Full Stack Azure Development

#### Preamble

My approach this semester is to not simply do three separate projects, but three separate Epics all related to one goal: create a DevOps style full stack development. I had started to think about how all the tools around todays IT work together, Git source control, containers, CI/CD tools – and of course, The Cloud (pause for dramatic music).

So, my idea was to bring all these fancy tools & toys together into a large eco-system and, in turn, really turn up the gas on Azure to see what it can do.

Essentially, over the course of three projects (Epics), I am going to craft an eco-system of these tools, to include (but may not be all):

* Azure
* Kubernetes
* Docker
* Jenkins
* Ansible

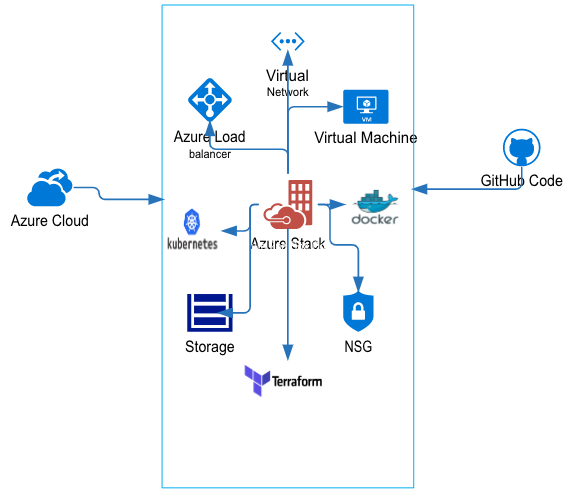
The goal of this endeavor is to end up with a web-application crafted in the cloud, for the cloud and by..well, me.

To start with, in project #1, we will see what it takes to spin up a VM – but not your basic run of the mill VM, but one powered up for using Containers & the latest CI/CD tools. We will combine the use of Azure CLI, Azure Portal and Terraform to create our VM and all the underlying goodies to make it tick.

Then, we are going show off some Docker love buy deploying a simple bulletin board app to the web. Following that, we will take the shiney Docker images and deploy them using Kubernetes & a suite of Kubernetes process. Well, that was the idea at least – read on to see how this turned out!

#### Infrastructure Layout

Life as one big happy family – or Stack. This diagram shows our high level components, along with Git support, from the Azure Cloud



#### First Epic

1. Containerization, Orchestration & Initial Web Application
   1. Deploy VM to Azure with Terraform to create workspace
      1. Services:
         1. Azure VM
         2. Azure Key Vault
         3. Azure Virtual Network – through Marketplace
         4. Azure Virtual Network Interface
         5. Azure Security Group, firewall rules
         6. Azure IAM: user accounts & roles
         7. Azure Service Principal
         8. Azure Resource Groups
         9. Azure Cost Management & Billing: Budget Alerts
         10. Azure Monitoring: VM metrics/alerts
   2. Configure backend services
      1. Create Virtual Network with Marketplace via Portal UI
      2. Create Network Interface via Azure CLI (Local machine)
      3. Create security groups via Portal UI
      4. Create Resource Groups via Portal UI
      5. Create Service Provider via Azure CLI (Local machine)
      6. Provision Users with IAM via Portal UI
   3. Configure workspace with Docker & Kubernetes
      1. Create Dockerfile and build image for containers in the workspace VM
      2. Create Kubernetes spec for creating deployment of application
   4. Application deployment
      1. Serve as Proof of Concept
      2. Deploy Bulletin Board application via Kubernetes
      3. Demonstrate interaction of Docker & Kubernetes pods
      4. Demonstrate interaction with application via browser
   5. Configure supporting services via Portal UI
      1. Setup billing alarm for usage
      2. Setup Auto shutdown (failsafe to remember to shut things down!)
      3. Setup metric alarms for VM usage: Storage, CPU and/or others
   6. Testing the stack
      1. Bring pods up and down
      2. Test metric alarms for proper response (email/text)

#### Sprint Plan

1. Configure Backend Service
   1. Objectives:
      1. Create Backlog Document
      2. Define deliverables & configuration details; add to backlog
      3. Setup all backend Services
      4. Document steps to create services
      5. Create/deploy VM with Terraform specification
      6. Update Backlog; burn-down list if necessary
2. Configure workspace with Docker & Kubernetes
   1. Objectives:
      1. Define deliverables & configuration details; update backlog
      2. Create Docker & Kubernetes specifications
      3. Upload application
      4. Run specifications (iterative with sprint 3
      5. Troubleshoot and resolve issues
      6. Document activity steps
      7. Update backlog; burn-down list if necessary
3. Application Deployment:
   1. Objectives
      1. Check burn-down list for remaining deployment actions on Docker/Kubernetes; Resolve outstanding deliverables
      2. Complete application deployment (iterative with Sprint 2)
      3. Test application usage
      4. Document activity steps
4. Configure supporting services & Test
   1. Objectives
      1. Complete any burn-down items from previous sprints
      2. Create supporting services
      3. Test Kubernetes deploy/redeploy
      4. Ensure application stability after redeploying
      5. Test any Alarms
      6. Document activity steps
      7. Complete documentation & submit Sat, 13 June before midnight

#### Process Documentation

# Sprint #1

|  |  |  |
| --- | --- | --- |
| **Activity** | **Specification** | **Status** |
| Create Resource Group | Name: UMLfullStackrg |  |
| Create User | Add user: developer |  |
| Install Azure CLI |  |  |
| Update User roles |  |  |
| Create Key Vault |  |  |
| Create Service Principal |  |  |
| Upload PEM to Key Vault |  |  |
| Test SP login |  |  |
| Create Net Security Group | Allow for traffic against the VM, expose only needed ports |  |
| Create virtual network | Name: umlnet |  |
| Create NICs |  |  |
| Create & Deploy Terraform |  |  |
| Test VM functionality & Access |  |  |

# Sprint #2

|  |  |  |
| --- | --- | --- |
| **Activity** | **Specification** | **Status** |
| Upload application code | Via Git |  |
| Create Dockerfile |  |  |
| Build Docker image |  |  |
| Configure kubectl |  |  |
| Create Kubernetes spec |  |  |
| Bring up pods with docker images |  |  |
| Troubleshoot, iterate if necessary – document issues! |  |  |

# Sprint #3

|  |  |  |
| --- | --- | --- |
| **Activity** | **Specification** | **Status** |
| Complete any burn down tasks for deployments |  |  |
| Complete any deployment actions |  |  |
| Test App |  |  |

# Sprint #4

|  |  |  |
| --- | --- | --- |
| **Activity** | **Specification** | **Status** |
| Setup billing alerts |  |  |
| Setup metric alerts |  |  |
| Test stability of Kubernetes redeployment |  |  |
| Test alarms |  |  |
| Ensure all backlog tasks are complete |  |  |
| Complete project documentation & submit |  |  |

**PROJECT #1: Node.js with Docker/Kubernetes**

**Sprint #1 Notes:**

* + - Create & deploy Terraform for VM
    - Test login to VM
    - Create Network Interface via Azure CLI (Local machine)
    - Create Virtual Network with Marketplace via Portal UI
    - Create security groups via Portal UI
    - Provision Users with IAM via Portal UI
    - Create Key Vault
    - Create Service Principal via Azure CLI (Local machine)
    - Create Resource Groups via Portal UI

**Create a Resource Group (RG) for the project**

Log into the Azure Portal

Use search bar to navigate to the following: Resource Groups

Click "+ Add"

Create the resource with the following settings:

Subscription: Select appropriate subscription for the project

Resource group: UMLfullStackrg

Region: (US) East US

Next: Tags

Name: UMLProject

Value: FullStack

Next: Review + create > Create

Note: At this point, it takes a minute or two for the Resource Group to appear in the list, be patient

**Update Roles for authorized users**

In this step, we will add a role for a user to be able to access the project Resource Group. In this case, I will add Prof. Bob so that he can review the project and operate the resources defined in the UMLfullStackrg. Without proper roles, even though a user is (and must be!) defined in the subscription, they have no cross resource rights without provisioning proper roles. That is, users without explicit Roles on your resources cannot see or interact with any of the resources in your Resource Group.

Use the search bar to navigate to: Resource Groups

Click on the Resource Group: UMLfullStackrg

From the details pane, select Access Control (IAM)

In the next pane, click "+ Add" > Add role assignment

From the Add role assignment pane:

Select Contributor from the Role

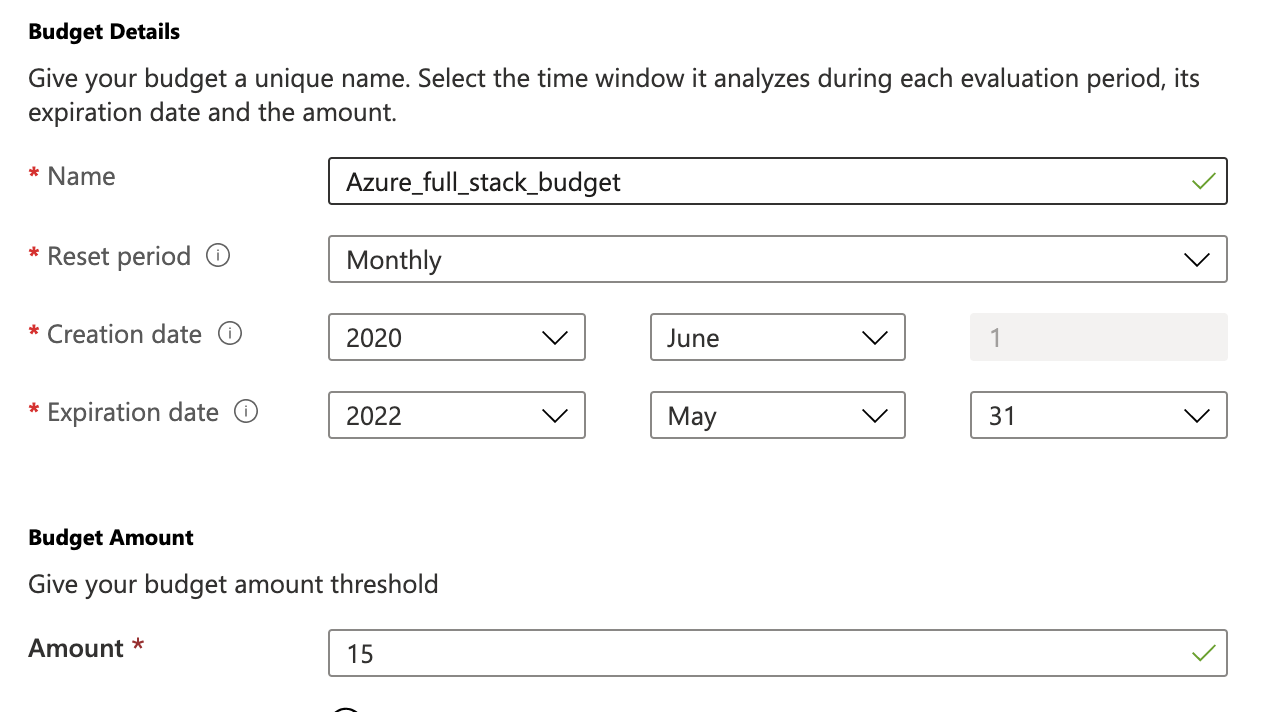
Assign access to Bell1, Robert J uml.edu

**Create Billing Alarm**

Now that we have an active resource group, it is a good time to setup an alarm to let us know if our spend is getting too high.

From the upper left next to the Microsoft Azure logo, click the vent menu and select Cost Management and from the left pane, select Budgets and click "+ Add"

The Cost Calculater for Azure suggests that my spend for the resources needed for the project should be around $5/mo. (allowing for about 10-15 hours of "UP" time), so I will set budget for 3x that ($15) and an alert threshold of 40% (



After clicking next, I can enter my 40% threshold and my email to receive the alert, then click Create.

Now I will receive an email if my spend goes over $6 in the month.

**Create a Key Vault**

Before creating the SP, we need a place to store our security items. In Azure, this is called a Key Vault, we will use the Key Vault to store a set of credentials for use in creating the SP. Using the Key Vault means that users do not need to store security items locally - like RSA tokens, passwords, etc.

Use the search bar to navigate to: Key Vault and click "+ Add"

Use the following details to create the Key Vault

Subscription: Choose appropriate sub

Resource Group: choose the project RG, UMLfullStackrg

Key Vault Name: UMLfullStackKV

Region: East US (keep in the same region as the rest of your resources)

Pricing Tier: Standard

Soft Delete: Enable

Retention Period: 90

Purge Protection: Disable

Next: Access policy

Select all the checkboxes: VMs, Resource Mgr, Disk Encryption

Next: Networking

Select Public enpoint(all networks)

Next: Tags

Name: UMLProject

Value: FullStack

Next: Review & create

Click Create

**Installation of Azure CL**I

I am using a Mac, so this is fairly simple; execute the following in a terminal:

$ brew update && brew install azure-cli

Login into azure, note if you have multiple subscriptions, you will need to execute additional steps before logging in (mainly, setting the right subscription!).

Details for dealing with multiple subs can be found at:

<https://docs.microsoft.com/en-us/cli/azure/manage-azure-subscriptions-azure-cli?view=azure-cli-latest#:~:text=Use%20multiple%20Azure%20subscriptions&text=However%2C%20if%20you%20are%20part,both%20globally%20and%20per%20command.>

**Create Service Principal (SP)**

Service principals are used to securely access resource groups from outside the Portal. The following steps requite use of the Azure CLI service.

**Execute "az" commands to create the SP**

$ az login -- This will output details for the sub, review to make sure you are in the right place!

**Show list of SP's**

$ az ad sp list --show-mine (This will likely return "[ ]" if you are new to this)

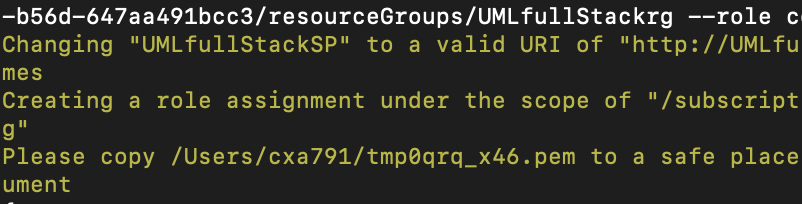
We need to create an RSA key pair to store in the Key Vault, fortunately, Azure automates this process along with our SP create

**Create SP**

The SP scope is set with your sub ID and path to your resourceGroup.

$ az ad sp create-for-rbac -n "UMLfullStackSP" --scopes /subscriptions/57db1d00-01f8-4538-b56d-647aa491bcc3/resourceGroups/UMLfullStackrg --role contributor --create-cert

The output will tell you where the certs are, but make sure you keep them OUT of any source control (git). We are going to upload them to Key Vault for safe keeping, as well.



Also, copy the output of metadata about the SP for use later (eg. logging in)

{

"appId": "8a74c7c3-e69c-4be6-8ce1-d7084317eab9",

"displayName": "UMLfullStackSP",

"fileWithCertAndPrivateKey": "/Users/cxa791/tmp0qrq\_x46.pem",

"name": "http://UMLfullStackSP",

"password": null,

"tenant": "4c25b8a6-17f7-46f9-83f0-54734ab81fb1"

}

**Upload to Key Vault**

Navigate to Home > Key Vaults > UMLfullStackKV and click "+ Generate/Import"

Select Import from drop down and use the file dialog to find and select your key (as shown in last step)

Give a name for the cert, like: UMLfullStackSPcert

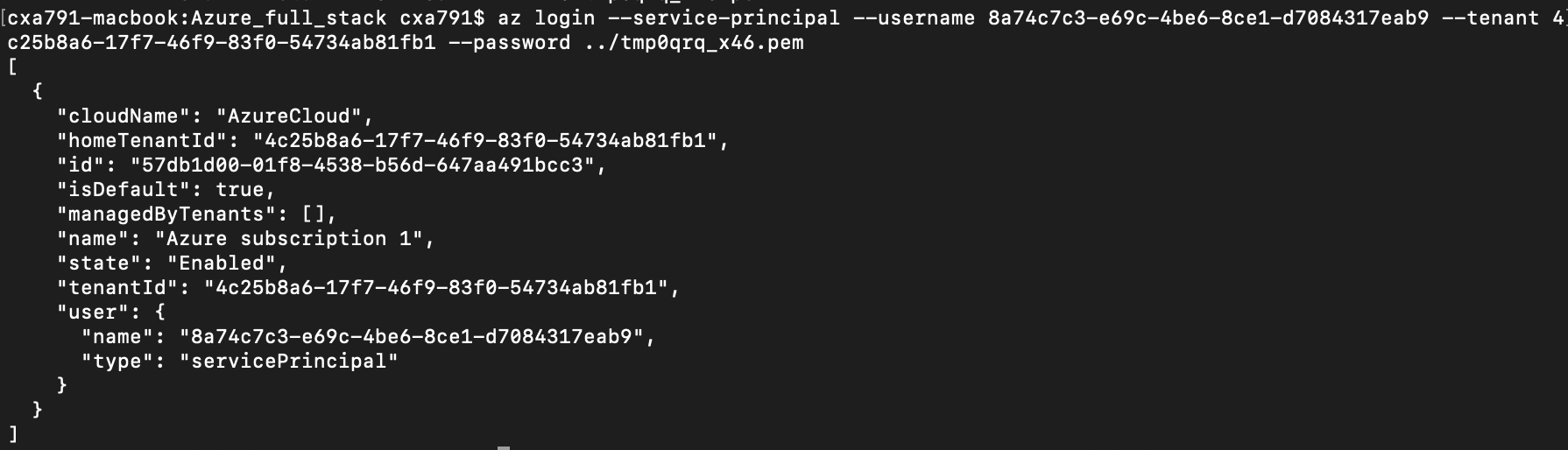
Set a password and click "Create" - MyUMLCert

Now we have a cert safely uploaded to the Cloud that can be shared with authorized personnel

**Test SP Login**

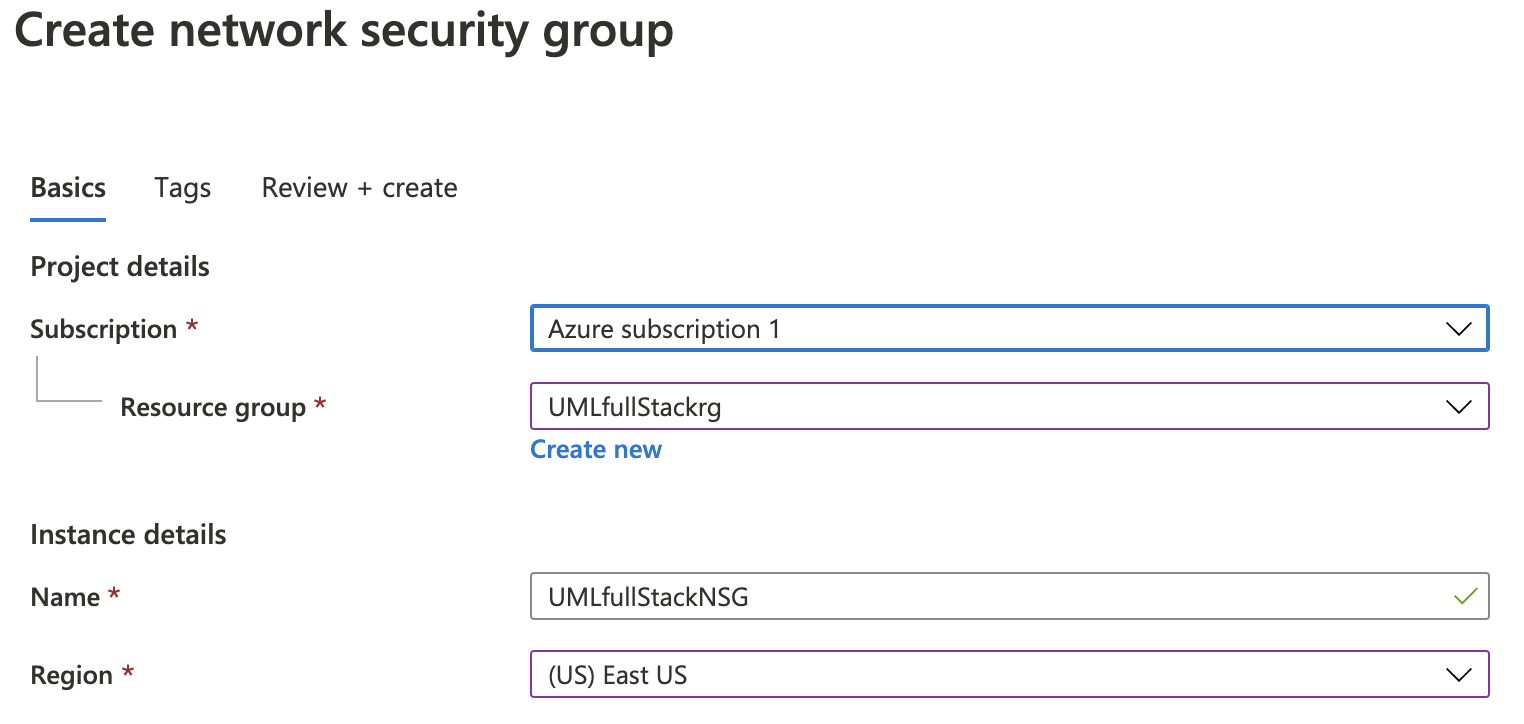
Use the following syntax to login using the SP

$ az login --service-principal --username APP\_ID --tenant TENANT\_ID --password /path/to/cert



**Create Network Security Group (NSG) via Azure Portal**

Use search for Network Security Groups and click "+ Add"



Don't forget to use Tags for easy identification:

Name: UMLProject

Value: FullStack

Click "Review + create” and Create

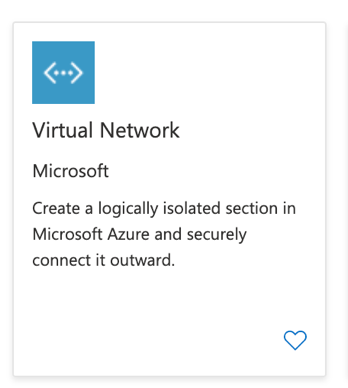
Once again, it may take a few moments to see the new NSG under the listings

We will be doing more configuration with NSG's in the future to support our VM's and Applications.

**Create Virtual Network through Marketplace using the Portal**

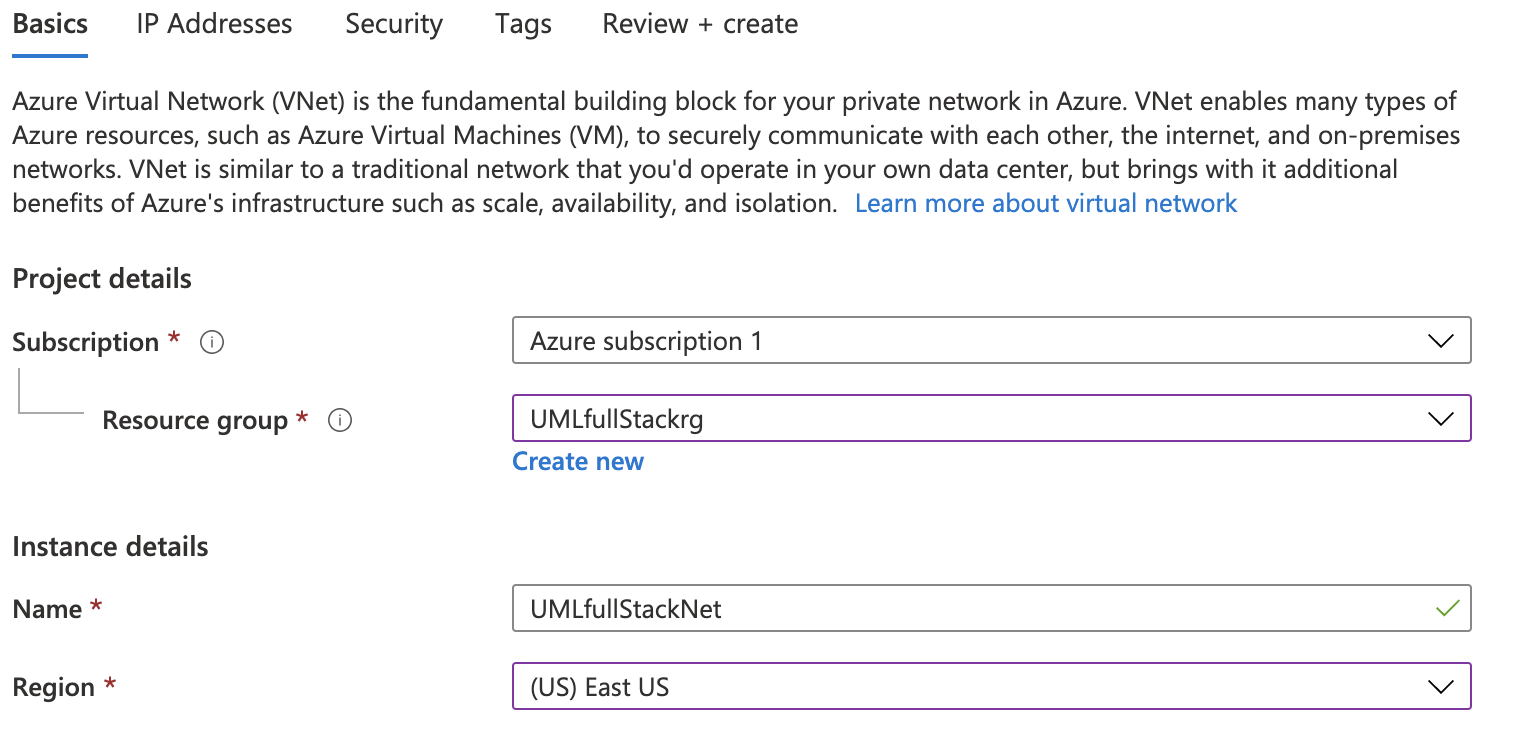
Use search Marketplace and then in the Marketplace search, type "Virtual Network"

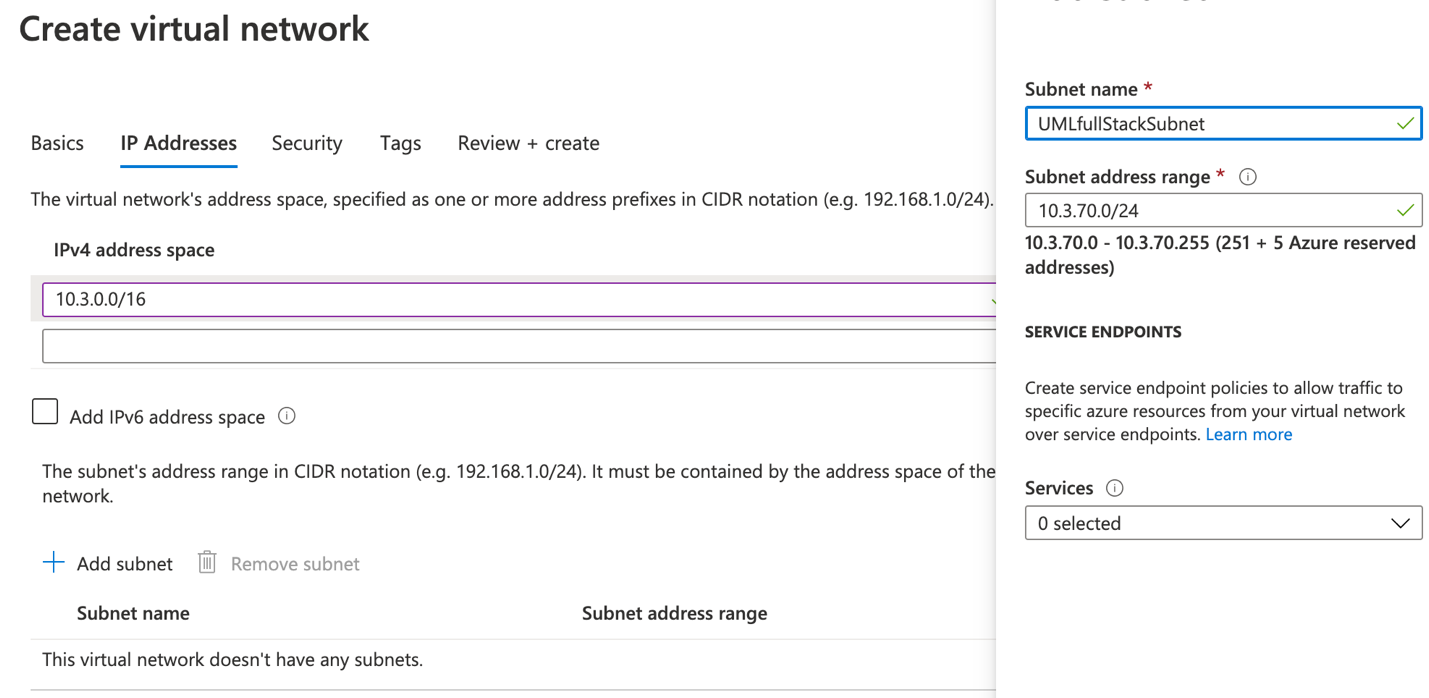
Select Virtual Network

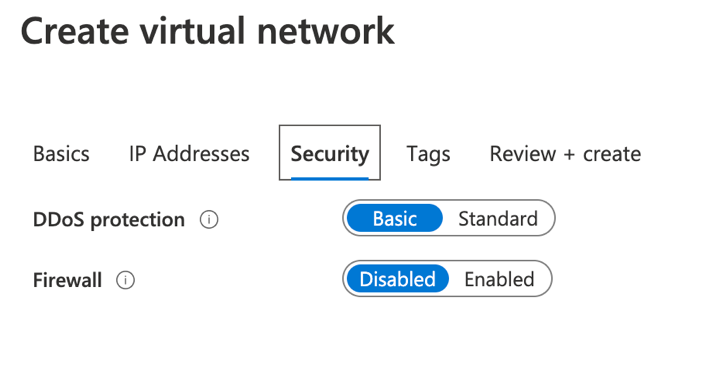


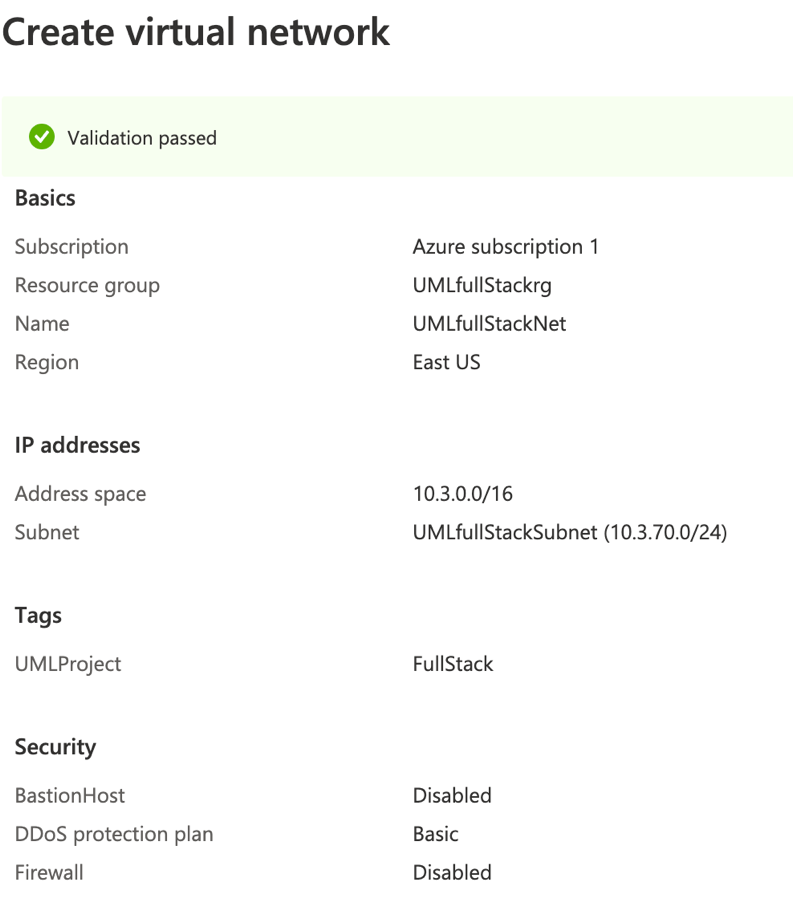
click "Create"

follow the images below to create the Virtual Network









The new Virtual Network will show on Home > Virtual Networks.

Now we must "associate" the NSG with the subnet to guard our network traffic lanes.

Use the Azure search and find Network Security Groups, select the NSG we created (UMLfullStackNSG) and select Subnets from the left-most pane

At the top of the details pane for our subnets, click "+ Associate" and select the Virtual Network Group we created in previous steps (UMLfullStackNet) as well as the proper subnet (UMLfullStackSubnet).

click Ok

**Create Network Interface (NIC) with Azure CL**I

We are going to shift gears from the Portal to again use the Azure CLI. This time, we will create a network interface for the VM we will create in the next section.

Login via the SP:

$ az login --service-principal --username 8a74c7c3-e69c-4be6-8ce1-d7084317eab9 --tenant 4c25b8a6-17f7-46f9-83f0-54734ab81fb1 --password ../tmp0qrq\_x46.pem

**Create Public IP**

**For the VM**

az network public-ip create -n UMLfullStackPIP \

-g UMLfullStackrg \

--allocation-method Static \

--l eastus \

--tags UMLProject=FullStack

**For the Load Balancer**

az network public-ip create -n umlfullstacklb \

-g UMLfullStackrg \

--allocation-method Static \

--l eastus \

--tags UMLProject=FullStack

**Create NIC:**

az network nic create -n UMLfullStackNIC \

-g UMLfullStackrg \

--subnet UMLfullStackSubnet \

--vnet-name UMLfullStackNet \

--network-security-group UMLfullStackNSG \

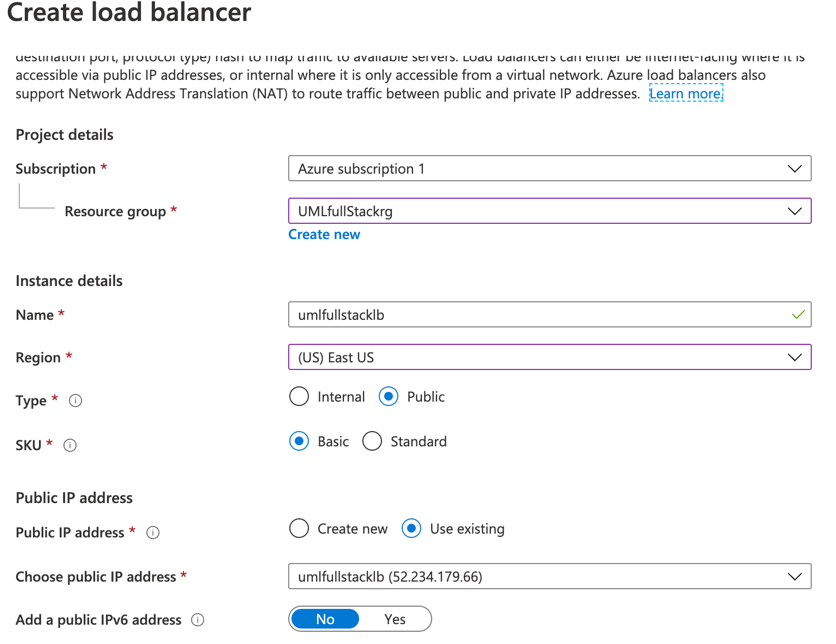
--private-ip-address 10.3.70.10 \

--public-ip-address UMLfullStackPIP \

--tags UMLProject=FullStack

Now our NSG, Virtual Network, Subnets & NIC are all "plumbed together". We will do even more configuration in later tasks.

**Create Load Balancer for Kubernetes pod access**



IP address: 52.234.179.66 (umlfullstacklb)

**Configure & Deploy VM via CLI and Kubernetes**

**Install Terraform**

We will be creating a few files for our Terraform deployment, but first, install the Terraform CLI for your O/S.

Mac using brew: brew install terraform

Check version:

$ terraform versio

**Source Control Setup**

Create a file for the project: Azure\_full\_stack

  ! Important: BEFORE you setup version control (SCM) - Certs, PEMs, RSA key & other secrets are NOT in SCM.

We will be using git for this project, so I will create a file named .gitignore (dot is required!)

$ touch .gitignore

$ echo .terraform >> .gitignore

$ echo secrets.tfvars >> .gitignore

$ echo \*.pem >> .gitignore

Add any other items you don't want in source control

**Initialize the repository**

Inside the Azure\_full\_stack file directory, execute the following:

$ git init

$ git add -A

$ git commit -am 'First commit'

**Create Terraform files**

As we want to keep our project portable, we will use a modular approach to creating our Terraform files, starting with vars.tf, which is a definition file. This allows us to instantiate some variables for later use.

**Create vars file for definitions**

$ touch vars.tf

Add the following lines:

# var.tf

# Variable definitions

# general

# Client & Subscription Data

variable "azure\_sub\_id" {

default = ""

}

variable "azure\_client\_id" {

default = ""

}

variable "azure\_client\_secret" {

default = ""

}

variable "azure\_tenant\_id" {

default = ""

}

variable "prefix" {

default = ""

}

variable "vm\_admin\_name" {

default = ""

}

We will also create another vars files with some common definitions that will automatically load when we run a deployment

$ touch FullStack.auto.tfvars

Add the following lines:

# FullStack.auto.tfvars

# Variable overrides for the sandbox workspace

# general

vm\_admin\_name = "fullstack"

prefix = "UMLfullStack"

**Next file will be our secrets file (never committed to git).**

Before creating the file, I find it useful to alias our PEM cert created with the SP for easy identification.

I copy the PEM file to another file in this top directory called azureauth.pem

$ touch secrets.tfvars

There is an example file in the Git repo for this project.

We need add the following lines to our file:

# Variable overrides for the workspace (SECRETS! DO NOT COMMIT TO SOURCE CONTROL)

# See var.tf for more information

# general

# Get sub information by executing: az account show

# The subscription id for the sub you need to access

azure\_sub\_id = "xxxxx-xxxx-xxxxx-xxxxx-xxxxx"

# The homeTenantId from that subscription

azure\_tenant\_id = "xxxx-xxxx-xxxx-xxxx-xxxxx"

# As we are using a service principal, we need the following additional vars

# Get SP information using: az ad sp list --show-mine

# The appId for the service principal

azure\_client\_id = "xxxxx-xxxx-xxxxx-xxxxx-xxxxx"

# The name (and location, if not in same dir) of the PEM file you use to login with SP

azure\_client\_secret = "xxxxxxxxxx"

**Now, create the providers.tf**

This file defines a few providers that we use for Terraform. Providers are method definitions specific to the platform or module they reference. Keeping the providers.tf separate means we could define a different provider if we wanted to, and still run the same main.tf file (eg. we could have a providers file for Azure, then change it for AWS)

In the Azure\_full\_stack directory:

$ touch providers.tf

Add the following (for use with Azure):

providers.tf

# Resource definitions for providers

# Configure the Microsoft Azure Resource Manager provider

provider "azurerm" {

version = ">= 2.0.0"

features {}

subscription\_id = var.azure\_sub\_id

tenant\_id = var.azure\_tenant\_id

# Allows to access the subscription resource group through a service principal

#

client\_id = var.azure\_client\_id

client\_secret = var.azure\_client\_secret

skip\_provider\_registration = true

}

# Configure the Microsoft Azure Active Directory provider

provider "azuread" {

version = "= 0.4.0"

subscription\_id = var.azure\_sub\_id

tenant\_id = var.azure\_tenant\_id

# Allows to access the subscription resource group through a service principal

#

client\_id = var.azure\_client\_id

client\_secret = var.azure\_client\_secret

}

# provider "random" {

# version = ">= 2.1"

# }

# provider "null" {

# version = ">= 2.1"

# }

# Retrieve details for the subscription

data "azurerm\_subscription" "subscription" {}

**Create service\_data.tf for getting data on existing services**

# The following data calls will gather information about existing services.

# This is useful in a production (and even development) environments where some services are

# typically provisioned by another group, like networking or security services

#

# The data adatpers here assume the following services have already been provisioned and that the user

# has proper access via owner role or service principal:

# Resource Group

# Networking: Virtual Network, Subnets, NSG & NIC

#

data "azurerm\_resource\_group" "UMLfullStackRG" {

name = "UMLfullStackrg"

}

output "id" {

value = data.azurerm\_resource\_group.UMLfullStackRG.id

}

data "azurerm\_virtual\_network" "UMLfullStackNet" {

name = "UMLfullStackNet"

resource\_group\_name = data.azurerm\_resource\_group.UMLfullStackRG.name

}

output "virtual\_network\_id" {

value = data.azurerm\_virtual\_network.UMLfullStackNet.id

}

data "azurerm\_subnet" "UMLfullStackSubnet" {

name = "UMLfullStackSubnet"

virtual\_network\_name = data.azurerm\_virtual\_network.UMLfullStackNet.name

resource\_group\_name = data.azurerm\_resource\_group.UMLfullStackRG.name

}

output "subnet\_id" {

value = data.azurerm\_subnet.UMLfullStackSubnet.id

}

data "azurerm\_network\_security\_group" "UMLfullStackNSG" {

name = "UMLfullStackNSG"

resource\_group\_name = data.azurerm\_resource\_group.UMLfullStackRG.name

}

output "network\_security\_group\_id" {

value = data.azurerm\_network\_security\_group.UMLfullStackNSG.id

}

data "azurerm\_network\_interface" "UMLfullStackNIC" {

name = "UMLfullStackNIC"

resource\_group\_name = data.azurerm\_resource\_group.UMLfullStackRG.name

}

output "network\_interface\_id" {

value = data.azurerm\_network\_interface.UMLfullStackNIC.id

}

**Create main.tf for creating the VM deployment**

Before continuing on with creating the main.tf file, we need to attend to a housekeeping task - creating the RSA keys for the VM.

In the Azure\_full\_stack directory, execute the following:

$ ssh-keygen -t rsa -b 4096 -C "[*your\_email@example.com*](mailto:your_email@example.com)"

enter ./id\_rsa when prompted to create the key in the Azure\_full\_stack directory

Ensure the private key is NOT in source control:

$ echo id\_rsa\* >> .gitignore

Our Terraform template is going to automatically upload the keys needed to access the vm via rsa key

Create a file called main.tf

Add the following Terraform data:

# Main definitions for our Azure\_full\_stack VM creation

# # For creating some random strings

resource "random\_string" "random" {

length = 6

upper = false

special = false

#override\_special = "/@£$"

}

# # Create public IPs

resource "azurerm\_public\_ip" "UMLfullStackPIP" {

name = "UMLfullStackPID"

location = "eastus"

resource\_group\_name = data.azurerm\_resource\_group.UMLfullStackRG.name

allocation\_method = "Dynamic"

tags = {

UMLProject = "FullStack"

}

}

# Used for storing diagnostic data

resource "azurerm\_storage\_account" "UMLfullStackSA" {

name = "${var.prefix}sa${random\_string.random.result}"

resource\_group\_name = data.azurerm\_resource\_group.UMLfullStackRG.name

location = "eastus"

account\_tier = "Standard"

account\_replication\_type = "LRS"

tags = {

UMLProject = "FullStack"

}

}

# Create virtual machine

resource "azurerm\_virtual\_machine" "UMLfullStackVM" {

name = "umlapp01"

location = "eastus"

resource\_group\_name = data.azurerm\_resource\_group.UMLfullStackRG.name

network\_interface\_ids = [data.azurerm\_network\_interface.UMLfullStackNIC.id]

vm\_size = "Standard\_B1s"

os\_profile {

computer\_name = "umlapp01"

admin\_username = var.vm\_admin\_name

}

os\_profile\_linux\_config {

disable\_password\_authentication = true

ssh\_keys {

key\_data = file("./id\_rsa.pub")

path = "/home/${var.vm\_admin\_name}/.ssh/authorized\_keys"

}

}

storage\_os\_disk {

name = "${var.prefix}-disk01"

caching = "ReadWrite"

create\_option = "FromImage"

managed\_disk\_type = "Premium\_LRS"

disk\_size\_gb = 30

}

storage\_image\_reference {

publisher = "Canonical"

offer = "UbuntuServer"

sku = "18.04-LTS"

version = "latest"

}

boot\_diagnostics {

enabled = true

storage\_uri = azurerm\_storage\_account.UMLfullStackSA.primary\_blob\_endpoint

}

tags = {

UMLProject = "FullStack"

}

}

After creating all the above Terraform files, we need to initialize the Terraform providers and then check our work.

$ terraform init

$ terraform plan

When all looks good,

$ terraform apply

**Test Login to VM & Setup Kubernetes/Docker**

Now we should be able to access our vm via our public IP

$ ssh -i <id\_rsa in Azure\_full\_stack directory> [fullstack@52.188.50.234](mailto:fullstack@52.188.50.234)

You will need to add the code from the Git repo: <https://github.com/dockersamples/node-bulletin-board> to new VM. These files contain the sample application we will deploy in containers on the Azure VM

Keep the Bulletin-board code in a separate directory under the main directory setup:

Example tree in my repo:

|-- Full\ Stack\ Azure\ Development.docx

|-- FullStack.auto.tfvars

|-- azureauth.pem

|-- bulletin-board

| |-- Dockerfile

| |-- LICENSE

| |-- app.js

| |-- backend

| | |-- api.js

| | |-- events.js

| | `-- index.js

| |-- dockerBB.yaml

| |-- fonts

| | `-- geomanist

| | `-- hinted-Geomanist-Book.woff2

| |-- index.html

| |-- package.json

| |-- readme.md

| |-- server.js

| `-- site.css

|-- id\_rsa

|-- id\_rsa.pub

|-- main.tf

|-- providers.tf

|-- secrets.tfvars

|-- secrets.tfvars.example

|-- service\_data.tf

|-- terraform.tfstate

`-- vars.tf

Clone your repo to the VM so you can use the Azure\_full\_stack files.

$ git clone "<your repo https>

**Install Docker**

**$** sudo apt-get update -y

$ sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

gnupg-agent \

software-properties-common

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"

$ sudo apt update

$ apt-cache policy docker-ce

$ sudo apt install docker-ce

$ sudo systemctl status docker

$ sudo usermod -aG docker fullstack

You will need to log out and back in for the group change to take effect

**Install Kubernetes** 52.188.50.234

As this is a development box, we will be using minikube to test out Kubernetes functions. A production setup has steeper compute requirements and needs a few more packages. However, minikube is more than sufficient to kick the tires and do some fun local development work.

We will install & configure from binary

$ curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl

$ chmod +x ./kubectl

$ sudo mv ./kubectl /usr/local/bin/kubectl

And the minikube development setup

$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/

Test its working

$ kubectl version --client

**Start a Kubernetes Cluster**

* Commit & push all the code in Azure\_full\_stack to Git from local machine
* On the VM, clone the repo so we can begin development work

Navigate into the bulletin-board directory and have a look at the Dockerfile & dockerBB.yaml files - these are used to build our images and deploy to pods.

First, build the Dockerfile image:

$ docker build --tag bulletinboard:latest .

**TROUBLESHOOTING #1**

At this time, as I was testing the deployment to document the steps for Kubernetes, I discovered that my "free" VM (Standard\_B1s) was not up to the task. Had I taken a second to review, I would have noticed that this SKU is only a single core - that won't meet the requirements for Kubernetes.

So, now we get to see the magic of two special capabilities.

1. You can upgrade your SKU on the fly in the cloud
2. Terraform is declarative, so we just update the SKU and run terraform apply to make the changes in the cloud

Step one, update the Terraform definition in main.tf under the virtual machine resource and update the vm\_size as follows:

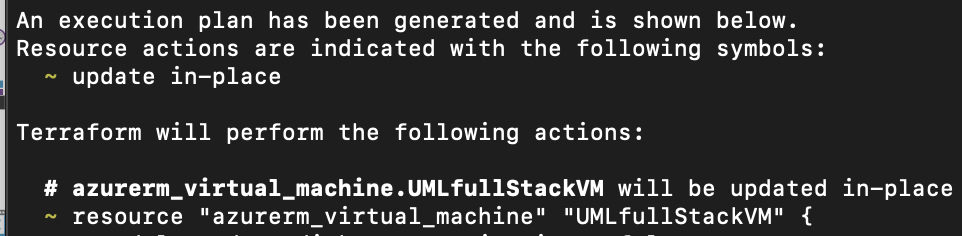
vm\_size = "Standard\_B2s"

This will tell Terraform we "expect" the VM to be a B-Series 2vcpu.

From your local machine, in the Azure\_full\_stack directory, check the plan

$ terraform plan

We can see Terraform has noted our changes and intends to update:



note the following specific lines in the output

~ vm\_size = "Standard\_B1s" -> "Standard\_B2s"

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

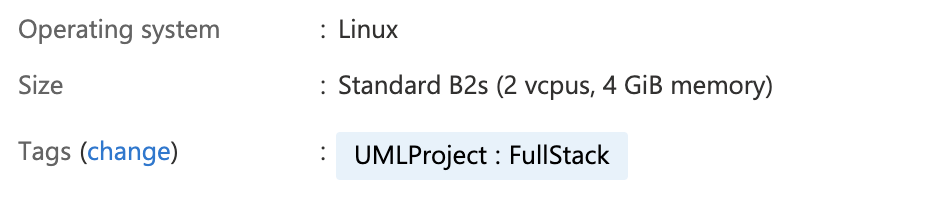
Looks good, let's see what Terraform and Azure do with this definition change.

$ terraform apply

Awesome: **Apply complete! Resources: 0 added, 1 changed, 0 destroyed.**

Now lets double check with Azure to see our VM state.

Excellent:



Ok, crisis avoided, back to our development work!

First, log back in to the VM using our id\_rsa cert and fullstack user

$ ssh -i id\_rsa [fullstack@52.188.50.234](mailto:fullstack@52.188.50.234)

And check for our existing files:

$ ls -lhtr # Our Azure\_full\_stack directory is intact

$ docker image ls # our bulletin board image is still there

Great, Azure upgraded the VM SKU AND kept our files! Everything is going as planned (ya, right...)

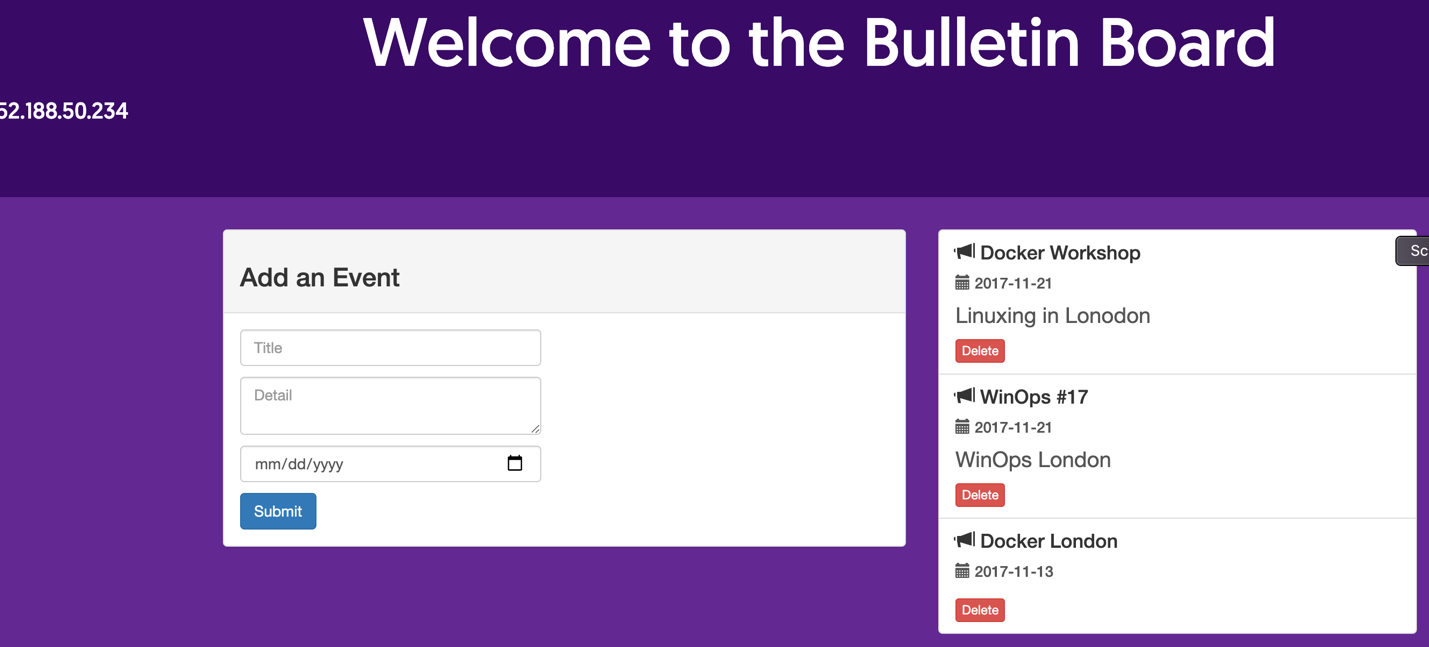
So, lets try to deploy the Docker application without Kubernetes and make sure it is working.

$ cd ~/Azure\_full\_stack

$ docker run --publish 8000:8080 --detach --name bb bulletinboard:latest

From our LOCAL machine, open a browser and goto: 52.188.50.234:8000

Success!



Now, we'lll bring down the Docker container and move right to using Kubernetes

Back on the VM:

$ docker rm --force bb

Navigate to Azure\_full\_stack/bulletin-board

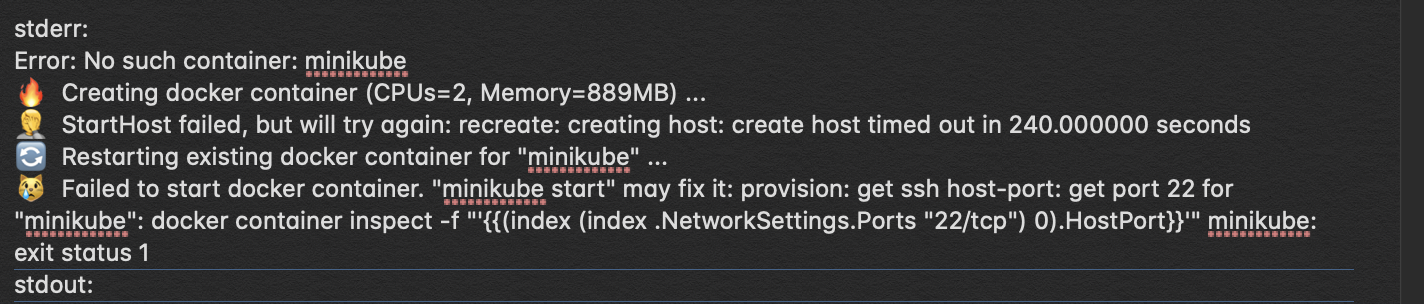
$ minikube start

output as minikube starts: 🔥 Creating docker container (CPUs=2, Memory=889MB) ... Cool, it sees the 2 cpus!

Bah, more issues!

**TROUBLESHOOTING #2**

Seems Docker and minikube are a bit confused - maybe related to SKU upgrade:

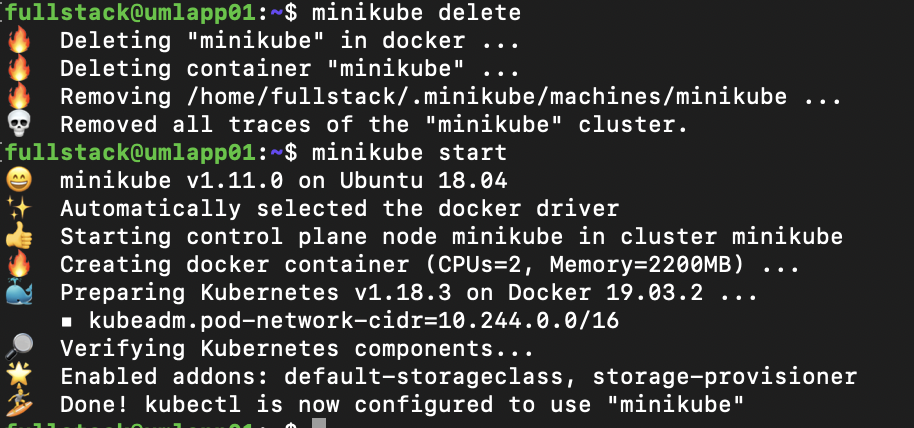


Let's blow away minikube and try a fresh start:

$ minikube delete

$ minikube start

Ok, back on track!



**Setup External kube routing**

The standard setup for minikube doesn't include a good way to access our pods, so we need to do a little preparatory work for internet access (access from outside the VM)

$ minikube addons enable ingress

Check for the ingress services

$ kubectl get pods -n kube-system

Now, we'll try and deploy the cluster (make sure you are in the Azure\_full\_stack/bulletinboard directory)

First, we need to point the Docker commands to use minikube for image building and kubectl deployments

$ eval $(minikube docker-env)

Next, build the app in minikube docker repo

$ docker build --tag bulletinboard:latest .

Finally, deploy the image to a pod

$ kubectl apply -f dockerBB.yaml

Sanity checks:

$ kubectl get pods

**TROUBLESHOOTING #3**

Ok, everything has gone pear-shaped! I didn't know from the research I had done, that Minikube DOES NOT SUPPORT LOADBALANCER!!

At this point, I will need to re-tool the Kubernetes deployment. This means moving the chains on the field to the next iteration (i.e. Project #2). I am confident that after reviewing the issues, I will be able to resolve the cluster issue without impacting the project steps to greatly.

**Musings & take-aways**

I will say that so far, this project has really ramped up my knowledge of Azure, Kubernetes, Docker & Terraform. Prior to this, I had very little Terraform skill and even less Kubernetes.

All in all, I don’t see missing the Kubernetes deployment deadline as a failure, but rather a learning opportunity. I was able to Terreform the target VM (even update it on the fly!) as well as deploy the cluster on my localhost. The Kubernetes Cluster worked from localhost browser, just like the Docker, so I feel vindicated in the fact that I “can” do this. I need to keep sharpening my Cloud/Container skills.

Project #2, here we come!