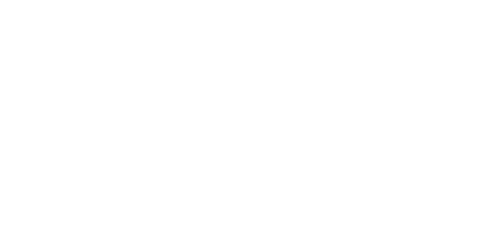
|  |
| --- |
| Infrastructure as code  Using Terraform and Ansible |
| Cloud Computing Project By (Group 9) :  Faiz Usmani 1323197  Parag Tambalkar 1322596  Pranay Raman 1321759  Shubham Girdhar 1323003  Prof name |





## Content

**3**

3

4

5

6

6

**7**

**8**

8

9

10

18

**20**

20

23

**26**

### **1. Introduction**

* 1. The Why’s and What’s
  2. Terraform – The Maestro Provisioner
  3. Ansible – The Configuration expert
  4. Azure and Cloud Shell – Cloud and Command Line Specialists
  5. Visual Studio 2019 – The Pro IDE

### **2. Deployment Plan**

### **3. Provisioning Infrastructure with Terraform**

### Understanding Terraform Workflow

### Prerequisites

### Understanding the Terraform code

### Bringing up the Infrastructure with Terraform

### **4. Installing and Configuring IIS with Ansible**

### 4.1 Understanding the Ansible code

### 4.2 Invoking Ansible Playbook

### **5. Deploying the Application using Visual Studio 2019**

### **Introduction**

* 1. **The Why’s and What’s**

Before the concept of DevOps, the deployment of applications into various environments was a daunting task requiring time, effort, and skills. A separate Operations team with extensive knowledge and experience of various servers, Virtual Machines (VMs), and host Operating System’s (OS) used to manually access the servers and configure them according to the requirements of the application. In the case of small organizations usually, developers of the application were in charge of doing this (well, not a good sight to imagine).  
  
Now, thanks to the advancements in the field of cloud and DevOps in recent years, these tasks are becoming less complex. The virtual servers on cloud and onsite physical servers can be provisioned and configured easily in minutes just by executing a script written in a user-friendly language (mostly YAML). This is possible by a concept known as Infrastructure as Code (IaC) which allows us to deal with servers, networks, security groups, databases, etc. as if they are a part of the software. The leading solutions in the market nowadays for IaC are provided by Terraform, Ansible, Chef, Puppet, etc.

Let’s look at some of the advantages offered by Infrastructure as Code:

1. **Code Once, Iterate Multiple times:** As the configuration files are coded initially for creating and configuring the environment for an application. These files can be used to spin up multiple environments (for example testing, production, staging, etc.) and can also be used to deploy similar environments for applications with the same OS and package requirements. They can also be tweaked a little bit if the configurations change and Voila! We have a new environment setup and ready to go.
2. **Immune to Human error:** As before, the infrastructures were configured manually and by different people, there used to be errors known as infrastructure deviations. These were minor differences in the configuration of environments due to the manual process of deployment by different people on multiple servers. Now, because of using scripts for provisioning and configurations, this process is standardized.
3. **Scaling, a piece of cake!:** When there is high incoming traffic in your application at a particular time of the day or year, you can create new environments with just a few clicks and commands and deploy multiple instances of your application. Similarly, you can bring down environments when there is less traffic leading to cost efficiency. Scaling environments up and down is as easy as it gets.

Now, let‘s have some basic understanding of the tools we will be using in this project.

* 1. **Terraform – The Maestro Provisioner**

Think of Terraform as a Master Civil Engineer/Builder, you will provide him with what you need and he will go ahead and build it for you in a matter of minutes. Also, before building the actual thing he will inform and take your consent about the plan he is going to implement.

In the language of Software development – It is a tool for creating, modifying, and versioning infrastructures safely, swiftly, and efficiently.

General structure of Terraform:  
main.tf – In the case of simple projects this is the file that has all the code to build up infrastructure, it contains all the modules and data sources that are needed. If the project is bigger and complex, the files are logically split based on the functionalities.

variables.tf – It stores the declarations for the variables which are referenced in main.tf.

terraform.tfvars – It is used to define the default values of the variables.

The configuration files written in YAML describe the resources and components which are required to run your application. Terraform will then go ahead and produce an execution plan describing the steps to reach the desired state, and then goes ahead and executes it. If there are changes that are applied to the configuration files, terraform sees what is changed and based on it creates incremental execution plans.

Terraform is supported by multiple cloud providers and the infrastructure that can be provisioned by it includes application instances, networking resources, databases, SaaS features, DNS items, etc.

Also, the main difference between Terraform and the other IaC tools is that it does not re-provision resources that are successfully provisioned if it encounters them in some future execution of the configuration script. Instead, it focuses on the new additions and the failed resources from the previous executions.

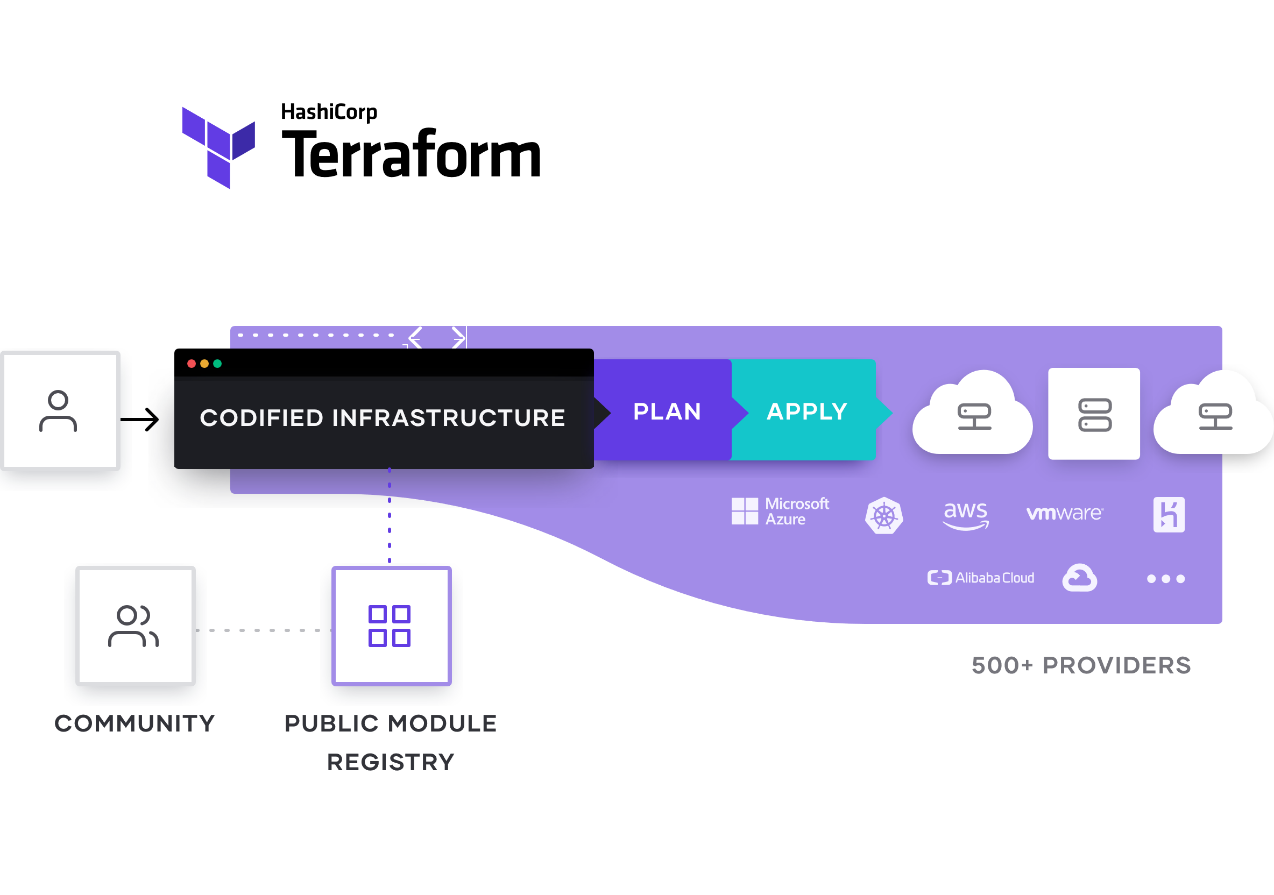


Figure: <https://www.terraform.io/_next/static/images/how-terraform-works-33c33a9c82bf5aef0bf9f75eb5f5f9b2.svg>

* 1. **Ansible – The Configuration Expert**

Now, after having the basic infrastructure, you find out that it requires some modifications and touchups. For this Ansible is your go-to-guy. Think of it as an experienced craftsman.  
  
Although Ansible is a tool that can do a lot more than configuring existing infrastructures. It is primarily known for the configuration tasks as it is very easy to do it using Ansible. A whole network of computers can be configured using Ansible at once.   
  
The primary architecture of Ansible consists of two kinds of nodes – Control and Managed nodes.  
A Control node is the main system that has Ansible installed on it, while the managed nodes are the clients that are connected to the Master node using SSH (secure shell) or any other authentication techniques.

The Control node has a host inventory file containing the addresses of the Managed nodes.  
The Control node sends configuration programs which are known as modules to the Managed nodes.  
These programs are saved as Ansible playbooks (name of the Ansible specific configuration files written in YAML) on the Control node. These modules compare the states of various Managed nodes to what is mentioned in the modules, finds the mismatch, and then updates the state of the managed nodes.

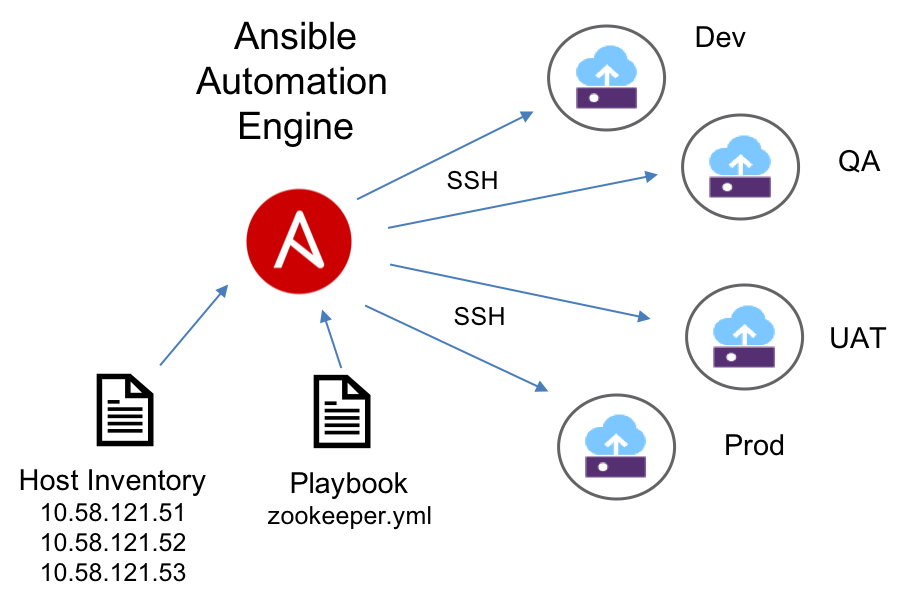


Figure: [Ansible\_ov.png (904×602) (s81c.com)](https://1.cms.s81c.com/sites/default/files/2018-11-22/Ansible_ov.png)

* 1. **Azure and Cloud Shell – Cloud and Command line Specialists**

**Azure** is a cloud computing service provider by Microsoft. It offers a multitude of services including Software as a Service, Infrastructure as a Service, Platform as a Service, etc. In the context of our project, it will provide space and resources for our infrastructure to be built in it. The Virtual Machines which will be built by Terraform and configured by Ansible will be hosted on Azure Cloud.

Alas! Like most of the things in the world, it is not free to use but comes with a flexible charging plan. If you have a student email ID it will provide you with a credit of 100 dollars for a period of 1 year and for a general email ID it gives credit of 200 dollars for 1 month. On a side note do not forget to clean up the resources after successfully executing and testing your project, because keeping them on Azure for a long period of time may turn out to be a costly affair.

Here is the link to get you started:

[Azure for Students – Free Account Credit | Microsoft Azure](https://azure.microsoft.com/en-us/free/students/)

**Azure Cloud Shell** is a browser integrated shell that is used to manage resources on Azure. You can use PowerShell, Bash, or Azure CLI (a cross-platform command-line tool that can be used to run commands on the resources present in Azure Cloud) on it. Head over to the Azure portal and click on the command line icon at the top near the search bar to get going.

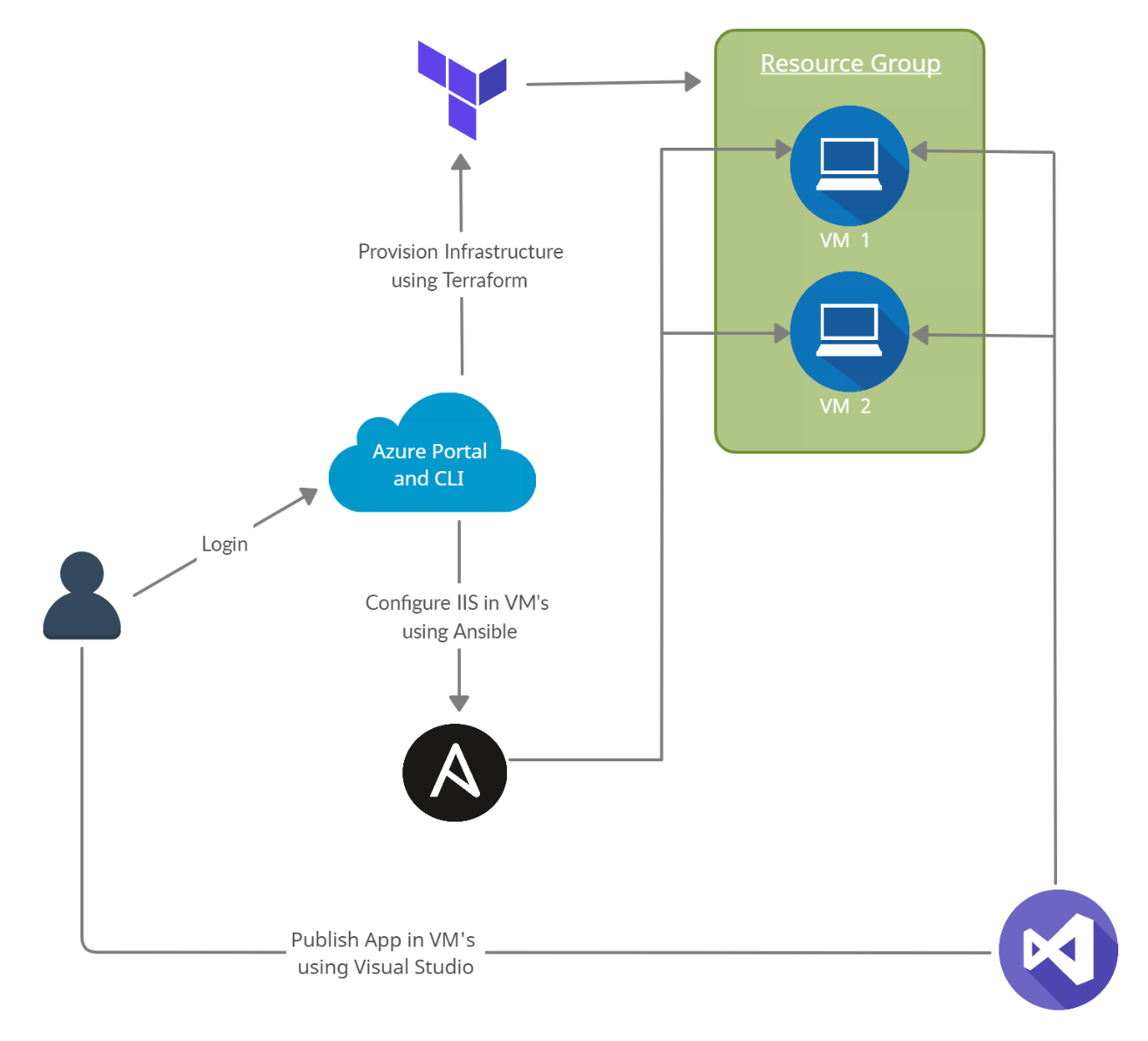
* 1. **Visual Studio 2019 – The Pro IDE**

A powerful Integrated Development Environment that can be used to develop software and programs for windows, web apps, services, etc. The reason for choosing it was because of the ease by which the ASP.NET web apps can be easily published on Azure VMs through its interactive User Interface.

Just make sure while installing Visual Studio 2019, these workloads are also installed – “ASP.NET and web development”, “Azure development”, “.NET Core cross-platform development”. If Visual Studio 2019 is already installed, make sure these workloads are present by opening up the “Visual Studio Installer” through windows search and then clicking on the “Modify” button and checking the workloads.  
  
Note – The email id used to sign-in into Visual Studio and Azure portal must be the same as the email id used for signing in for Windows. Otherwise, during publishing the apps into VMs, some errors may occur due to permissions issues. If the email ids are different, the id which is used in the Azure portal should be added into the appropriate permissions of the folder where the ASP.NET app is present.

### **2. Deployment Plan**

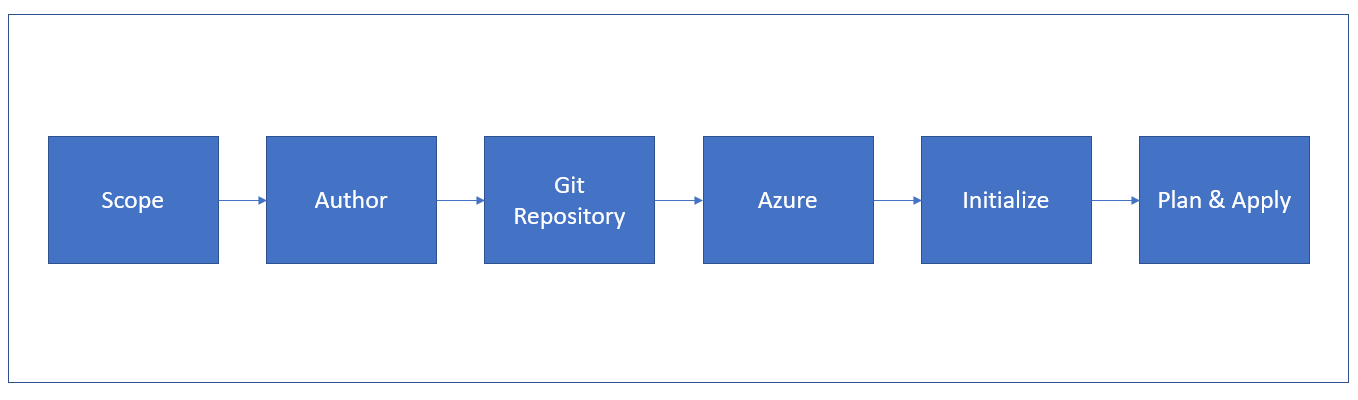
Now, after having some basic understanding of the tools and technologies used in this project. Let’s have a look at the design of the project which we are going to implement.

****

1. The user will login into the Azure portal and use Azure Cloud Shell to run Terraform and Ansible scripts.
2. Terraform scripts are run and a resource group with two Windows VMs having proper network configurations are created.
3. After the successful creation of VMs, IIS (Internet Information Services – a web server software package specifically designed for windows) and some other modules are configured on both of the VMs using their IP’s. They are required to run an ASP.NET web app on a Windows server.
4. A self-created web app or app cloned from the provided GitHub link is published on both of the VMs using Visual Studio 2019 and accessed through the public IP of the load balancer.

### **3. Provisioning Infrastructure with Terraform**

### **Understanding Terraform Workflow**



Workflow of Terraform to build the cloud infrastructure will follow the steps shown in the block diagram above. The description of each step in the workflow is provided below,

1. **Scope:** In this step, we identify the resources required for the project. In this particular project we require resources such as Virtual Machines, resource groups, network security groups, load balancer, etc.
2. **Author:** In this step, we need to create a configuration of identified resources using HashiCorp Configuration Language (HCL). A detailed overview of this process is provided in the ‘Building the Infrastructure with Terraform’ section of this report.
3. **Git Repository:** After creating terraform configuration for the given project we need to add these files in the Git repository of our project.
4. **Azure:** We need to link the project repository at GitHub with the Azure account using which we are deploying our project. Detailed information about this process is mentioned in the ‘Deploying the Application using Visual Studio 2019’ section of the report.
5. **Initialize:** To download the appropriate provider plug-ins for the project and initialize the working directory with Terraform configuration files, we need to run *terraform init* command in the project directory containing terraform configuration files.
6. **Plan & Apply:** Execution of *terraform plan* command creates execution plan to verify infrastructure creation process. Also, terraform performs a refresh in order to determine the necessary actions to be performed to achieve the expected state of the infrastructure. This command helps us to know in advance that the infrastructure being created is matching with the expected infrastructure.

After successful verification of the execution plan, we need to run *terraform apply* command to build real infrastructure with specified resources. It also creates a state file that stores the existing state of infrastructure and compares configuration modification files with existing configuration of deployment environment.

The coming sections will provide information about how we used Terraform in our project.

### **Prerequisites**

**1. Azure Subscription**

We need a valid Azure subscription to deploy the project on cloud. In this project we are using Azure for Students subscription.

**In this project we will be using Azure CloudShell pre-built in Azure portal. If you want to run the scripts using local machine the next steps are necessary.**

**2. Install the Azure CLI tool**

We need to install Azure CLI tool to create and manage Azure resources from PowerShell. In order to install Azure CLI tool, we need to open PowerShell prompt as an administrator and run the command:

Invoke-WebRequest -Uri https://aka.ms/installazurecliwindows -OutFile .\AzureCLI.msi; Start-Process msiexec.exe -Wait -ArgumentList '/I AzureCLI.msi /quiet'; rm .\AzureCLI.msi

**3. Install Terraform**

In order to use Terraform, we need to install Terraform as a binary package (version 0.14.5) from the official website of Terraform.

**4. Authenticate using the Azure CLI**

After successful installation of Azure CLI tool, the next step is to link and authenticate Azure CLI tool with our Azure account. Terraform requires authentication from Azure to create infrastructure. For this we need to log into Azure account locally from the terminal using Azure CLI tool by running the command:

az login

This will open your browser window and ask you to enter your Azure login credentials. After successful authentication, your terminal will display your subscription information.

### **Understanding the Terraform code**

### To run our project in Azure, we need to create the required infrastructure for the web application using Terraform. We have written configuration for all the identified resources in main.tf file and defined some repeatedly used parameters of the variables in terraform.tfvars file. The following steps are needed to create the infrastructure,

### We need to define the service provider as azurem and its version that is 2.0. To create infrastructure, a resource group is required to hold all project-related resources. We defined an Availability set which is the logical grouping for isolating VM resources from each other after deployment.

### 

### We need to create Network Security Group (NSG) to filter the traffic to and from resources in Azure.

### 

1. It is very important to define the rules to filter the traffic. For our project, we have created four different rules for the following purpose:

a) Allowing Ansible to connect with the VMs.

b) Allowing Visual Studio installed in our local machine to interact with Azure to deploy web application.

c) Allowing web clients to connect to our web application.

d) Optional rule if we need Remote Desktop Protocol RDP for troubleshooting purpose.

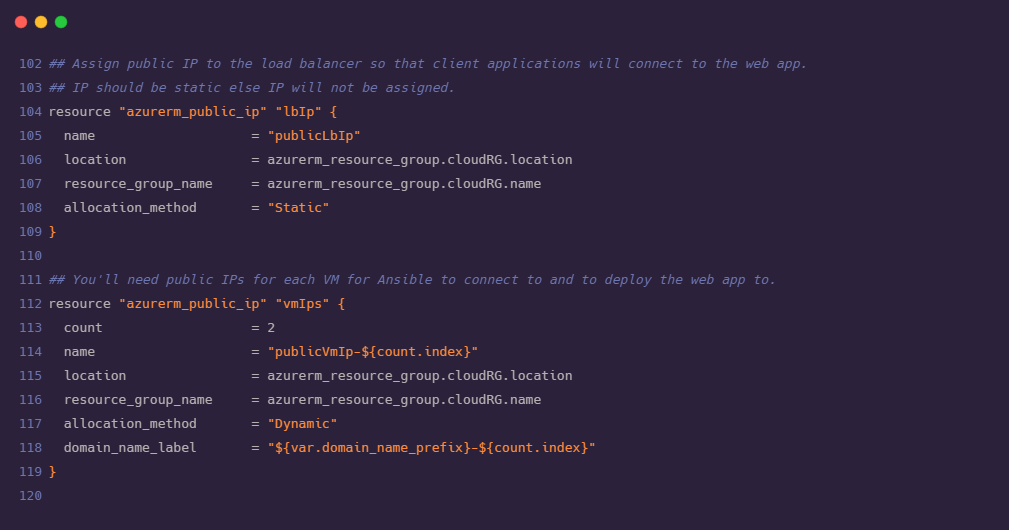




1. Now we need to define the infrastructure for Virtual Network which is the representation of our network in the cloud. We also defined subnets under the Virtual Network.



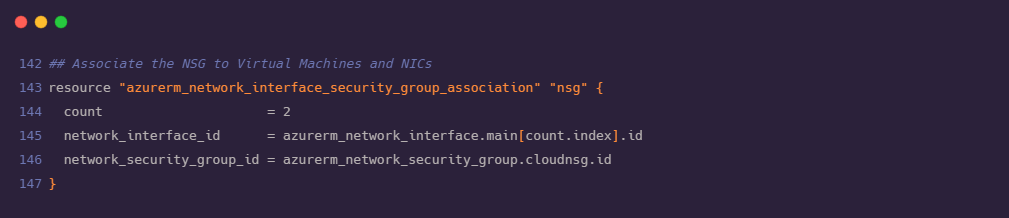
1. We need to assign a static IP address to the load balancer in advance. Also, need to assign public IP address for each VM that we are deploying. We require two VMs for this project.



1. It is important for every VM to connect to the network. For this, we need to create a virtual Network Interface Card (vNIC) for each VM. vNIC severs as a gateway between network and VMs. It is required to provide IP configuration to each of the Network Interface Card. Before creating vNIC it is mandatory to ensure that our subnet is up and running.



1. We need to apply the NSG rules that were created in second and third steps to the VMs and NICs in our network.



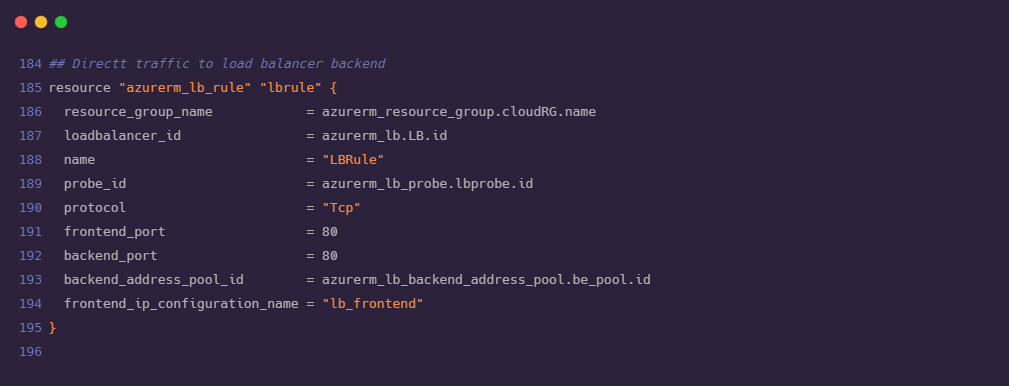
1. Now we are ready with the declaration for creating the VMs. We need a load balancer to rout the traffic to our VMs. The load balancer directs the traffic towards the healthiest VM and avoids redirection of traffic towards failed VM. As load balancer is acting as a mediator between outside traffic and VMs, we need to configure frontend and backend of load balancer where backend part will consist of NICs and VMs.



1. As described in the previous step, load balancer directs the traffic to the healthiest VM in the network. For this we need to configure the health probe for VMs this will periodically check on both VMs for an open port 80.



1. Using the health probe defined in the step above, we need to create a rule to direct the incoming traffic on port 80 to the healthiest VM available in the backend.



1. As an important step we need to associate VMs and vNICs created in earlier steps.



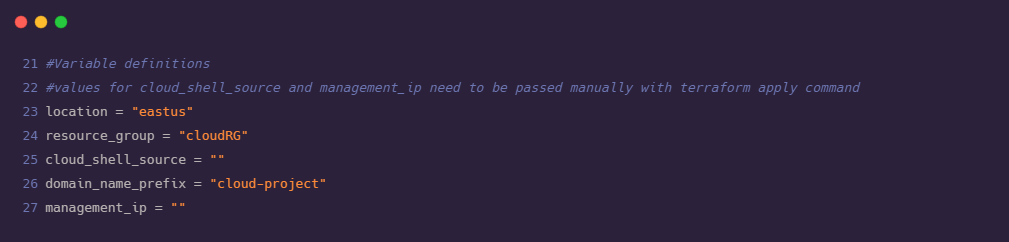


1. We need to install a custom script VM extension on each virtual machine. This will download the ConfigureRemotingForAnsible.ps1 script from GitHub. Execution of this script will open the Windows Remote Management (WinRM). Using WinRM Ansible connects with Azure Cloud Shell.



1. After declaring all the required resources in main.tf file, we are now ready with our Terraform code for the deployment of our infrastructure. Before proceeding to understand Ansible configuration and deployment steps, we need to look at terraform.tfvars and variables.tf file where we have stored values for frequently used variables in order to maintain consistency. The values of variables in these files can be modified without affecting the main configuration.





### **Bringing up the Infrastructure with Terraform**

1. Open the Azure Cloud Shell.

2. Create a new directory for Terraform using the command:

md terraform

3. Now we need to move to the newly created directory. Use the following command to move to the created directory:

cd terraform

4. Once we are in the terraform directory, we need to get the main.tf, terraform.tfvars, variables.tf files from the git repository.

foreach ($file in @('main.tf','terraform.tfvars','variables.tf'))

{

$uri = "https://raw.githubusercontent.com/Shubham-Girdhar/Infrastructure-As-Code/main/terraform/$file"

Invoke-WebRequest -Uri $uri -OutFile $file

}

The syntax above will download all the files from the git repository to the terraform directory which was created in step 2.

5. Now that we have the requisite files available in Azure Cloud Shell, it’s time to run the terraform configuration. We initialize Terraform using the command:

terraform init

6. After the initialization of Terraform we should bring up the plan using the command:

terraform plan

7. Find the public IP address of Azure Cloud Shell session by running the below command. We need to pass this IP to the Terraform configuration.

curl <http://ifconfig.me>

8. Find the local public IP address. This is because Terraform needs to open up a port in the Network Security Group to eventually deploy the web app with Visual Studio which is shown later in deploying the application part. We can find the local IP address by opening up PowerShell on the local machine and running the following with PowerShell:

curl http://ifconfig.me

9. Run the terraform plan to build all the Infrastructure using terraform apply using the syntax below. This process might take some time.

terraform apply -var="location=eastus" -var="resource\_group=cloudRG" -var="cloud\_shell\_source=14.72.123.66" -var="domain\_name\_prefix=cloud-project" -var="management\_ip=<local ip>" -auto-approve

Be sure to fill in the appropriate variable values with the -var parameters.

• location: The location where the Azure resource group is in.

• resource\_group: The name of the resource group to deploy resources into. This variable defaults to NoBSDevopsCloud.

• cloud\_shell\_source: The IP address of the Azure Cloud Shell which is obtained in step 7.

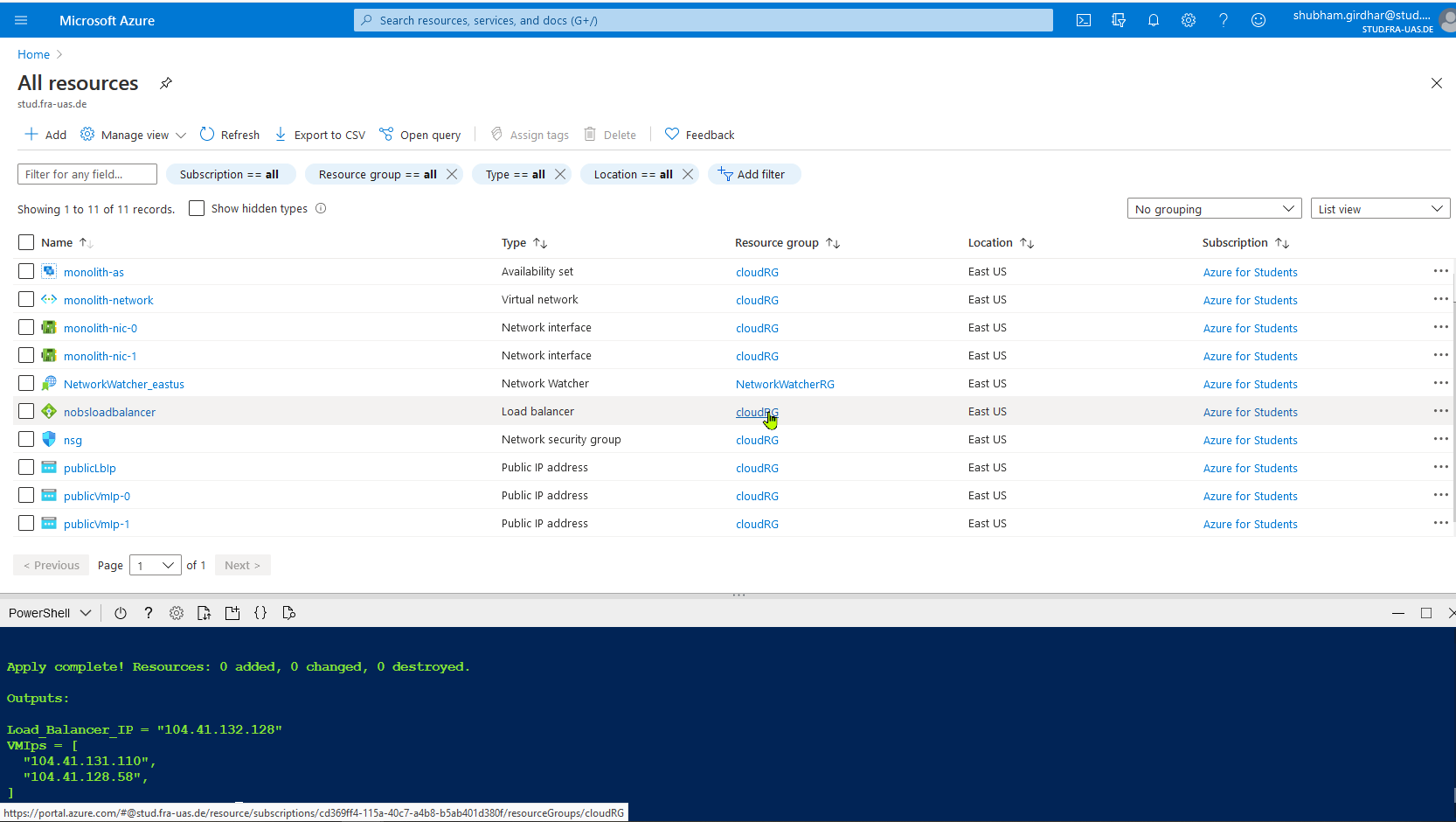
• domain\_name\_prefix: The DNS prefix that will be applied to each VM. This variable name must be unique.

• management\_ip: The public IP address of the local machine which is obtained in step 8.

• auto\_approve: This removes the usual prompt to ask if we really like to do this or not.

10. When terraform apply command is completed, we should have a resource group created with everything that is required for Ansible to take over in the later stages. After the apply syntax is executed successfully terraform will return the public IP address of the load balancer which will be placed in front of the web servers. It will also return both of the public IPs for the VMs. Make sure to note these IPs as it will be required later. If it does not return the IP address on the first run, then run the apply command again.

11. The below image shows the resource groups that are created along with the IP addresses of load balancer and the VMs.



12. By the end of step 11, the Infrastructure is successfully built.

### **4. Installing and Configuring IIS with Ansible**

### **4.1 Understanding the Ansible Code**

### In this step, we will use an Ansible playbook to configure our Windows VMs. After configuration, we will be able to publish .NET CORE web applications to Azure VMs directly using Visual Studio. The playbook prepares the VMs to receive the web application using WebDeploy.

### There are two important parts of the Ansible architecture required for configuration management.

### 1. Ansible playbook: This is the main script with the set of instructions that need to be implemented on multiple hosts.

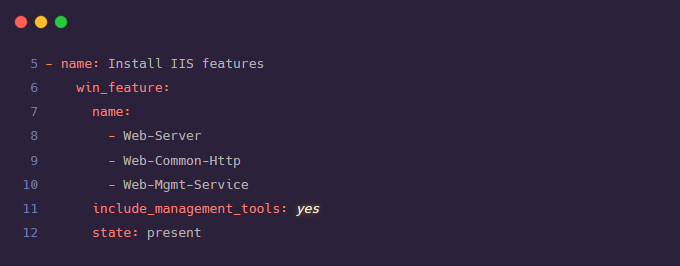
### 2. Inventory: This is the part that maintains the structure of the network environment. We create different nodes and pass these nodes to the playbook, so that it knows which action needs to be implemented on which resource.

**The Ansible playbook (iis.yml) for our deployment will do the following steps:**

1. Installs the basic set of IIS windows features including the Web Management Windows feature.

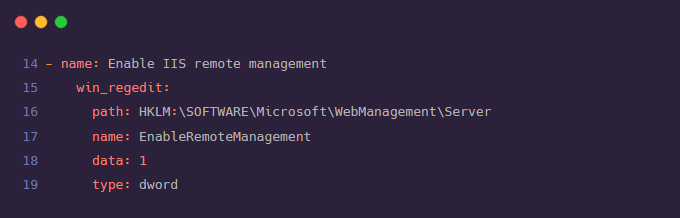
a) win\_feature: installs or uninstalls features or roles on windows server.

b) include\_management\_tools: adds corresponding management tools to specified features



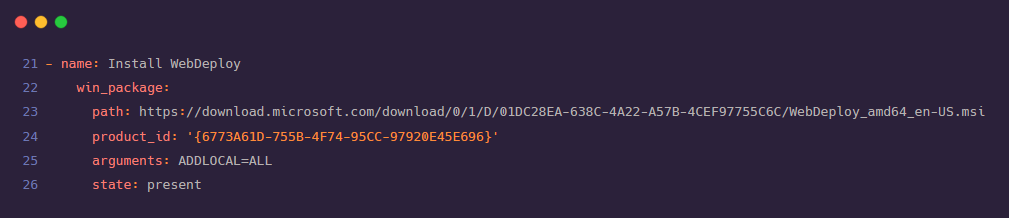
2. Set the registry value to enable remote management of the Web Management service.

a) win\_regedit: add, modify or remove registry keys and values.



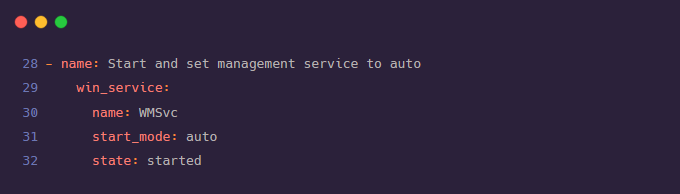
3. Downloads and installs the Web Deploy software package.

a) win\_package: installs or uninstalls software packages for Windows.



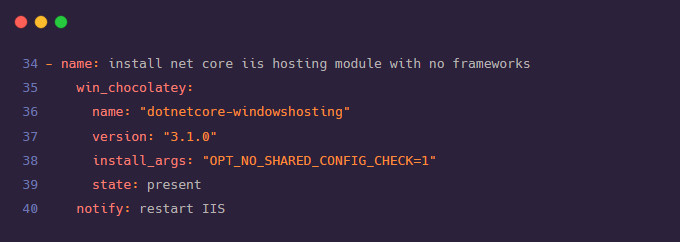
4. Starts and sets the Web management service to automatic.

a) win\_service: manage and query Windows services.



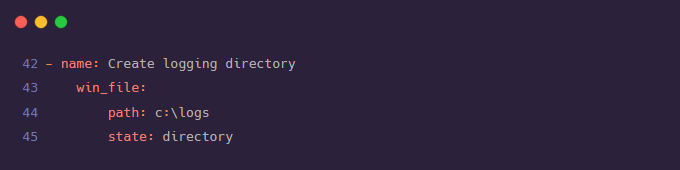
5. Installs .net core IIS hosting module with no frameworks.

a) win\_chocolatey: manage packages using chocolatey. If chocolatey is missing, the module will install it.



6. Create logging directory.

a) win\_file: create or remove files or directories.



**The inventory (hosts) for our ansible includes the following:**

1. [azurevms]: This is a node that contains the list of Windows VMs on Azure that we want to configure.

2. [azurevms:vars]: These are the variables that are needed to make the WinRM connection to work.

a) ansible\_user=testadmin (username)

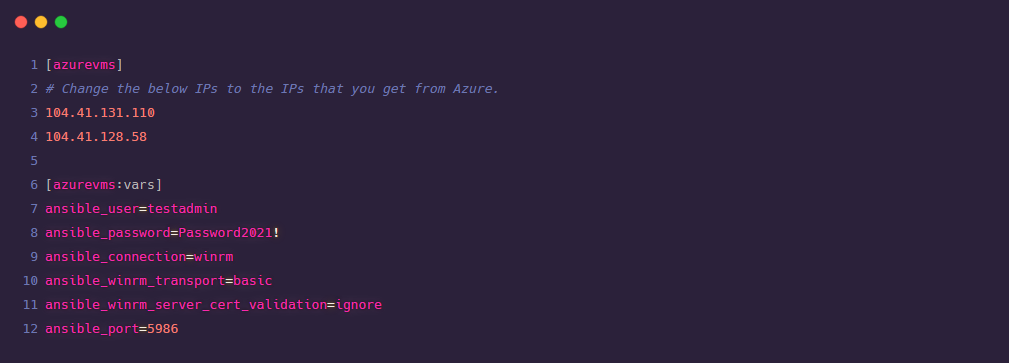
b) ansible\_password=Password1234! (password)

c) ansible\_connection=winrm (this connection is set to SSH by default, so to make the connection to Windows VM to work, this needs to be set as winrm)

d) ansible\_winrm\_transport=basic

e) ansible\_winrm\_server\_cert\_validation=ignore (set to ignore as the certificates are generated locally to virtual machines are self-signed)

f) ansible\_port=5986 (ansible port)



### **4.2 Invoking Ansible Playbook**

### 1. Open the Azure Cloud Shell.

### 2. Make a new directory for Ansible files and change to this directory using the following commands:

### mkdir ansibleConfigs

### cd ansibleConfigs

### 3. Get the hosts (inventory) and iis.yml (playbook) files from the git repo using the following command:

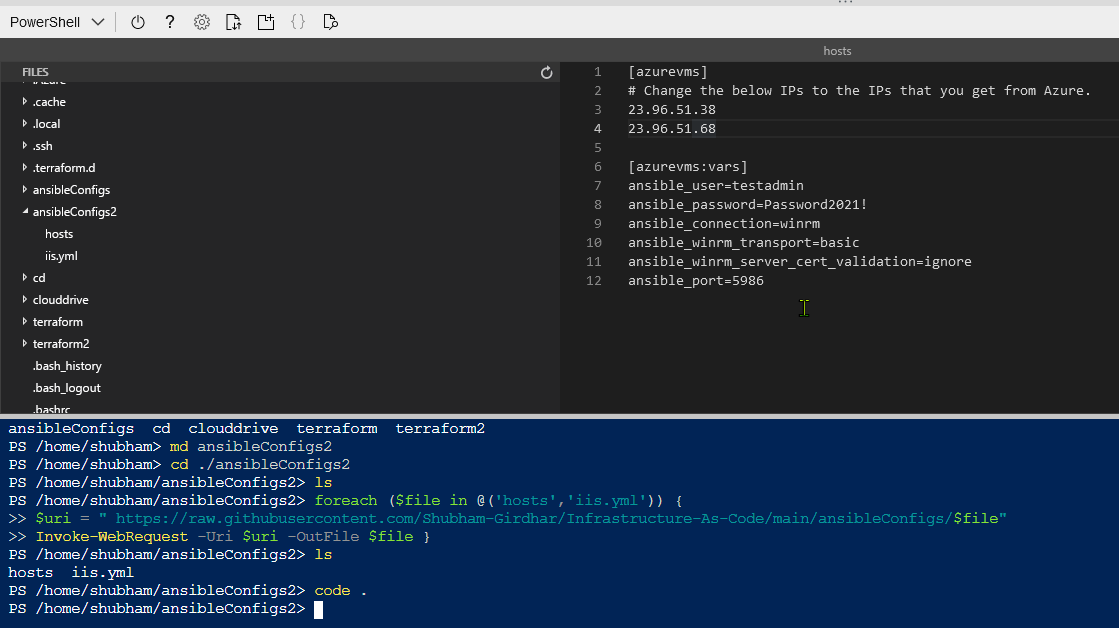
### foreach ($file in @('hosts','iis.yml')) {

### $uri = " https://raw.githubusercontent.com/Shubham-Girdhar/Infrastructure-As-Code/main/ansibleConfigs/$file"

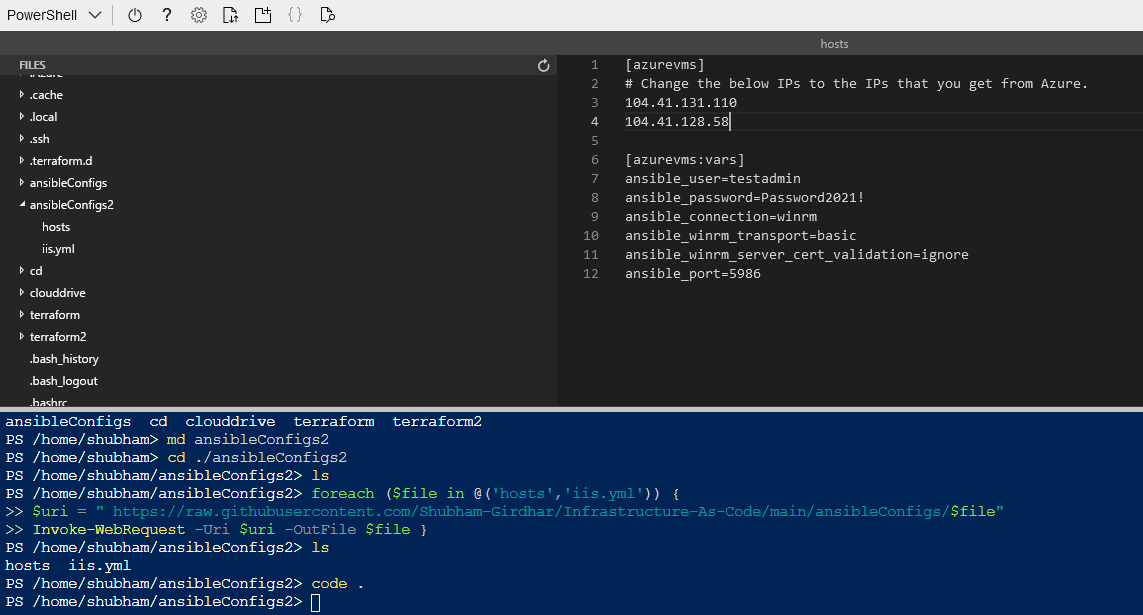
### Invoke-WebRequest -Uri $uri -OutFile $file }

### 4. Now, we update the hosts inventory and add the IP address of the VMs created using Terraform. This is done so that the inventory points the playbook to the correct VMs.

### 5. To Edit the inventory, type code . This will open VS Code editor. Now update the IP addresses under the [azurevms] node and add the IPs of the two VMs we got as the output of the terraform apply step.



6. Use Ctrl + s to save and Ctrl + q to exit the editor.



7. Finally, it’s time to execute the Ansible playbook. This is done using the *ansible-playbook* command which takes the following two parameters:

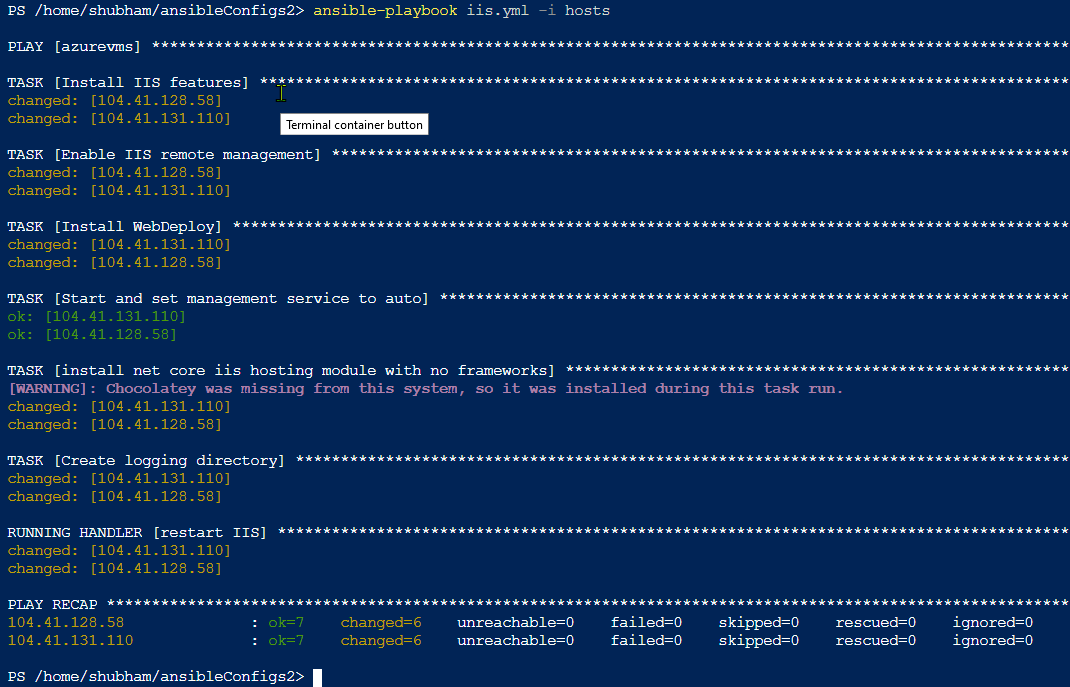
a) name of the playbook (iis.yml)

b) - i name of host (- i hosts)

8. So, the final command looks like:

ansible playbook iis.yml - i hosts

9. This should configure the VMs by installing IIS features, enabling IIS remote management, installing WebDeploy, starting and setting management service to auto, installing .net core iis hosting module and creating a logging directory.



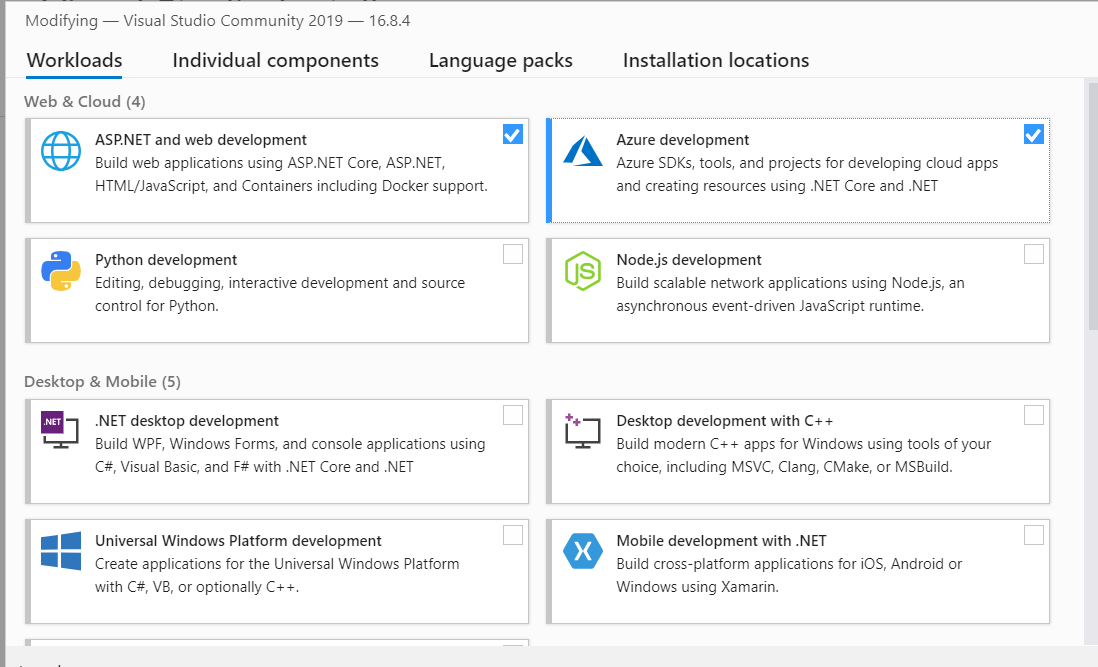
### **5. Deploying the application using Visual Studio 2019**

At this point, we will have the Infrastructure built and configured to support deploying the application. Now it’s time to build the web application and get it deployed onto the VMs.

1. Download Visual-Studio Community Edition 2019.

2. Login to the Visual Studio using university email id.

3. Download the workloads ASP.NET and web development, Azure development, .Net Core cross-platform development.

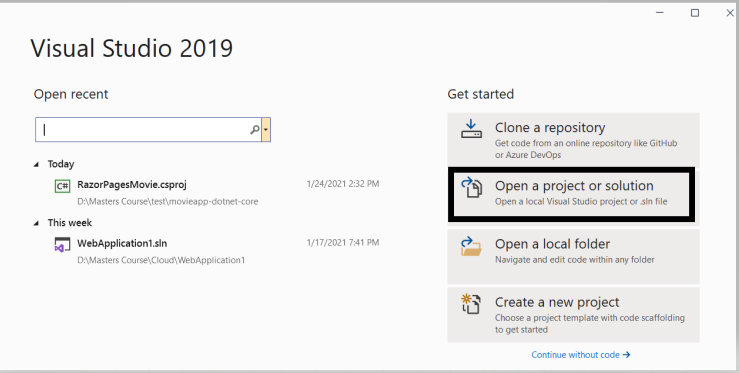


4. Once the required workloads are downloaded, we are ready to deploy an application to the cloud.

5. Clone a sample application in a folder on the local machine. Below is the command to use the Git command-line.

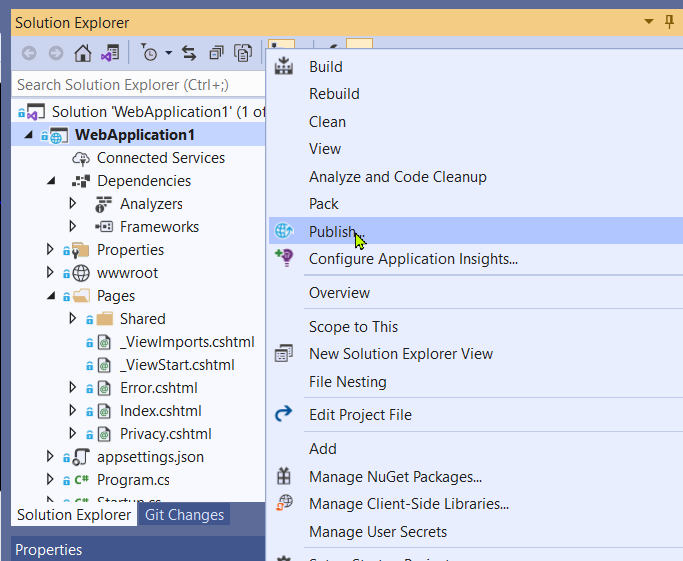
git clone <https://github.com/Shubham-Girdhar/Infrastructure-As-Code.git>

6. Open Visual Studio and click on “Open a project or solution”.

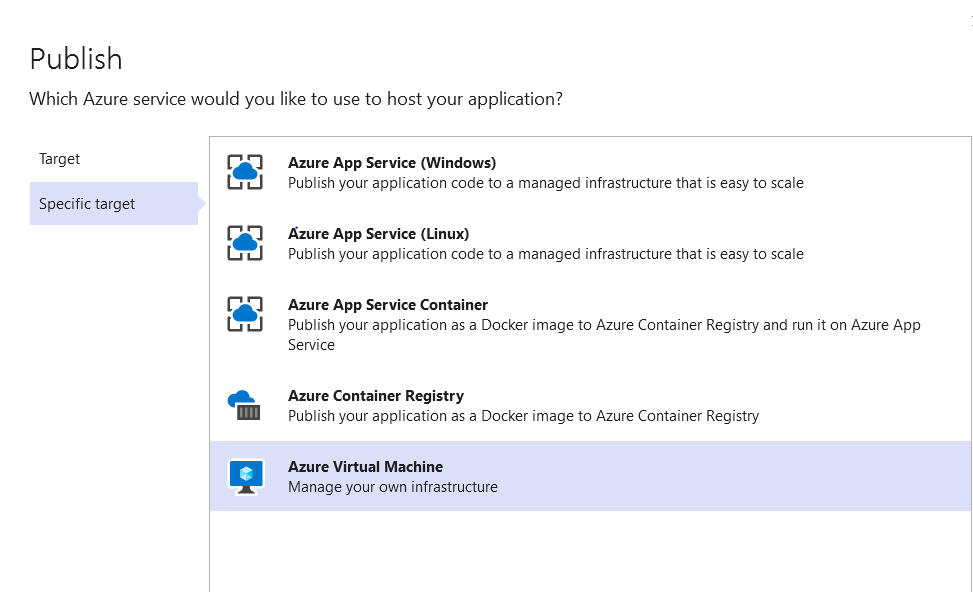


7. Open the folder where the application is cloned. Open the web application1 folder and select .csproj file or .sh file (These files specify a C# project file that contains a list of files that are included in the project to make the application work, along with the references to system assemblies).

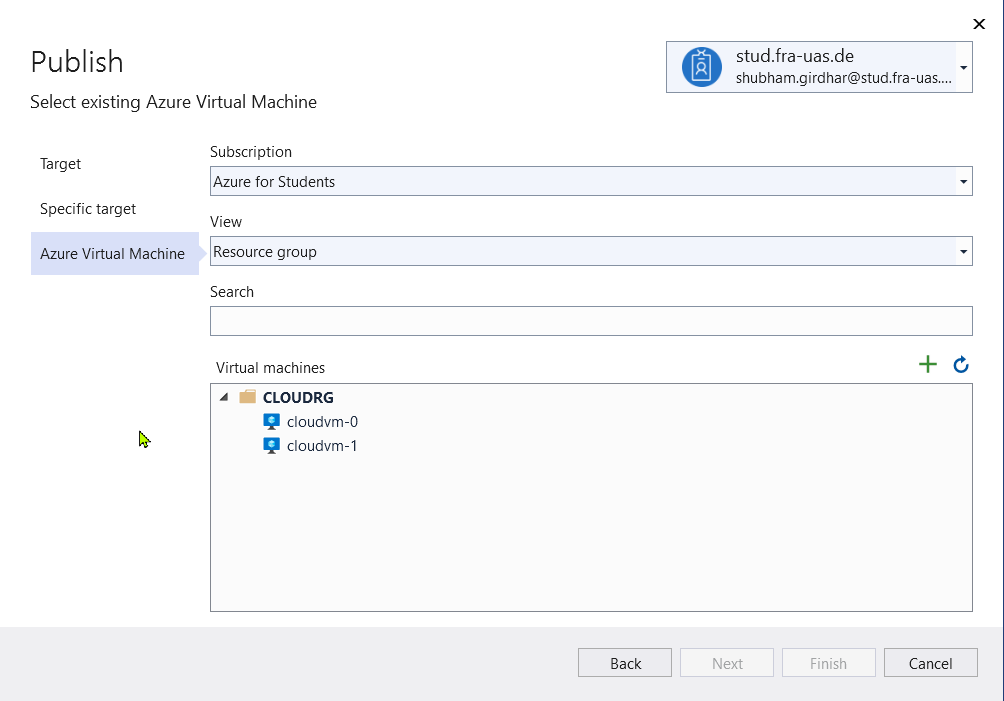
Once the project is opened in Visual Studio, right-click on the project and select **Publish** option.



8. This will open the publish window. Click on Azure and then on Specific target select **Azure Virtual Machine.**



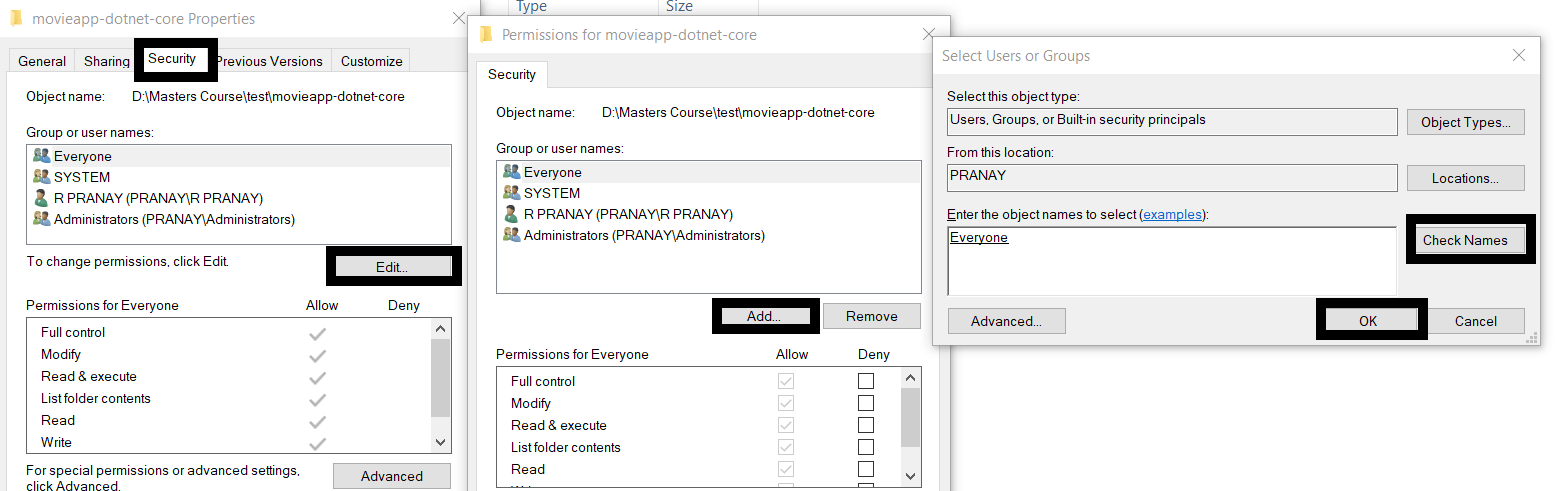
9. Once we select Azure Virtual Machine this will automatically fetch the details from our Azure cloud such as subscription type, resource group and also the available VMs.



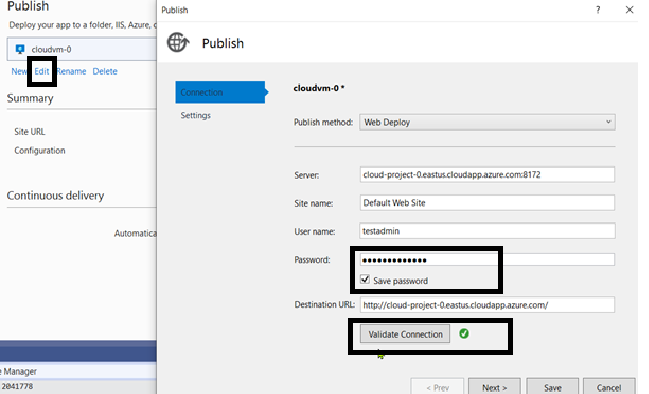
10. Select the first VM, we should see a progress circle below the window indicating that it is testing the connection. If everything goes well, there will be no error messages. If so, click on Finish to add the VM.

Note: As we have cloned a project from GitHub and trying to deploy it we might face an error in the above step. The error says “Web deploy cannot be succeeded”. This might be because of permission issues. This error occurs if we have used different accounts for Visual Studio and the local machine. For example, if the local machine is logged in with the account xyz@gmail.com and Visual Studio is logged in with xyz@stud.fra-uas.de then this error might occur. In order to overcome this error, we need to enable permissions for that particular folder in the local machine.

a) Select the folder which contains the application and right-click on it. Click on properties. Move to the security tab. Click on the Edit option, this will open the permissions of that folder. Click on Add, under Enter the object names to select type “Everyone” and select Check Names. Then click on OK and Apply. By performing the above process, we will enable the permissions to everyone and the error will be fixed.

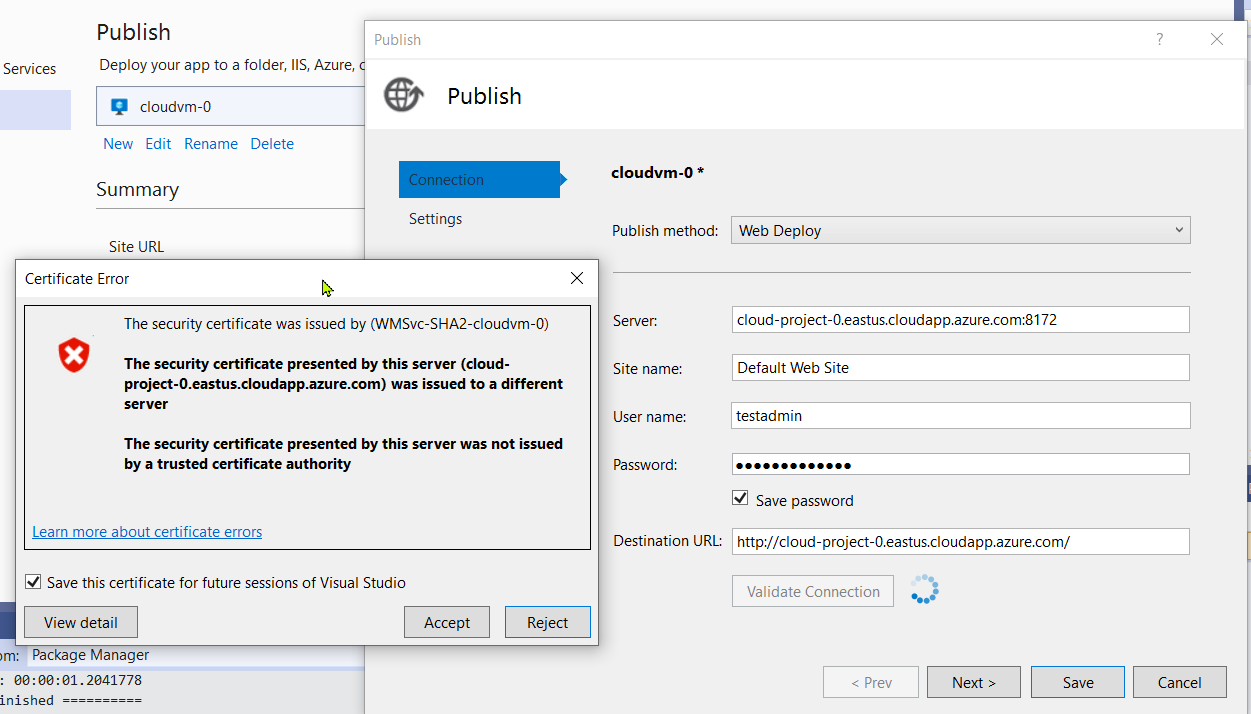


11. Now, the next step is to provide the credentials to connect to the VM and also accept the self-signed certificate on it. To do that, click on Edit as shown below.



12. Provide the Password and click on Save Password so that there is no need to provide the password every time we perform a deployment. When complete, click on Validate Connection. The main reason for validating the connection is to permanently install the self-signed certificate on the VM. If not, we won’t be able to connect.

13. When we click on Validate Connection button, we will receive an error about not trusting a certificate. This is an expected event. Check the Save this certificate for future sessions of Visual Studio checkbox and click on Accept.

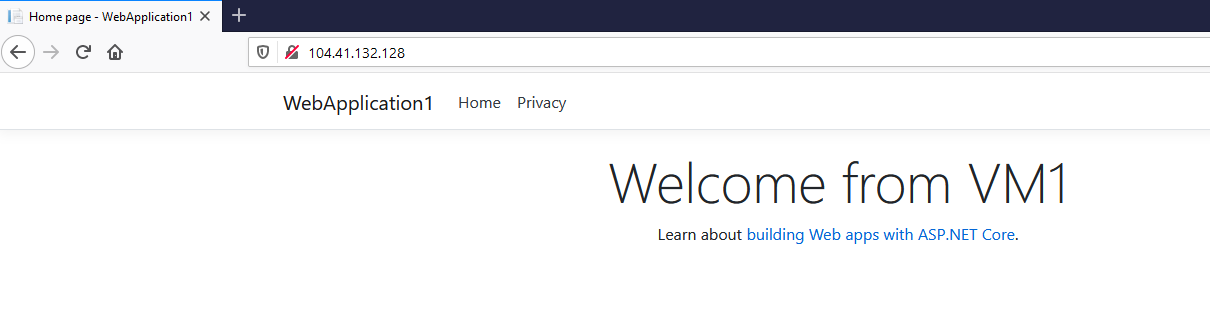


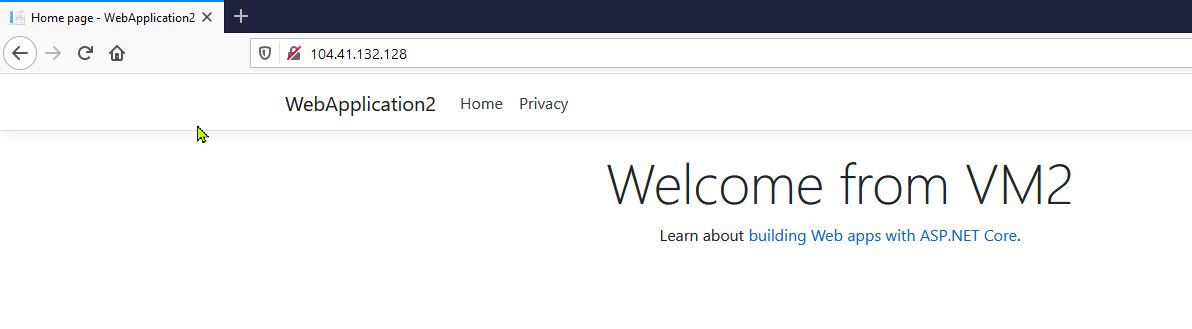
14. Now repeat the steps 6-13 for the second VM. Select the web application2 folder and run it on VM 2. Both the VMs have different web application deployed i.e., the structure of the application is same but the welcome message is different in each of the VMs. This is to know that which VM is picked by the load balancer in case of traffic.

15. Choose each VM and click on the Publish button. This action will tell Visual Studio to use WebDeploy to transfer all the files from local folders to the IIS site on each VM.

16. Once the Publish button is clicked, a web browser will be opened with the deployed application.

17. Below are the images of the deployed web application in first and second VM respectively. For the same load balancer IP, we can see that same application is running in different VMs.





**Conclusion and final remarks**

Infrastructure as Code tries to solve three different needs, which are:

1. Create/destroy infrastructure resources like network resources, virtual machines, load balancers etc.
2. Manage configurations for the created resources.
3. Deploy applications to the infrastructure.

After successful implementation of this project, we were able to achieve the above-mentioned points in the following manner:

1. Automating the infrastructure provisioning process through a reusable Terraform script.
2. Automating the configuration of the VMs using an Ansible playbook.
3. Publishing a .NET CORE web application to configured machines directly through Visual Studio using Web Deploy.

The next steps from here could be automating this process further by configuring these steps into a CI/CD (Continuous Integration/Continuous Deployment) pipeline. This would even automate the process of manually executing the Terraform and Ansible scripts and replace it with just triggering build and release pipelines. While the process of configuring these steps into pipelines could be a little tricky, once achieved, this could make the life of the DevOps or cloud engineer much easier. This would mean that new environments could now be spun in a matter of minutes and that too without the worry of infrastructure deviations/drifts.

**References:**

[1] [https://docs.ansible.com/ansible/latest/collections/ansible/windows](https://docs.ansible.com/ansible/latest/collections/ansible/windows/)

[2] <https://cloudskills.io/blog/configure-ansible-windows>

[3] <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/?view=aspnetcore-3.1#install-the-net-core-hosting-bundle>

[4] <https://learn.hashicorp.com/collections/terraform/azure-get-started>

[5] <https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde>

[6] <https://azuredevopslabs.com/labs/vstsextend/terraform/>

[7] <https://azuredevopslabs.com/labs/vstsextend/ansible/>

[8] <https://dotnet.microsoft.com/learn/aspnet/hello-world-tutorial/intro>