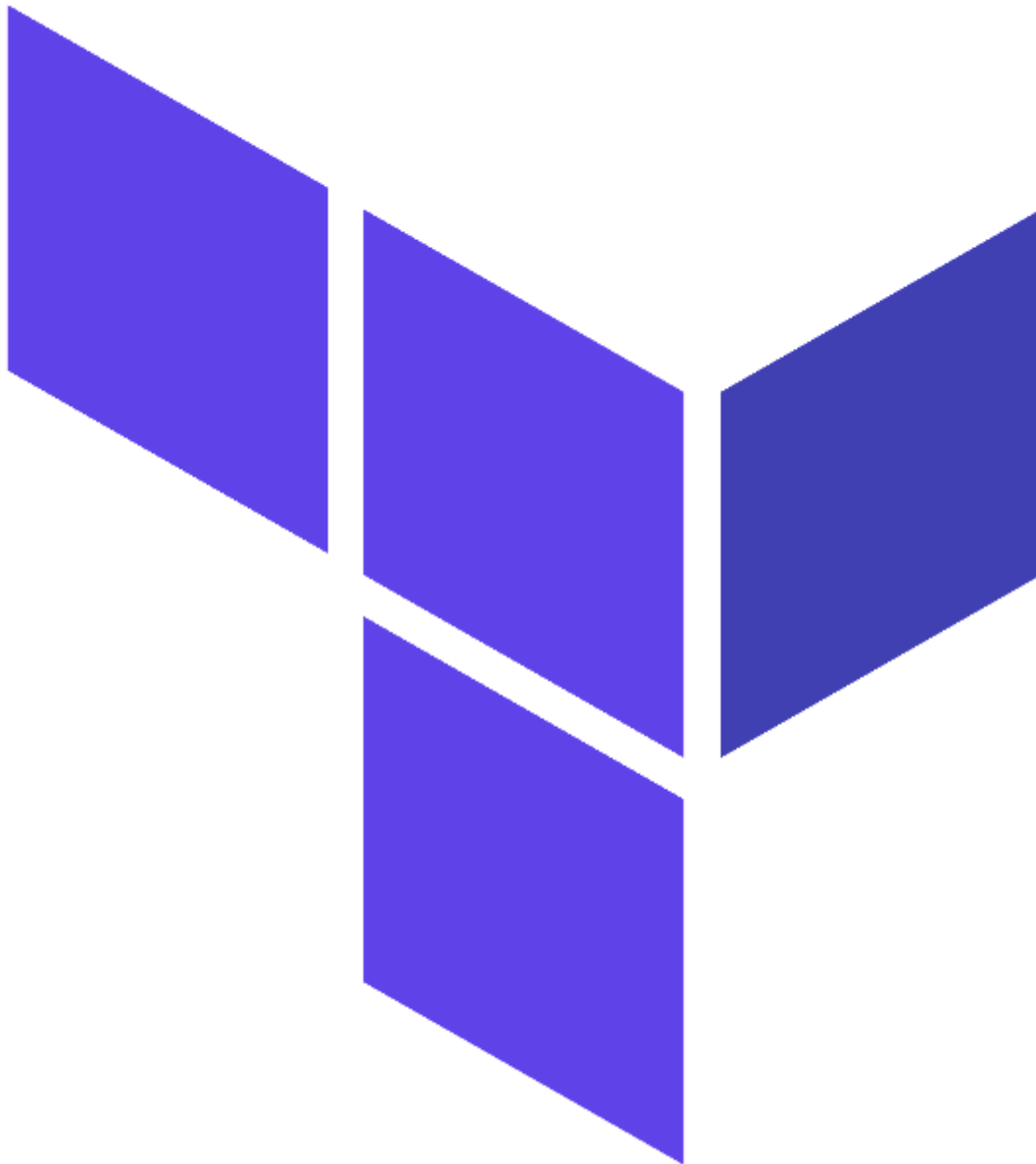
# Chapter 1



# Getting started with Terraform concepts

This chapter will be the only chapter that's all theory and explanation. Following this chapter will be all labs with explanations.

## What does Terraform do for us?



Terraform Logo

Before embarking on our coding journey to make spinning up cloud resource easier, let's think about the actual key concept here. What does infrastructure-as-code do for us? In our case to be specific, what does Terraform do for us? Terraform and infrastructure-as-code in general have one primary

purpose: Automate the creation of our infrastructure components and create reusable code to do so. There is nothing worse than creating an environment and then being asked to do it again five times manually. It's not only a manual effort, but it's a waste of company resources. Let's think about an example.

Throwing out fake numbers. Let's say you're a DevOps Engineer that makes \$60 dollars an hour. It takes you two hours to manually spin up a piece of infrastructure for each client. You onboard 3 new clients every week. That's \$360.00 dollars each week, \$1,440 dollars each month. In a six month period this is costing the organization \$8,640 dollars. If you took those two hours to write reusable automation for this process, it would only cost the organization \$120.00 dollars.

Let's think of another example from an engineering point of view. Let's take the same example as above. That's 6 hours per week that's taken out of your day to do manual labor for no reason at all. That time could be spent working on other projects, helping your peers, studying, making deployments faster and more efficient, or simply all of the above.

So, what does Terraform do for us? Organizations face many issues with change, but let's talk about the fundamental challenges for a change like implementing infrastructure-as-code. Technical challenges and organization challenges. Let's start with technical challenges.

Technical challenges is the manual process of building your environment, configuration drift, using different tools which require maximum upkeep, and inconsistencies throughout the environment. Terraform makes this extremely simple. It's a single engine that allows you to interact with multiple tools. Some of these tools include Azure, AWS, VMWare, PagerDuty, RabbitMQ, and even GitHub. The list goes on and on so to make it easier for you, take a look at this link: [<https://www.terraform.io/docs/providers/index.html>]

Business challenges is simply "how does it scale?". As the infrastructure scales, the complexity of that infrastructure scales. When you first start out, maybe you'll have a few VMs, a network, and a web app. Before you know it, that will scale to a sixteen node web cluster with a blue/green deployment for every customer. How do you maintain that? Surely you cannot manually. Not only will this cost the organization money, but it will cost the organization employees from burn-out.

## What is a Provider?

As I hinted at above, a Terraform Provider is a way to interact with different organizations (Microsoft, Amazon, etc.) via an API. What happens is a company (like Microsoft) will give Terraform an API to connect to. There will be certain permissions based on what the API can interact with on Microsoft's end. If you're looking for a piece of functionality that doesn't exist, it's most likely because that piece of the API isn't open to the public. Think of a provider "kind of" like a class if you're a developer.

## What is a Resource?

A resource is a way to interact with the provider (providers are like a class). Within the provider, there's a resource. For example, in the AWS provider there is a resource called 'aws\_vpc' which you can use to interact with an AWS VPC. Being a DevOps Engineer myself, a resource to me is very, very much like a method or a function in a programming language.

## What are Resource Parameters (attribute reference)?

Resource Parameters is what you would use to pass into your resource (resources are like methods or functions). If you've ever written code before or have a development background, you know that a method or a function has parameters that you can pass in. It's very similar in this case as well. Let's take the 'aws\_vpc' example. It has parameters (or references) for things like passing in the ID of a VPC, the CIDR block, the owner ID, and plenty more.

## What are Modules?

Modules in short are just files. Any configuration file inside of a folder is "module". You can call these modules in different modules for say dependencies, like an output that you want to capture.

## Terraform configuration files.

Terraform configuration files come in a few basic flavors;

- `main.tf` == This is where your main configuration will be for doing things like spinning up a vNet or creating a Virtual Machine.
- `variables.tf` == This is where your variable names will go along with the type (string, number, etc.)
- `terraform.tfvars` == Your `.tfvars` file will be variables that you would like to be passed in at runtime. You can do this at your terminal or simply create a file to do this.
- `output.tf` == An output file for any piece of information that you would like to make public for another module. For example, let's say you're building your network in one module and your virtual machine in another module. You can have an output of your network ID so it can be called in your virtual machine module to specify what network you're tying your virtual machine to.

## Summary

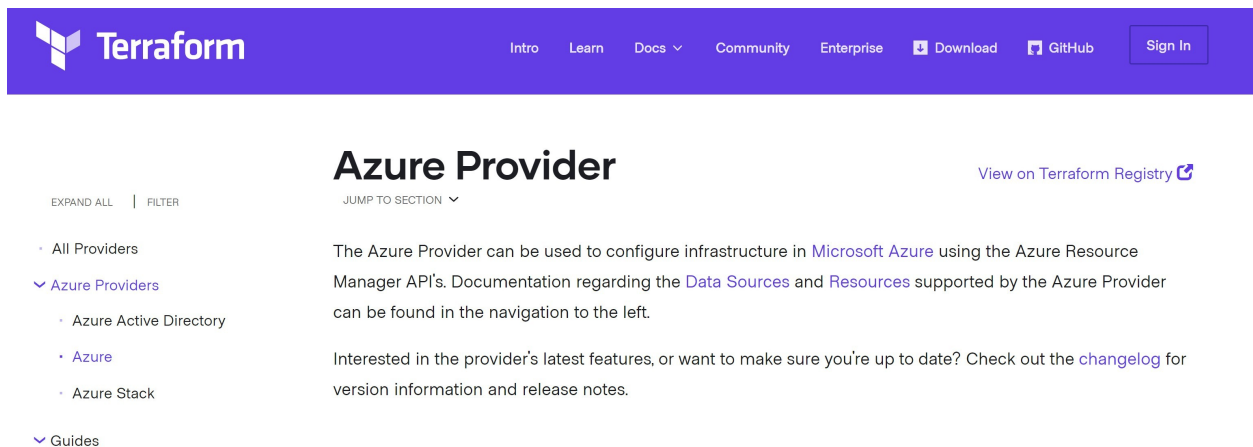
If some of this does not make perfect sense, don't worry. We will be using all of the concepts learned above in our labs in chapters 2-10. This is very much a "recipes" book, so this is the most verbose chapter when it comes to reading. After this will be a ton of labs with explanations for the labs.

# Chapter 2

# The Azure Terraform provider

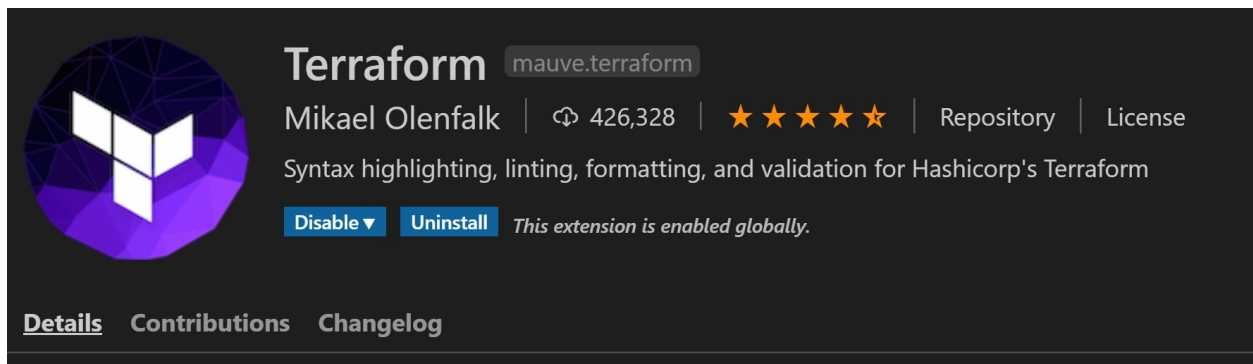
## Setting up the Azure Terraform provider

The first thing we're going to do is take a look at the Azure Terraform provider.

A screenshot of the Terraform website's Azure Provider page. The header is purple with the Terraform logo and navigation links: Intro, Learn, Docs, Community, Enterprise, Download, GitHub, and Sign In. The main content area has a purple sidebar on the left with a list of providers: All Providers, Azure Providers (selected), Azure Active Directory, Azure, Azure Stack, and Guides. The main content area is white and titled 'Azure Provider' with a link to 'View on Terraform Registry'. The text describes the provider's purpose and provides a link to the changelog.

### Azure Provider

The Azure provider gives us a programmatic way to interact with Azure using Terraform. I will be using VSCode and the Terraform extension within VSCode to write my code.

A screenshot of the Terraform extension page in VS Code. The page has a dark background. On the left is the Terraform logo. To the right of the logo, the text 'Terraform' is displayed in a large font, followed by 'mauve.terraform' in a smaller font. Below this, the author 'Mikael Olenfalk' is listed, along with a download count of 426,328 and a star rating of five stars. The text 'Repository' and 'License' are also visible. A description states: 'Syntax highlighting, linting, formatting, and validation for Hashicorp's Terraform'. At the bottom, there are buttons for 'Disable' and 'Uninstall', and a note that 'This extension is enabled globally.' Below the main content, there are links for 'Details', 'Contributions', and 'Changelog'.

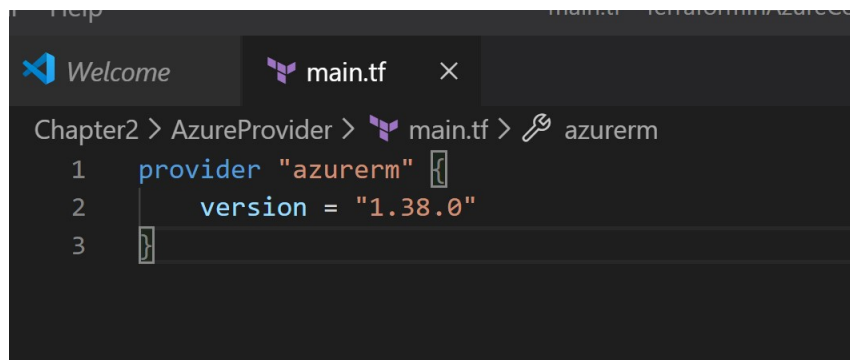
### Terraform Extension

Once I have my Terraform extension, I can start writing my code to interact with the Azure provider. The first thing we need to do in our main.tf is specify the provider we're using. In our case we will specify Azure and use version 1.38

```
1 provider "azurerm" {  
2  
3 }
```

As you can see above, we're specifying the `azurerm` provider. After we specify the provider, we can specify the version.

```
1 provider "azurerm" {  
2     version = "1.38.0"  
3 }
```



AzureRM

We now have our Azure provider. In the next two labs we'll see how to set up authentication using two methods; 1) AZ CLI 2) Azure Service Principal

## Authentication with the AZ CLI

There are four different methods for authenticating to Azure with the Azure provider:

- AZ CLI
- Managed Service Identity
- Service Principal with client cert
- Service Principal with client secret

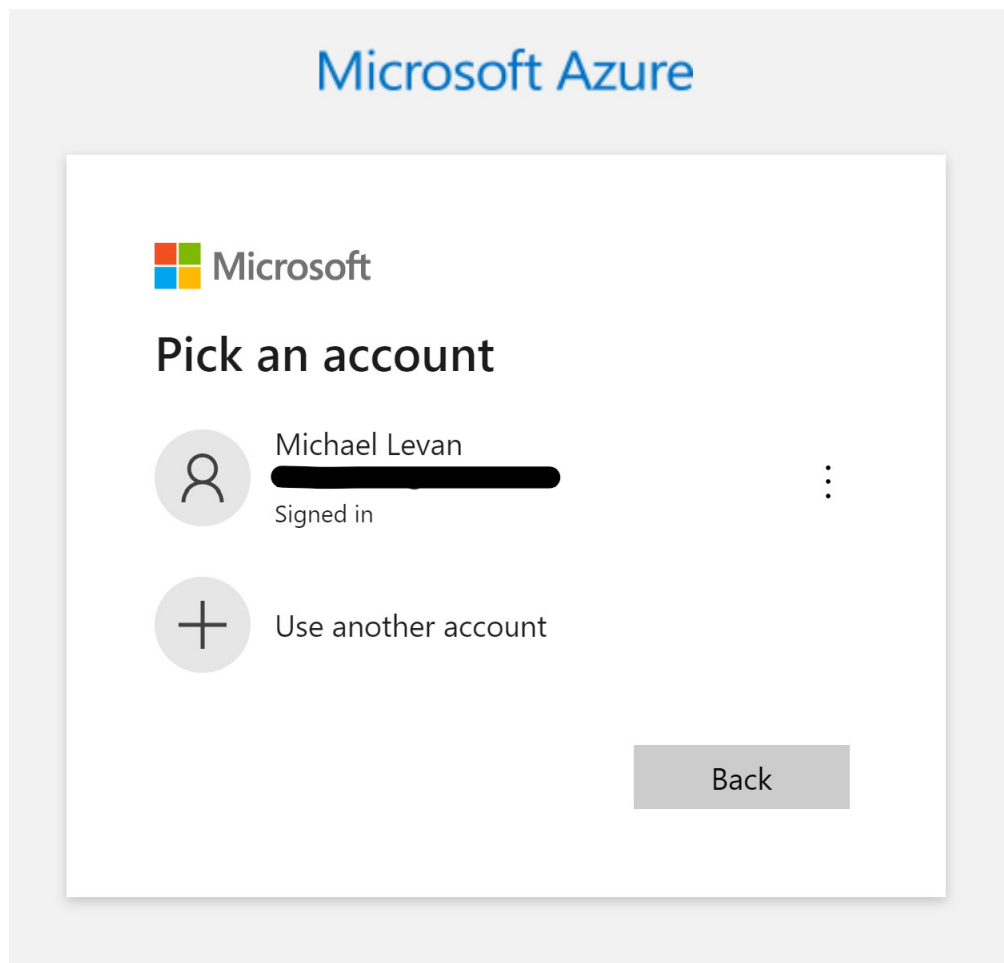
We're going to focus on one and two, or AZ CLI and service principal with client secret.

Let's start with the AZ CLI. Let's talk about what the AZ CLI actually is. AZ CLI is a command-line interface that allows you to interact with Azure resources. You can interact with anything on Azure. To do this however, you need to authenticate. First you'll need to download the AZ CLI. Follow the link below and depending on your OS, follow the proper path.

[<https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>]



Once you have the AZ CLI installed, go ahead and authenticate. Run `az login` on your terminal. A web browser will open and you'll be taken to a Microsoft login page.



Microsoft Login

Once you authenticate you will get a message in a web browser that you've logged in successfully.

### **You have logged into Microsoft Azure!**

You can close this window, or we will redirect you to the [Azure CLI documents](#) in 10 seconds.

### **Auth**

When you go back to your terminal, you'll see an output of your subscription information.

```
PS C:\Users\Mike\Desktop\TerraformInAzureCode> az login
Note, we have launched a browser for you to login. For old experience with device code, use "az login --use-device-code"
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "id": "11111111-1111-1111-1111-111111111111",
    "isDefault": true,
    "name": "Mike-Pay-As-You-Go",
    "state": "Enabled",
    "tenantId": "11111111-1111-1111-1111-111111111111",
    "user": {
      "name": "Mike",
      "type": "user"
    }
  }
]
PS C:\Users\Mike\Desktop\TerraformInAzureCode>
```

### CLI Auth

Once you are authenticated, set your subscription to the subscription you wish to interact with via Terraform.

```
az account set --subscription="SUBSCRIPTION_ID"
```

You will now be able to run any resource from the `azurerm` provider using your subscription.

```
1 provider "azurerm" {
2     version = "1.38.0"
3     subscription_id = "your_subscription_id"
4 }
```

The great part about this is it's extremely easy and straight-forward. If you're already using Azure, you're most likely using the AZ CLI already. When you start initiating your environment (`terraform init`), planning your environment (`terraform plan`), and creating your environment (`terraform apply`), this authentication method comes in handy. You don't need to worry about setting up service accounts, service principals, certs, and interacting with Azure Active Directory. It's simply an authentication method that's already built into what you're doing daily in the world of Azure. This is a great use case, but how about when you want to say, deploy Terraform code in a CI/CD pipeline? You'll need a way for your pipeline tool, whether it be Jenkins, Azure DevOps, GitLab CI, to authenticate to Azure. Let's find out how to do that next.

## Authentication with a service principal and client secret

Authenticating with a Service Principal requires access to Azure Active Directory. Using this method will give you the ability to create a service account with a tenant ID, client ID, and client secret, which is what is needed to authenticate using this method with Azure.

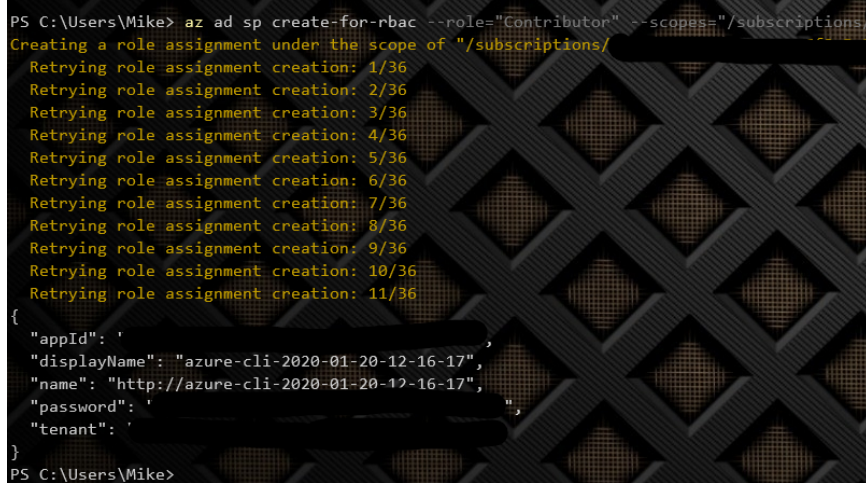
Because we have our account set using `az login` from the last lab, we can continue to use the same account. If you wish to use a different account, please use `az login` to authenticate to the account you wish to use.

We'll need to run an AZ CLI command to create our service principal and RBAC (Role Based Access Control) rules.

```
az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/your_subscription_id"
```

You should see an output like the one below. Ensure you save it as you'll need the tenant ID, client ID, and client secret.

- `appId == client_id`
- `password == client_secret`
- `tenant == tenant_id`



```
PS C:\Users\Mike> az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/your_subscription_id"
Creating a role assignment under the scope of "/subscriptions/your_subscription_id"
Retrying role assignment creation: 1/36
Retrying role assignment creation: 2/36
Retrying role assignment creation: 3/36
Retrying role assignment creation: 4/36
Retrying role assignment creation: 5/36
Retrying role assignment creation: 6/36
Retrying role assignment creation: 7/36
Retrying role assignment creation: 8/36
Retrying role assignment creation: 9/36
Retrying role assignment creation: 10/36
Retrying role assignment creation: 11/36
{
  "appId": " ",
  "displayName": "azure-cli-2020-01-20-12-16-17",
  "name": "http://azure-cli-2020-01-20-12-16-17",
  "password": " ",
  "tenant": " "
}
```

#### RBAC

Now that we have our information, we can modify our provider block from the previous lab to include our authentication information.

```
1 provider "azurerm" {
2     version = "1.38.0"
3
4     subscription_id = "your_subscription_id"
5     client_id = "your_client_id"
6     client_secret = "your_client_secret"
7     tenant_id = "your_tenant_id"
8 }
```

*Please Note*

Adding your client secret into plain text is of course a very poor security practice. This should be stored wherever you or your organization properly stores secrets. To give an example, I would do this in a CICD pipeline by;

- Storing the secret in Azure Key Vault
- Creating a pipeline in Azure DevOps
- Create a task for keyvault to call keyvault values from specific keys
- Have the output of the value as a variable
- Create a Terraform task
- Within the task pass in the variable that holds my secret value. For example, if my variable for my secret value was `$(secret_key)`, I would run `terraform apply -var="client_secret=$(secret_key)"`

You will see more details about the process above for adding secrets in a pipeline in chapter 11.

## Summary

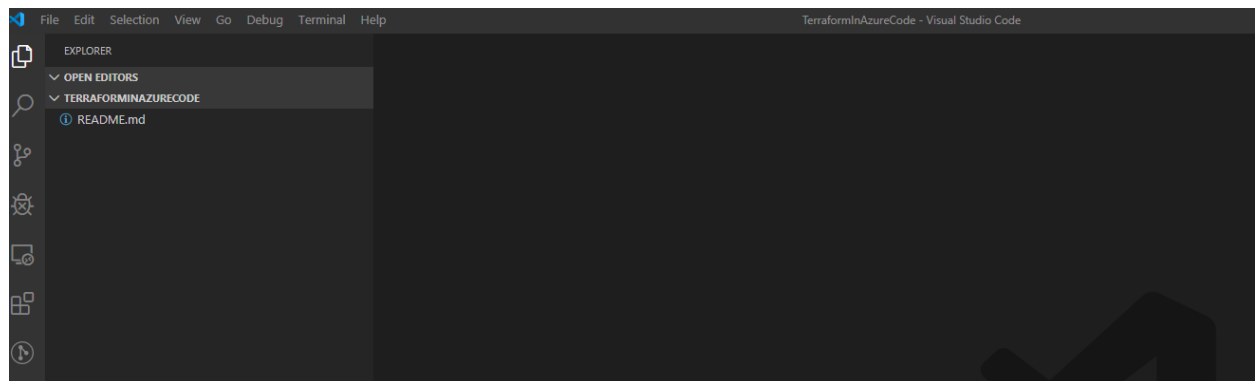
In chapter 1 we spoke about what Terraform is and why we want to use it. In this chapter we went over how to authenticate and interact with Azure using Terraform. This is of course the first step because without authentication, you cannot use the code you write. The authentication methods that Terraform provides us with are very understandable and usable. I especially like the AZ CLI authentication method. Since I use AZ CLI every day, it's incredibly useful to simply use the same authentication method I interact with daily with Terraform.

# Chapter 3

# Writing out first piece of Terraform code

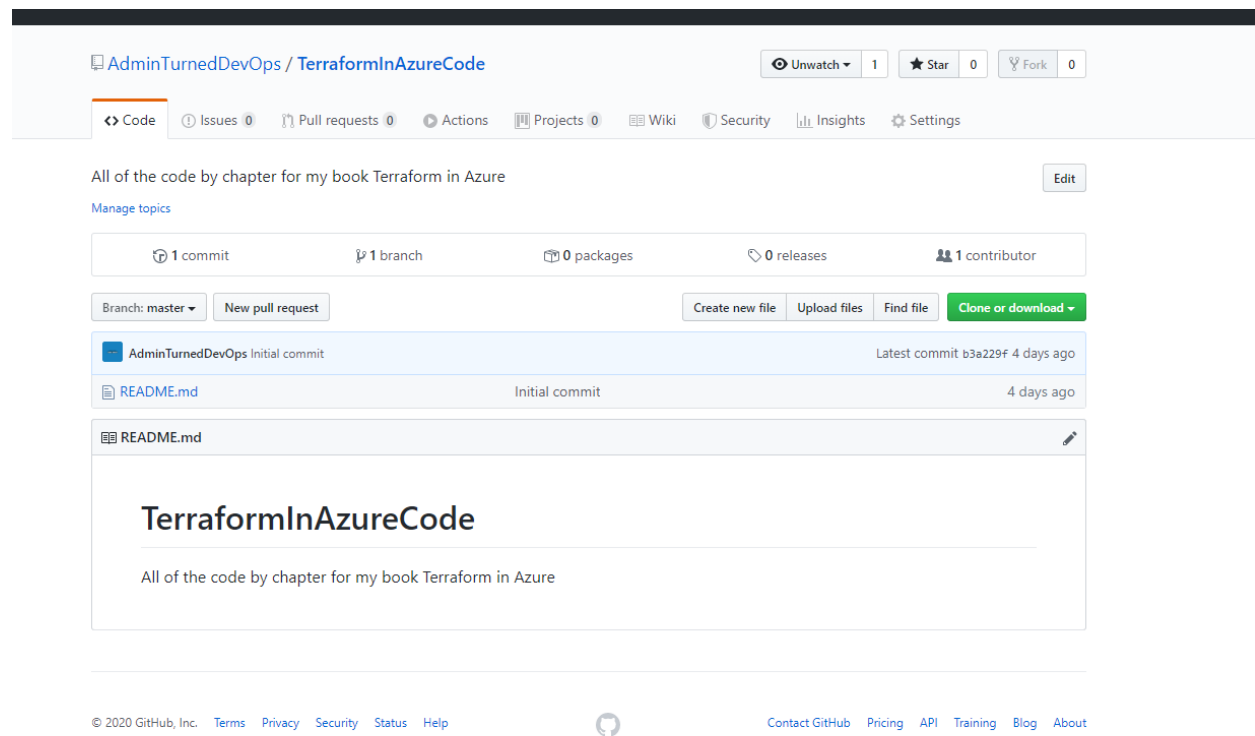
## Setting up our environment

Before we can write any code, we have to set up our environment. First things first, pick an editor or IDE. I personally prefer to use VSCode with the Terraform extension (as seen in chapter 2).



Visual Studio Code

Once you have your editor or IDE chosen, you'll need to store your code in a repo. Source control is very important **ESPECIALLY** for dev code. Why you make ask? Simple. Because Dev code is changed the most. If you change something and it breaks functionality, you want a way to roll back that code change to get back to a working state. Choose where you want your dev source code to be. I personally use GitHub for my source code. You can create a free account in GitHub and store your source code there.

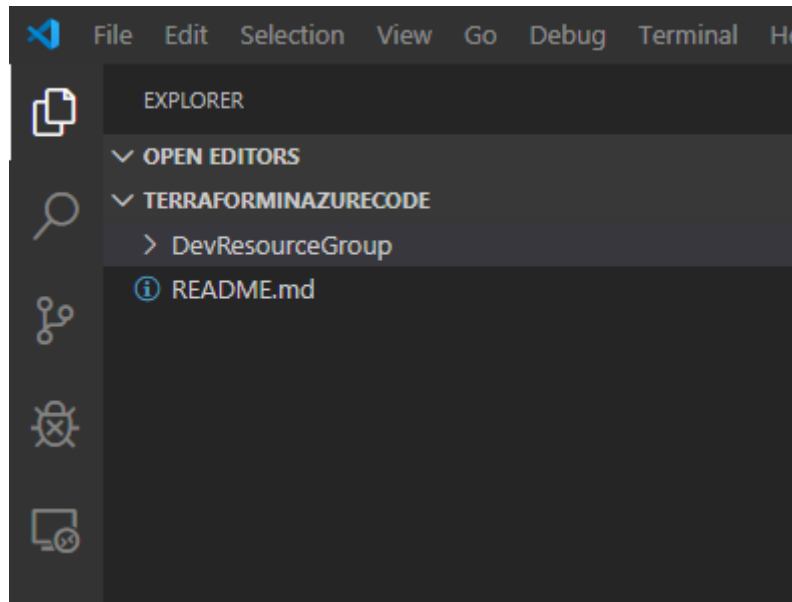


### Github

Once you decide on what distributed version control system to use (GitHub, Azure Repos, GitLab, etc.) clone your repo by running `git clone` to a location of your choosing. Open up that location in whichever editor or IDE you decided to go with. We're now ready to start writing our code.

## Getting our code ready

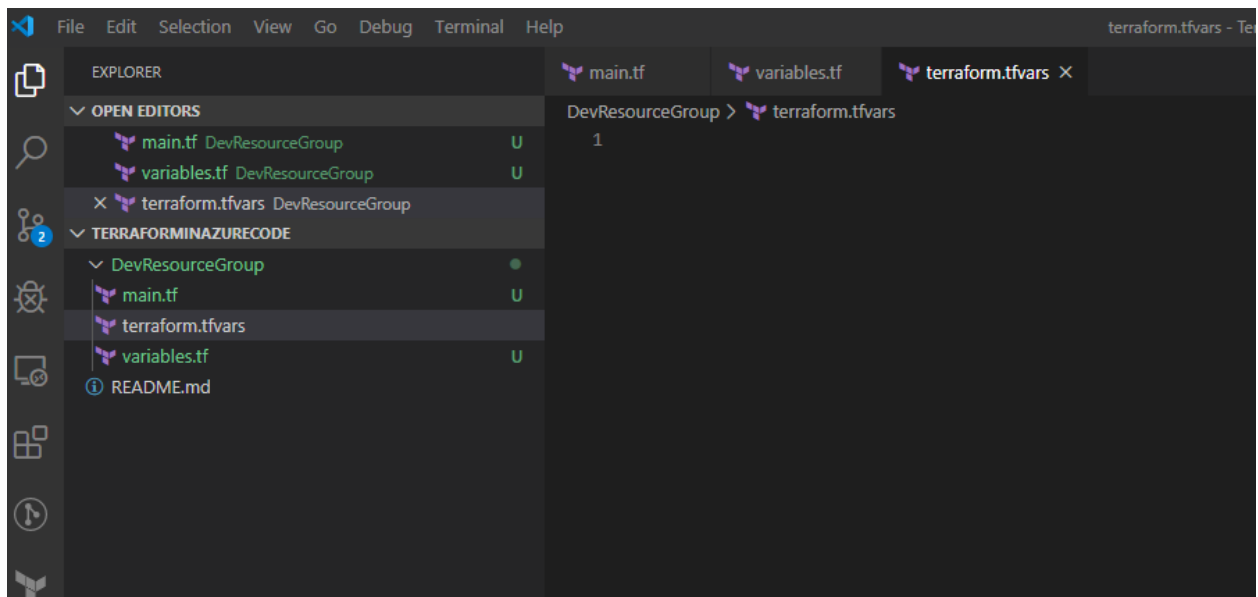
Let's create a directory and call it "DevResourceGroup".



### Dev Resource Group

Within our new directory, let's create 3 files;

- main.tf
- variables.tf
- terraform.tfvars



### Configs

Now that we have our configs, let's add in our Azure provider.



## Writing our code

```
1 provider "azurerm" {  
2     version = "1.38.0"  
3     subscription_id =  
4 }
```

As we can see, we need to specify our subscription ID. We don't want to hard-code this in as we want our Terraform code to be reusable across any environment. Let's go ahead and add a variable for that. Head over to your `variables.tf` config.

```
1 variable "subscriptionID" {  
2     type = string  
3     description = "Variable for our resource group"  
4 }
```

Now that we have our subscription ID variable, we want an easy way to pass in the value at runtime. Let's head over to our `terraform.tfvars` config.

```
1 subscriptionID = "your_subscription_id"
```

Now that we have our first set of variables up, let's now go back to `main.tf` and start creating our resource. We will be using the `azurerm_resource_group` resource to configuration our Azure resource group.

```
1 resource "azurerm_resource_group" "DevRG" {}
```

We'll need to pass in some parameters within our resource. The first two will be `name` and `location`.

```
1 resource "azurerm_resource_group" "DevRG" {  
2     name      = var.resourceGroupName  
3     location  = var.location  
4 }
```

As you can see, I want to pass in variables for my name and location as well. Let's head over to our `variables.tf` config to add in our new variables.

```
1 variable "resourceGroupName" {
2     type = string
3     description = "name of resource group"
4 }
5
6 variable "location" {
7     type = string
8     description = "location of your resource group"
9 }
```

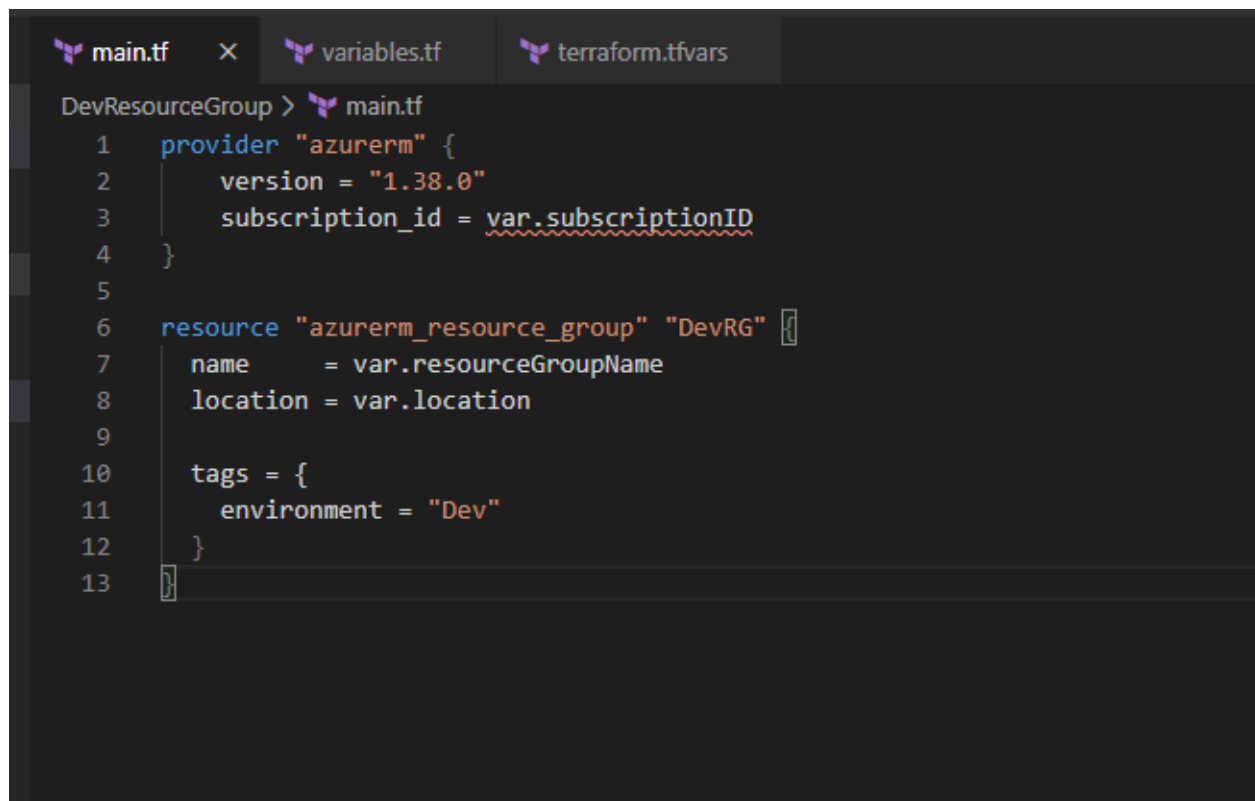
Now that we have our variables set up, let's head over to our `terraform.tfvars` config to add in the values for runtime.

```
1 resourceGroupName = "Dev5"
2 location = "eastus"
```

We have one more thing to add into our resource, the tags.

```
1 resource "azurerm_resource_group" "DevRG" {
2     name      = var.resourceGroupName
3     location  = var.location
4
5     tags = {
6         environment = "Dev"
7     }
8 }
```

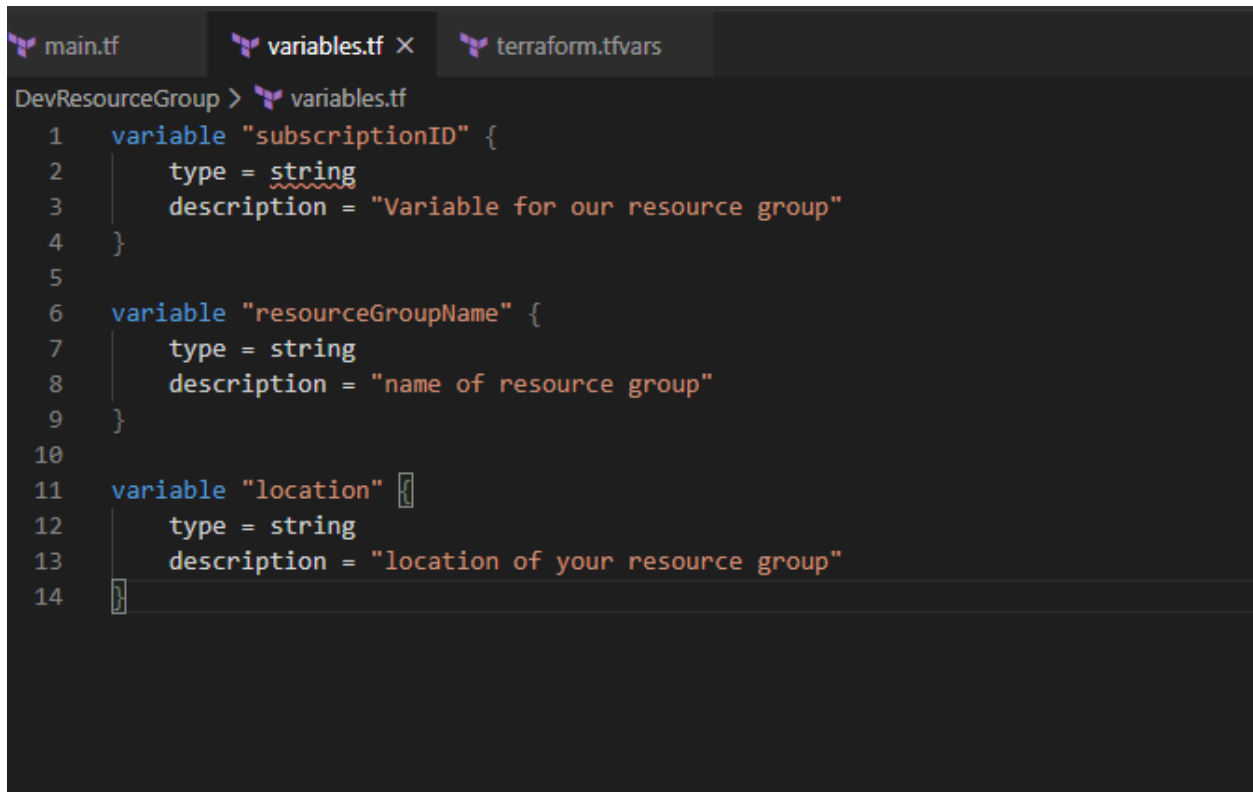
Tags are of course important because we let the user and Azure know what environment this will be in. We can also use tags to deploy certain criteria later on.



The screenshot shows a code editor with three tabs: `main.tf`, `variables.tf`, and `terraform.tfvars`. The `main.tf` tab is active, displaying Terraform code for creating an Azure Resource Group. The code is as follows:

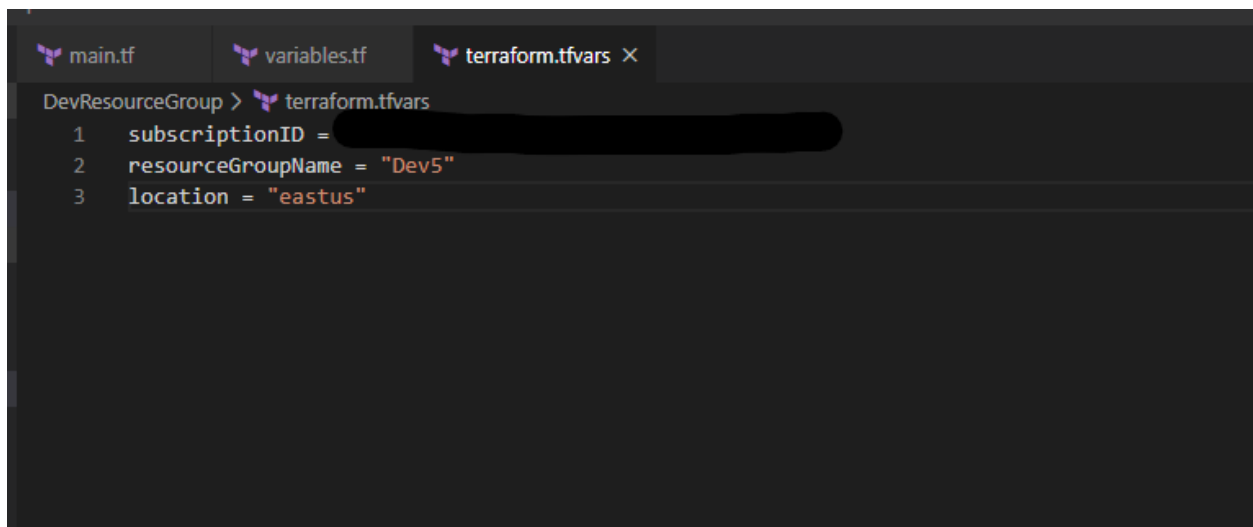
```
DevResourceGroup > main.tf
1 provider "azurerm" {
2     version = "1.38.0"
3     subscription_id = var.subscriptionID
4 }
5
6 resource "azurerm_resource_group" "DevRG" {
7     name      = var.resourceGroupName
8     location  = var.location
9
10    tags = {
11        environment = "Dev"
12    }
13 }
```

Main



```
DevResourceGroup > variables.tf
1  variable "subscriptionID" {
2      type = string
3      description = "Variable for our resource group"
4  }
5
6  variable "resourceGroupName" {
7      type = string
8      description = "name of resource group"
9  }
10
11 variable "location" {}
12     type = string
13     description = "location of your resource group"
14 }
```

Variables



```
DevResourceGroup > terraform.tfvars
1  subscriptionID = 
2  resourceGroupName = "Dev5"
3  location = "eastus"
```

TFVARS

## Run our code.

We're finally ready to run our code!

Ensure that your terminal is in the same directory as your Terraform code.

```
DevResourceGroup > main.tf
1 provider "azurerm" {
2   |   version = "1.38.0"
3   |   subscription_id = var.subscriptionID
4   | }
5
6 resource "azurerm_resource_group" "DevRG" {
7   |   name      = var.resourceGroupName
8   |   location = var.location
9   |
10  |   tags = {
11  |     environment = "Dev"
12  |   }
13  | }
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup> ls

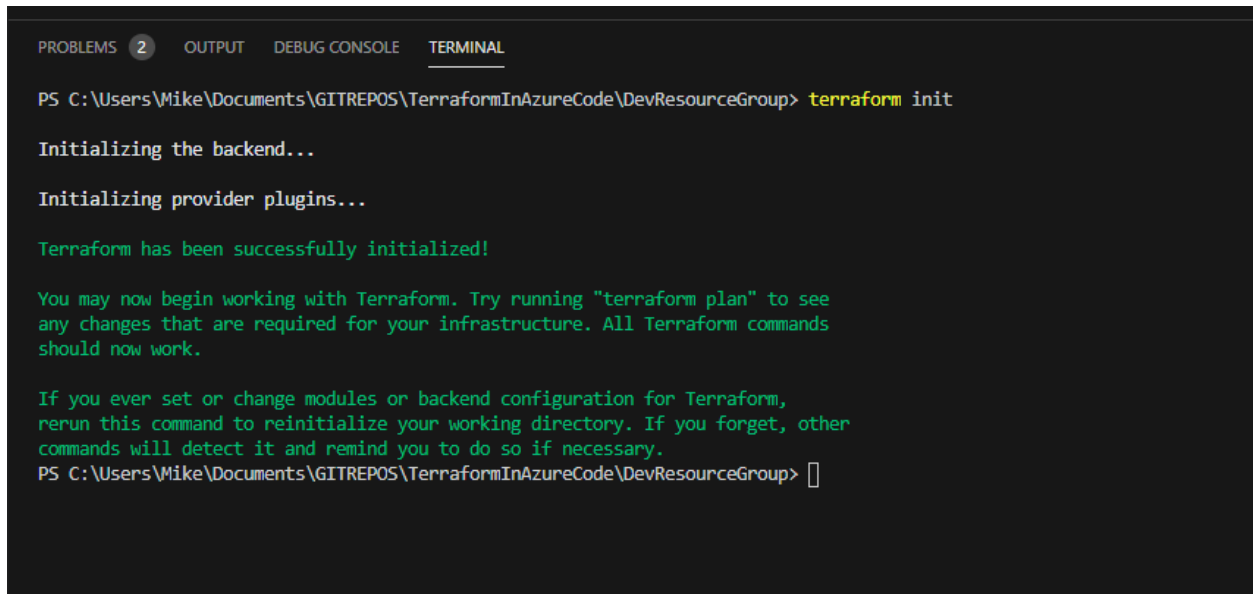
Directory: C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup

Mode                LastWriteTime         Length Name
----                -
d-----          1/23/2020  7:46 AM             .terraform
-a---          1/23/2020  7:45 AM             246 main.tf
-a---          1/23/2020  7:45 AM             104 terraform.tfvars
-a---          1/23/2020  7:46 AM             302 variables.tf

PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup>
```

Directory

On your terminal, run `terraform init` to initialize your environment.

A screenshot of a terminal window with a dark background. At the top, there are tabs labeled 'PROBLEMS', '2', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active. The terminal shows the command 'terraform init' being executed in a PowerShell prompt. The output includes messages about initializing the backend and provider plugins, followed by a success message and instructions on how to use Terraform. The prompt ends with a cursor.

```
PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup> terraform init

Initializing the backend...

Initializing provider plugins...

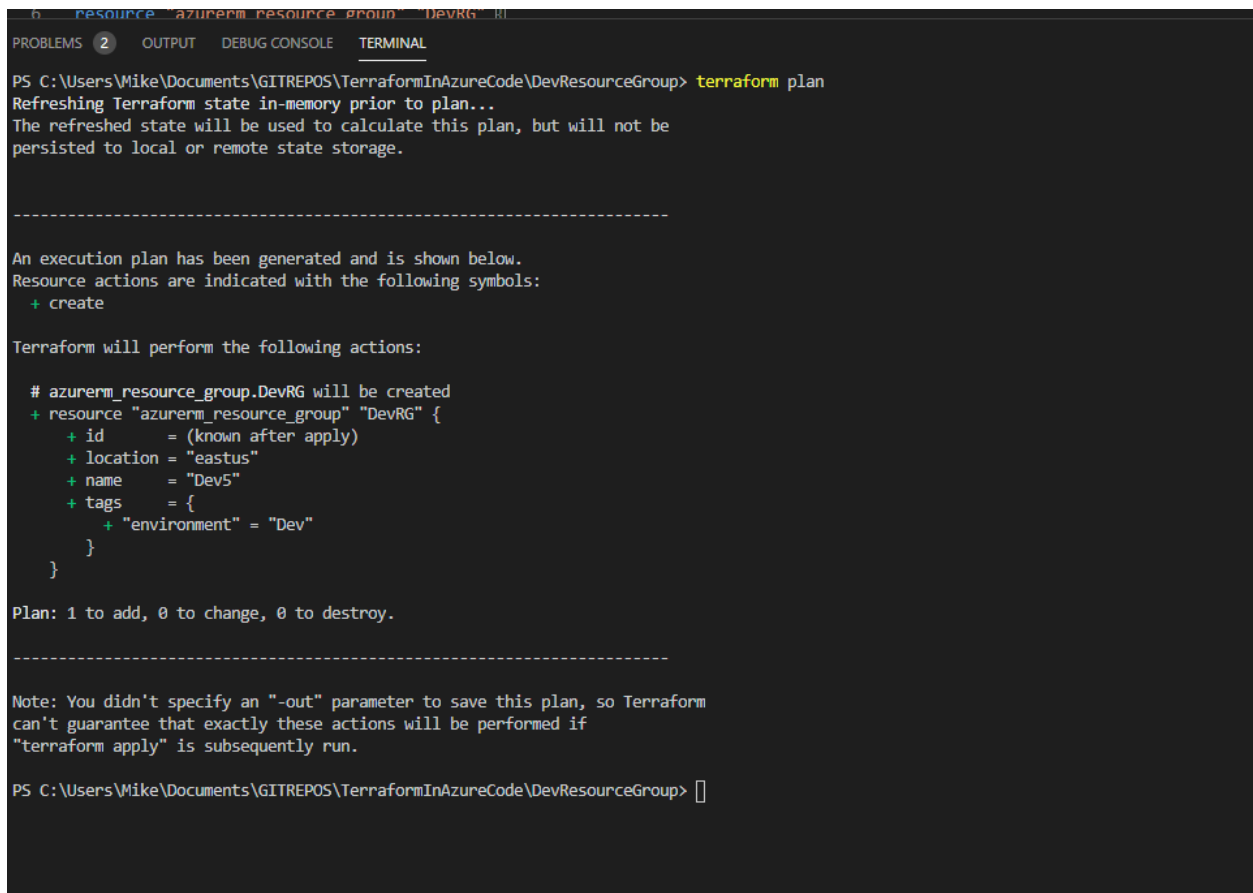
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup> 
```

### Initialize

Now let's run a terraform plan to ensure there are no errors and we see the output of what will be created.



```
6 resource "azure_rm_resource_group" "DevRG" {}
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup> terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azure_rm_resource_group.DevRG will be created
+ resource "azure_rm_resource_group" "DevRG" {
  + id          = (known after apply)
  + location    = "eastus"
  + name        = "Dev5"
  + tags        = {
    + "environment" = "Dev"
  }
}

Plan: 1 to add, 0 to change, 0 to destroy.

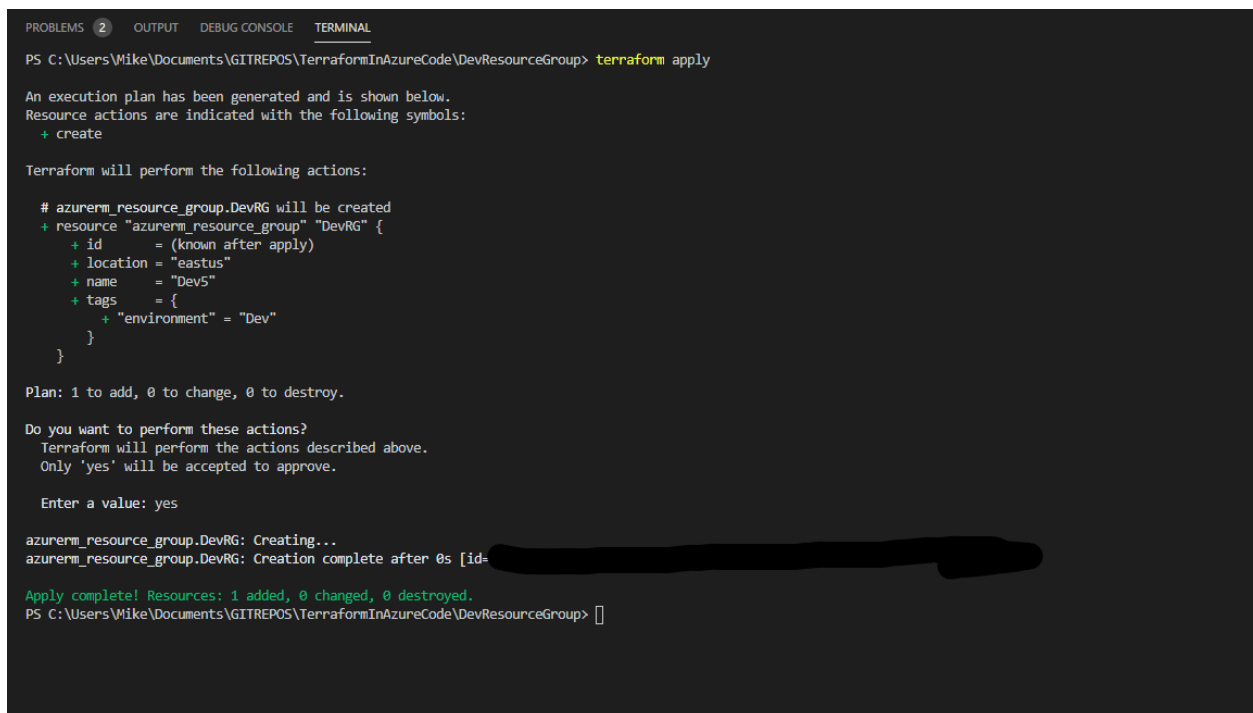
-----

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup> 
```

### Plan

Our plan looks good so we're now ready to create our resource by running `terraform apply`. Ensure you enter a value of `yes` when prompted.



```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup> terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azurem_resource_group.DevRG will be created
+ resource "azurem_resource_group" "DevRG" {
+   id       = (known after apply)
+   location = "eastus"
+   name     = "Dev5"
+   tags     = {
+     "environment" = "Dev"
+   }
}

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

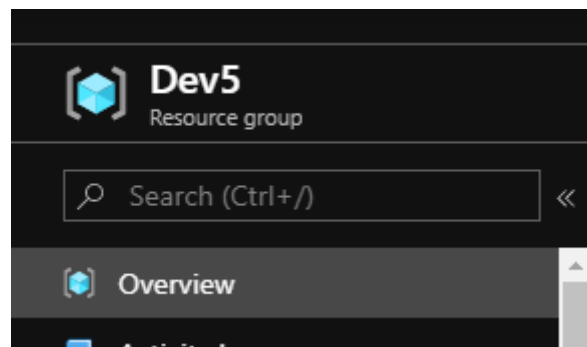
Enter a value: yes

azurem_resource_group.DevRG: Creating...
azurem_resource_group.DevRG: Creation complete after 0s [id=REDACTED]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\DevResourceGroup> 
```

Apply

If we head out over our Azure portal, we'll see the newly created Resource Group!



Resource Group

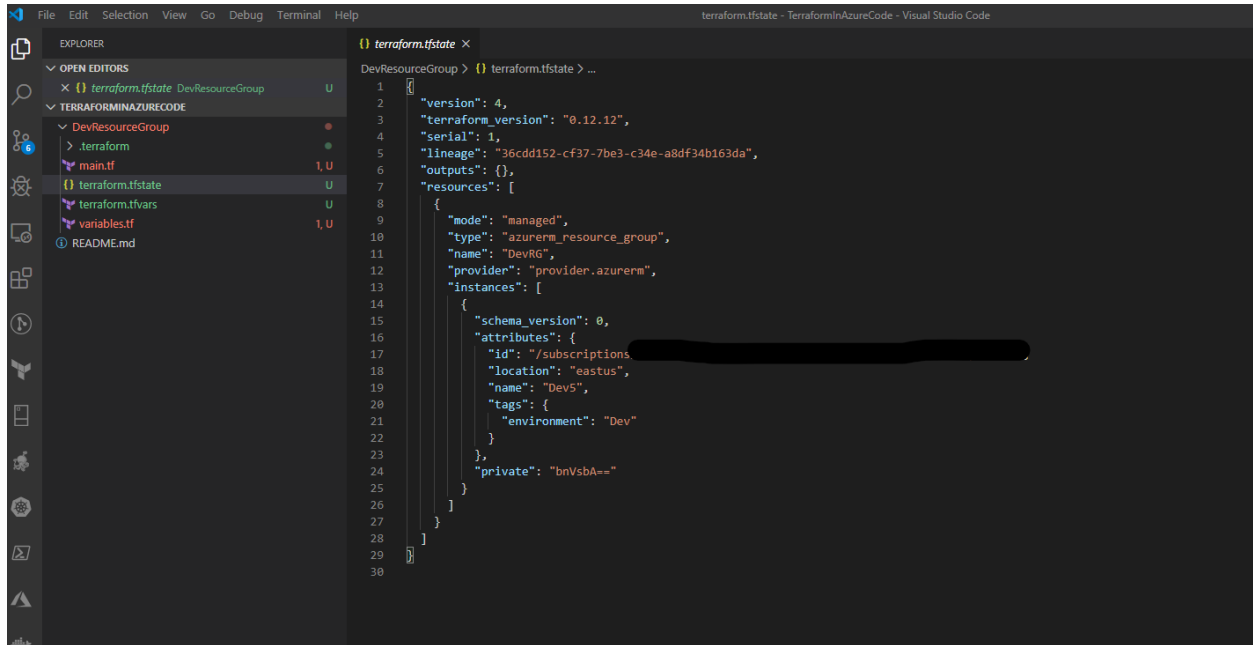
## Summary

Terraform simply makes our lives easier and more efficient. Something as simple as creating a resource group or as complex as an entire Dev environment (which we'll see later) can be created by Terraform. The beautiful thing is this is in code, so it's reusable and far more efficient than any manual process. In the next chapter we'll see how to create an entire network using Terraform.

One more important thing I want to point out. After you ran your Terraform code, you'll see a new config called `terraform.tfstate` get automatically generated. This is the metadata configuration of



your resource(s). Terraform has to store state data about your cloud resources that are being created. The “state” is used by Terraform to map resources to our configuration.



```
1  {
2    "version": 4,
3    "terraform_version": "0.12.12",
4    "serial": 1,
5    "lineage": "36cdd152-cf37-7be3-c34e-a8df34b163da",
6    "outputs": {},
7    "resources": [
8      {
9        "mode": "managed",
10       "type": "azurerm_resource_group",
11       "name": "DevRG",
12       "provider": "provider.azurearm",
13       "instances": [
14         {
15           "schema_version": 0,
16           "attributes": {
17             "id": "/subscriptions/[REDACTED]",
18             "location": "eastus",
19             "name": "Dev5",
20             "tags": {
21               "environment": "Dev"
22             },
23             "private": "bnVsbA=="
24           },
25           "private": "bnVsbA=="
26         }
27       ]
28     }
29   ]
30 }
```

## TFSTATE

# Chapter 4

# Creating a network

Creating a network, like on-prem, is the core infrastructure needed. Without it, your components (VMs, serverless apps, containers, etc.) cannot send packets to each other for communication. You also will not be able to do something simple like reach the internet. The other big component, and possibly the biggest component, is security. You need a way to block certain traffic, log traffic, and protect the environment.

We're going to take a look at the Terraform resources needed to create a network security group, network security group rules, a vNet, a firewall, then put it all together.

## Setting up our storage account for TFSTATE

Before we get started on our network, let's remember the last thing we saw in Chapter 3 which was the TFSTATE. If you noticed while we ran `terraform plan` or `terraform apply`, the resources that were going to be creating for our environment were shown. How did Terraform know what it was supposed to create and what it was supposed to manage? The way Terraform knows the information about what cloud resources we're creating or managing is from the Terraform state file (TFSTATE).

Let's talk about two primary reasons why we want to store the TFSTATE configuration in a shared space.

- If the TFSTATE is not in a shared location, how will your team members use it? Each member of your team needs access to the Terraform state files to build upon/create the same resources you are. If you're working on a Dev environment at home, this isn't really important because you're the only one creating and/or managing resources. If you're working on a team, this becomes necessary.
- From a configuration standpoint, there is a concept in Terraform called "locking". As soon as you share data, you run into this issue, which isn't actually an issue because there's a reason for it. If you didn't have locking, two engineers could be running Terraform at the same time and you could run into race conditions because multiple Terraform processes are running at the same time.

Because of the reasons above, we want to store the TFSTATE in a shared location. To do that we'll store our TFSTATE in a storage account by calling a backend resource in Azure. What's a backend? A Terraform backend how your Terraform state is loaded and how operations like `apply` or `plan` are executed.

The first thing we'll need to do is create a storage account to store our Terraform state.

```
1 $RG = "your_resource_group"
2 # Storage account name with random number attached to make the storage account unique
3 $storageAccountName = "devtfstate$(1..100 | Get-Random -Count 5 | Select -First 1)"
4 $containerName = "tstate"
5
6 # Create storage account
7 az storage account create -g $RG -n $storageAccountName --sku Standard_LRS --encrypt\
8 ion-services blob
9
10 #Retrieve primary connection key
11 $storageAccountKey = $(az storage account keys list -g $RG --account-name $storageAc\
12 countName --query [0].value -o tsv)
13
14 # Create container
15 az storage container create -n $containerName --account-name $storageAccountName --a\
16 ccount-key $storageAccountKey
```

Let's go over the above code;

1. The first set of variables are the names of your resource group, storage account, and container.
2. The `az storage account create` AZ CLI command creates a storage account for you.
3. The `$storageAccountKey` variable calls the storage account primary key for creating the container because authentication is required
4. The `az storage container create` AZ CLI command is to create a container to store your Terraform state in.

Once your Terraform storage account is created, we can look at the terraform code needed. First, let's set up our backend.

```
1 terraform {
2   backend "azurerm" {
3   }
4 }
```

As you can see, we're calling the `azurerm` backend. Next we'll start adding in our parameters.

```
1 terraform {
2   backend "azurerm" {
3     resource_group_name = "Dev5"
4     storage_account_name = "devtfstate65"
5     container_name       = "tstate"
6     key                   = "terraform.tfstate"
7   }
8 }
```

Above you will see that I hard-coded values for the purposes of demonstrating what the Azure backend will look like. It'll make more sense to store the resource group name, storage account name, and container name in variables like we saw in Chapter 3. The key `terraform.tfstate` can stay hard-coded.

Now that our Terraform state configuration is ready, we can proceed with creating a network security group. The Terraform backend config will travel with us through the rest of our configurations in this book.

## Creating a network security group

The first portion of our network we'll create is a network security group. A network security group contains a list of rules. These rules enable inbound or outbound traffic for both private and public connections. This is very important from not only a management perspective, but a security perspective. If you don't pay close attention, you could have a component public facing and not even realize. For example, a SQL server or a domain controller. Both of which we of course don't want public facing.

To create a network security group, we'll need to use the `azurerm_get_security_group` resource.

The first thing we'll do is bring over our Terraform provider (`azurerm`) and backend configuration.

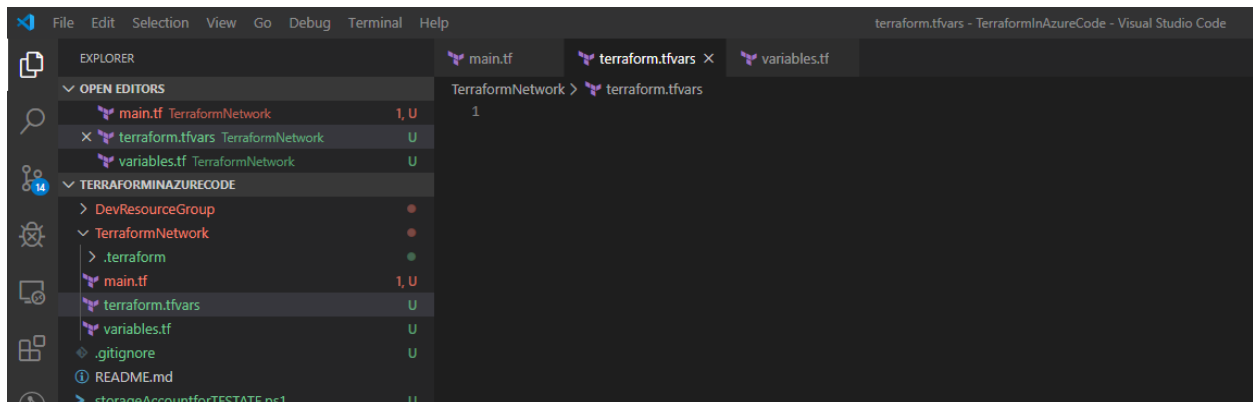
```
1 provider "azurerm" {
2   version = "1.38.0"
3   subscription_id = var.subscriptionID
4 }
5
6 terraform {
7   backend "azurerm" {
8     resource_group_name = "Dev5"
9     storage_account_name = "devtfstate65"
10    container_name       = "tstate"
11    key                   = "terraform.tfstate"
12  }
13 }
```

Next let's set up our resource. We'll be using the "azurerm\_network\_security\_group" resource. This resource will give us the 3 parameters we want to use to set up your security group:

- name of the security group
- location of the security group
- resource group that the security group will live in

```
1 resource "azurerm_network_security_group" "DevNet" {  
2     name =  
3     location =  
4     resource_group_name =  
5 }
```

Since we want to have reusable code, let's create `variables.tf` and `terraform.tfvars` configurations like we did for creating our resource group.



### Variables

Let's open up our variable configuration and create the three variables we want;

- name
- location
- resource group

```
1 variable "subscriptionID" {
2     type = string
3     description = "Variable for our resource group"
4 }
5
6 variable "name" {
7     type = string
8     description = "Name of security group"
9 }
10
11 variable "location" {
12     type = string
13     description = "region that security group will exist in"
14 }
15
16 variable "RG" {
17     type = string
18     description = "resource group that security group will live in"
19 }
```

Next we'll open up our `terraform.tfvars` configuration to input our values for runtime

```
1 name = "devnetsg"
2 location = "eastus"
3 RG = "Dev5"
4 subscriptionID =
```

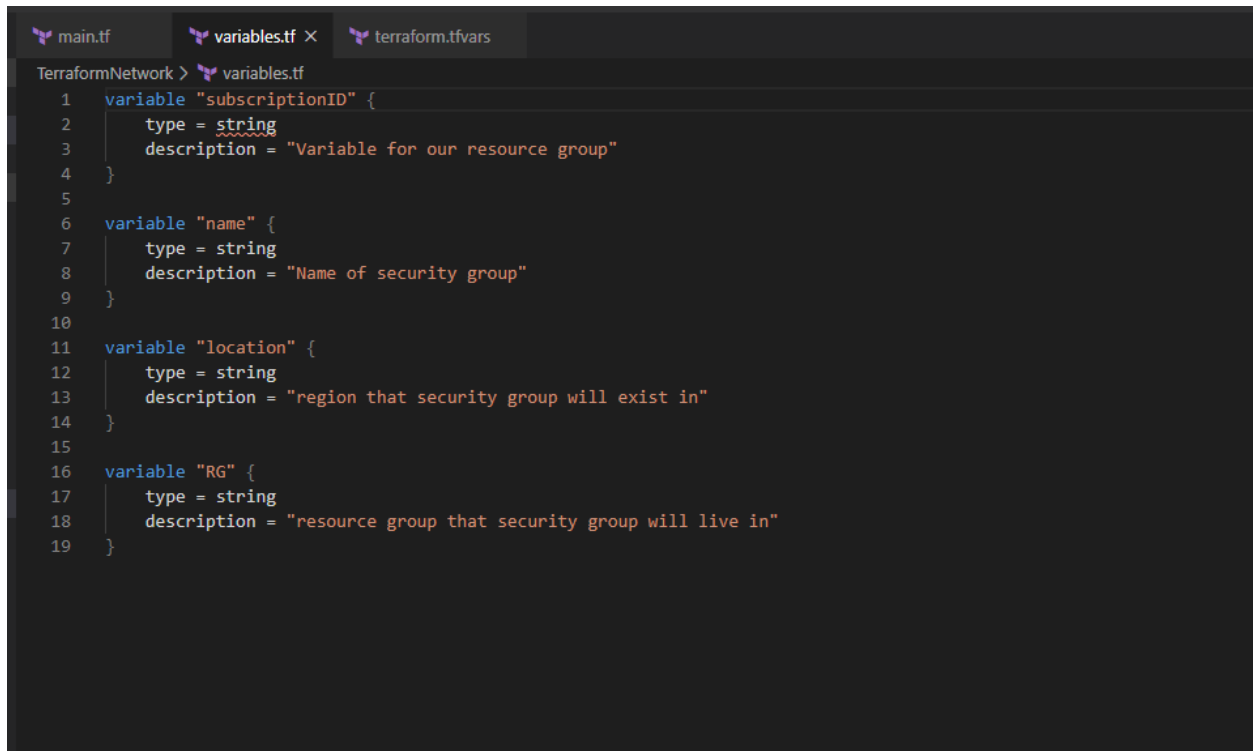
Finally let's go into our `main.tf` and start plugging in the variables.

```
1 resource "azurerm_network_security_group" "DevNet" {
2     name = var.name
3     location = var.location
4     resource_group_name = var.RG
5 }
```

```
TerraformNetwork > main.tf
1  provider "azurerm" {
2      |      version = "1.38.0"
3      |      subscription_id = var.subscriptionID
4  }
5
6  terraform {
7      |      backend "azurerm" {
8      |          resource_group_name = "Dev5"
9      |          storage_account_name = "devtfstate65"
10     |          container_name      = "tstate"
11     |          key                  = "terraform.tfstate"
12     |      }
13 }
14
15 resource "azurerm_network_security_group" "DevNet" {
16     |     name = var.name
17     |     location = var.location
18     |     resource_group_name = var.RG
19 }
```

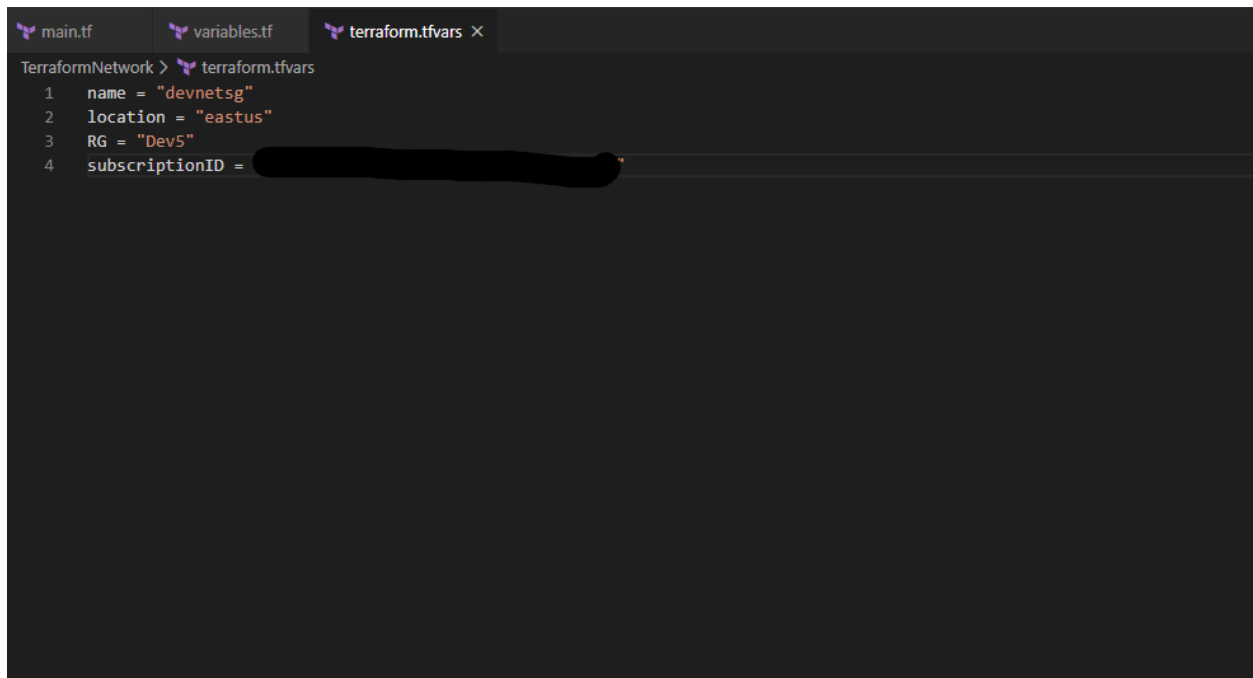
Main





```
main.tf  variables.tf ×  terraform.tfvars
TerraformNetwork > variables.tf
1  variable "subscriptionID" {
2      type = string
3      description = "Variable for our resource group"
4  }
5
6  variable "name" {
7      type = string
8      description = "Name of security group"
9  }
10
11 variable "location" {
12     type = string
13     description = "region that security group will exist in"
14 }
15
16 variable "RG" {
17     type = string
18     description = "resource group that security group will live in"
19 }
```

### Variables



```
main.tf  variables.tf  terraform.tfvars ×
TerraformNetwork > terraform.tfvars
1  name = "devnetsg"
2  location = "eastus"
3  RG = "Dev5"
4  subscriptionID = [REDACTED]
```

### Tfvars

Let's go ahead and run the code. `cd` (change directory) into the directory that has your configuration files.

Run `terraform init` to initialize your environment then run `terraform plan` to ensure the environment looks accurate.

```
PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\TerraformNetwork> terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azure_rm_network_security_group.DevNet will be created
+ resource "azure_rm_network_security_group" "DevNet" {
  + id                = (known after apply)
  + location          = "eastus"
  + name              = "devnetsg"
  + resource_group_name = "Dev5"
  + security_rule     = (known after apply)
  + tags              = (known after apply)
}

Plan: 1 to add, 0 to change, 0 to destroy.

-----

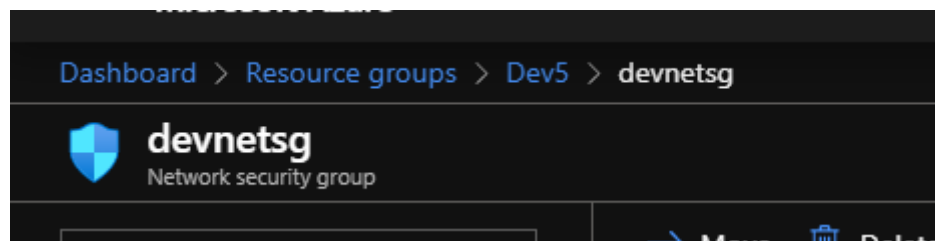
Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

PS C:\Users\Mike\Documents\GITREPOS\TerraformInAzureCode\TerraformNetwork> 
```

plan

We're now ready to create our resource by running `terraform apply`

I'm now able to see my newly created security group in Azure.



Security Group

## Creating a network security group rule

Now that the network security group itself is created, we can start talking about rules. First and foremost, what's a network security group rule? If you've ever worked with UFW on Ubuntu, Windows firewall on Windows, or any type of firewall rules on a network firewall/router, that's

what a network security group rule is. It allows you to allow or deny any inbound or outbound traffic based on a port, like port 80 or port 22.

Let's jump into the code. We'll start with our `main.tf` configuration.

```
1 resource "azurerm_network_security_rule" "Port80" {}
```

The Terraform resource we'll use is `azurerm_network_security_rule`. For our metadata name tag, we'll use `Port80` because we're going to create an inbound rule for port 80. for our parameters, there are a lot. Let's break them down one by one.

```
1  name                = "Allow80"
2  priority            = 102
3  direction           = "Inbound"
4  access              = "Allow"
5  protocol            = "Tcp"
6  source_port_range   = "*"
7  destination_port_range = "80"
8  source_address_prefix = "*"
9  destination_address_prefix = "*"
10 resource_group_name  = azurerm_network_security_group.CloudskillsSG.resource\
11 e_group_name
12 network_security_group_name = azurerm_network_security_group.CloudskillsSG.name
```

- Name = The name is simply a metadata name of your security group rule.
- Prioty = This is the priority of the traffic for the overall network security group rules, depending on how many others you have.
- Direction = The direction is which way you want your traffic to go, inbound or outbound.
- access = Access is to allow or deny the network traffic.
- Protocol = The protocol is what network protocol you're using, TCP or UDP.
- source\_port\_range = The source port range is what ports you want allowed to have inbound connectivity over port 80. Let's say you have a dev environment with ports 6000-6100. You could put this source port range that says "only networks with these ports are allowed for inbound access".
- desination\_port\_range = What port you want to be allowed for this network security group rule. In our case, it's port 80.
- source\_address\_prefix = CIDR or source IP range. You can also put \* to match any IP.
- desination\_address\_prefix = CIDR or source IP range. You can also put \* to match any IP.
- resource\_group\_name = The resource group name that your network security group rule is in.
- network\_security\_group\_name = This is the name of the network security group that we created in the previous section.

```

1 resource "azurerm_network_security_rule" "Port80" {
2   name                = "Allow80"
3   priority            = 102
4   direction          = "Inbound"
5   access              = "Allow"
6   protocol            = "Tcp"
7   source_port_range   = "*"
8   destination_port_range = "80"
9   source_address_prefix = "*"
10  destination_address_prefix = "*"
11  resource_group_name   = var.RG
12  network_security_group_name = azurerm_network_security_group.DevNet.name
13 }

```

*wipes sweat off forehead* Yeah, that was a lot. Being specific in your security group rule is crucial. It could be the difference between your Devs having access to your environment and the entire world having access to your network. Let's be honest, no one wants to have one of their servers end up on Shodan.

Luckily for this portion we don't have to create and new variables or tfvars. Putting it all together will look like the below:

## Creating a vNet

In the previous sections of Chapter 4 we saw how to create resources that are needed for a Virtual Network, but what about the actual network itself? Let's jump into that.

For the Terraform resource, we're going to use the `azurerm_virtual_network` resource. Our metadata tag name will be `vNetDevTest`

```

1 resource "azurerm_virtual_network" "vNetDevNet" {}

```

Let's go over our parameters for the `azurerm_virtual_network` resource.

- `name` = Name of your vNet
- `location` = Region that you want your vNet to be in
- `resource_group_name` = Resource group that you want your vNet to be in
- `address_space` = Here's where things start to get interesting. The address space is your CIDR range for your virtual network. You cannot choose any CIDR that you currently have within your environment already.
- `dns_servers` = Your DNS servers for your organization. If you don't have any DNS servers, you can just use Googles (8.8.8.8 or/and 8.8.4.4)
- `Tags` = Tags is a hash table of any specific tags you want to give your vNet

```

1 resource "azurerm_virtual_network" "vNetDevNet" {
2   name                = "vNetDevNet"
3   location            = var.location
4   resource_group_name = var.RG
5   address_space       = ["10.0.0.0/16"]
6   dns_servers         = ["8.8.8.8", "8.8.4.4"]
7
8   tags = {
9     environment = "Dev"
10  }
11 }

```

Creating the virtual network is pretty straight forward, but you won't have any IPs without a proper subnet. Let's create that as well.

For the Terraform resource we'll use `azurerm_subnet` and give a metadata name of `devnet-sub`

```

1 resource "azurerm_subnet" "devnet-sub" {}

```

Let's go over the parameters for the `azurerm_subnet` resource

- `name` = Name of your subnet group
- `location` = Region that you want your subnet to be in
- `resource_group_name` = Resource group that you want your subnet to be in
- `address_prefix` = Address prefix is the subnet range. You must choose a subnet range that's in your virtual network CIDR.

```

1 resource "azurerm_subnet" "devnet-sub" {
2   name                = "testsubnet"
3   resource_group_name = var.RG
4   virtual_network_name = azurerm_virtual_network.vNetDevNet.name
5   address_prefix      = "10.0.1.0/24"
6 }

```

Woohoo! We now have our entire virtual network, security groups, rules, and subnet created. Let's put it all together.

## Putting it all together

In the previous sections we saw bits and pieces of our network and how to create it. Now we're ready to put it all together.

Below you will see the entire code base for the network setup. This code base is also available at [<https://github.com/AdminTurnedDevOps/TerraformInAzureCode/tree/master/TerraformNetwork>]

```
1 provider "azurerm" {
2     version = "1.38.0"
3     subscription_id = var.subscriptionID
4 }
5
6 terraform {
7     backend "azurerm" {
8         resource_group_name    = "Dev5"
9         storage_account_name   = "devtfstate65"
10        container_name          = "tstate"
11        key                     = "terraform.tfstate"
12    }
13 }
14
15 resource "azurerm_network_security_group" "DevNet" {
16     name = var.name
17     location = var.location
18     resource_group_name = var.RG
19 }
20
21 resource "azurerm_network_security_rule" "Port80" {
22     name                = "Allow80"
23     priority             = 102
24     direction           = "Inbound"
25     access              = "Allow"
26     protocol             = "Tcp"
27     source_port_range    = "*"
28     destination_port_range = "80"
29     source_address_prefix = "*"
30     destination_address_prefix = "*"
31     resource_group_name  = var.RG
32     network_security_group_name = azurerm_network_security_group.DevNet.name
33 }
34
35 resource "azurerm_virtual_network" "vNetDevNet" {
36     name                = "vNetDevNet"
37     location             = var.location
38     resource_group_name = var.RG
39     address_space        = ["10.0.0.0/16"]
40     dns_servers          = ["8.8.8.8", "8.8.4.4"]
41
42     tags = {
43         environment = "Dev"
44     }
45 }
```

```
44     }
45 }
46
47 resource "azurerm_subnet" "devnet-sub" {
48     name                        = "testsubnet"
49     resource_group_name       = var.RG
50     virtual_network_name     = azurerm_virtual_network.vNetDevNet.name
51     address_prefix            = "10.0.1.0/24"
52 }
```

## Summary

There's a beauty behind Infrastructure-as-code when you look at it whole. I can't even remember how long it would take me to create a network manually because it's been such a long time since I've done it.

When people talk about codified infrastructure, they often times talk about the automation aspect and how fast you can go. Let's take that out of the equation for a second and think about the pure accuracy of it all. Humans make mistakes. We click the wrong button, we type the wrong number, we create something in the wrong place. Mistakes happen, we're human, but code doesn't lie. Not to say that you can't type the wrong letter or number into code, but with code and source control, you can review it. Others can review it as well. You can a trail leading back to what was done. It's pure precision at it's finest.