Readit Deployment with Compute Resources

Creating a VNET, VM for Readit App Deployment

We are deploying the Catalog app from our local machine to IIS Server on Azure VM :

1 : Create a Vnet with default subnet.

2 : Create a VM as usual via the portal in the vnet. Consider a B1s or B2ms machine.

3 : In the networking section, ensure that a new static public IP is created as the default one is dynamic. We cant use it when load balancing as the IP will constantly change after restarts.

4 : After the VM is created and deployed, we have to configure the IIS Server.

After logging in the VM :

5 : From the Server Manager on the VM, inorder to install and host the dotnet application on the IIS Server, select "Add roles and features" and in that enable Web Server Checkbox which will install the IIS Server

6 : This will open an IIS homepage on localhost. Which means our machine is configured as a Web Server.

7 : Inorder for our .Net App to run on IIS Server, we need to download a hosting bundle from Microsoft Website.

Download .NET 8.0 (Linux, macOS, and Windows)

8 : Publish the dotnet application in our local machine inside Vscode by running the actual c# publish command. Copy all the files that are generated in a single folder to the VM's C drive.

- dotnet build : ensure the app is working
- dotnet publish -c Release -o ./publish

9 : Open the IIS Manager on the VM and add a new site by selecting Sites and point it to the folder we have the contents of the dotnet application on VM's drive.

Steps to Add a Site :
1 : Select VM => sites => add a new site and point it to the Drive.
2 : Try changing the port no as well to 8080.

Now try running the application on IIS Server. It should run.

10 : Inorder for the application to run in the browser and be accessible to the internet, we should first :

- add an inbound rule to the VM using its networking settings using the portal to restrict traffic at the VM level
- add a rule to the firewall settings of the VM. This is at the Azure Infra Level
- also check bindings in the IIS Website in the IIS dashboard

Creating another VM for hosting the Weather API on Linux Machine.

After deploying the Catalog Application application on Windows Machine, we will deploy a Weather Nodejs API on Linux machine which will display the weather based on an IP address.

1 : Create a VM following the usual steps using the Linux Image. We will be deploying this VM to the same resource group. We can choose a less powerful machine as the Nodejs API is light-weight.

2 : We can connect to the Linux VM via the Azure cloud shell using SSH and type the following command in sequence :

- sudo apt install git

- sudo apt update

- (Recommended from Node-source) curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
  - OR
- sudo apt-get install -y nodejs
- sudo apt install nodejs
- sudo git clone https://github.com/memilavi/weatherAPI.git
- cd weatherAPI
- sudo apt install npm
- npm start

3 : After successfully running the application, we should keep the catalog VM instance running and then run this separately in cloud shell.

4 : We are essentially connecting to the weather-api hosted on the Linux Machine running Node.js weather-api.

We can enter the private IP of this weather VM in the Weather app that is running on the other VM's catalog app which we ran on IIS Server.

Inorder to expose the Linux VM to the just this Catalog-VM, we can add an inbound rule to the linux VM's Networking settings with Catalog's Public IP.

5 : They should connect and return the weather as they are in the same Vnet.

Deploying Inventory Application on Azure Web App

When we create an App service, there are 2 components viz. App Service (hosts code) and Pricing Plan (Actual Infra hosting the app service)

Pricing plan can have multiple App Services associated with it.
We will select the **Premium V3 (For Vnet Integration)** OR **Standard S1** as we can use staging slots with it.

Ensure we have a dedicated subnet to use Vnet Integration which helps our Azure Web App connect and access our VM inside our Vnet

1 : Create an App Service plan for the Web App. Select the Standard Pricing tier for this application.

2 : It will essentially have a Default Domain waiting for our code to be deployed.

3 : We can log into Azure using VSCode as well and deploy the code files. The steps are :

a : Select Deploy to Web App which is enabled by Azure tools extension and then select the appropriate subscription, RG and Web App.

b : After successful deployment, we can check the code files using App Service Editor and login in to the VM underneath using the console

4 : We can change the pricing plans as well as autoscaling according to the demands of the application.

5 : Web App has multiple outbound IP Addresses to make calls to external services.

6 : Enable Vnet Integration by going into the Networking Settings of the Web App and then selecting the Dedicated Subnet created just for this Vnet integration.

7 : To test for successful connection, we can in Development Tools => Advanced Tools => Go => Debug Console.

- Then run the command :

  tcpping <pvt IP of the other resource> <port no>
  eg. **tcpping 10.0.0.4 1433**

Deployment Slots :

We can even use deployment slots for gradual deployment of our app. Each slot will have a unique URL and we can redirect the traffic to various slots. It will act as a separate App Service.

eg. we can redirect a part of the traffic to the production slot and a part of the traffic to the new slot for testing purpose.

We should deploy the application from VSCode by selecting the deployment slot from the Azure Tools extension and then deploy to slot. After deploying it to the new slot, we can swap them so that the code from the newer staging slot is swapped with the production slot.

Types of Deployment are :

<u>1 : Basic Deployment :</u>
It involves deploying the whole application on all the instances.

<u>2 : Rolling Deployment :</u>
The application is gradually rolled out to the instances incrementally on each server. If it performs well on one server, then it is gradually rolled out to the other instances.

3 : Blue-green Deployment :

It involves deploying the application in separate environments.

- Blue (Current Production) : Existing Version of the application currently serving users

- Green (New Version) : New Version deployed in a separate environment where testers can test the application's functionality.

Docker & Kubernetes

I : Deploying Cart Application on AKS

We have a cart application which we will deploy first on Azure Container Registry.

Azure CLI must be installed and configured for user by logging in. It can be downloaded from Microsoft Learn.

It is accessed by running the command az.

The default method to login is WAM and is done by issuing the following commands in VSCode.

- az account clear
- az config set core.enable_broker_on_windows=false
- az login

Steps are :

1 : Download Docker VS Code Extension that will help us build the image.

<u>We can build the docker image locally on our system but that process is complicated so we'd rather use Container Registry which would build an image from the Dockerfile.</u>

2 : Create a Container Registry on Azure. It will build the actual image from the Dockerfile.
After creating the registry, ensure the **admin user** is enabled under the access keys section.

3 : We need to Login by using the command **az login** in the VSCode terminal which will show us the Container Registry we created in the Docker Extension panel.

4 : We can right click on the Dockerfile and select **Build Image in Azure**.

5 : When prompted, select Linux as the OS for this application.
If failed on this step, repeat the login process with az login command.

6 : After successfully building the image, it will appear in the registry under repositories section.

II : Deploying the Container on AKS

1 : Create a Kubernetes Cluster by searching AKS. Create in the same Resource Group.

2 : For cluster details, select dev/test as it is the least expensive one.

3 : Select recommended approach for Automatic Upgrade.

4 : Go to Integrations tab and select the Image from ACR which the k8s can use for deployment.

5 : Create the AKS.

6 : Inorder to deploy the cart container and access & manage the AKS cluster from our VSCode, download and install the AKS-cli to deploy the container.

- command : **az aks install-cli**

7 : For safety measure, run az login command again.

8 : For connecting to the **kubectl,** try running the command :

az aks get-credentials --resource-group readit-app-rg --name cart-aks
modify it according to the rg or app name

<u>This command will merge the credentials with kubectl</u>

9 : After connecting to kubectl, run the command :
kubectl get nodes

which indicates we can access the cluster and shows us the running nodes (VM's).

10 : We will then deploy the **yaml** file to the cluster which contains all the configuration for the workloads. Try changing the image name in the container section to the appropriate one.

The command to deploy is :

kubectl apply -f <yaml file name>

This command creates :

1 : deployment : The actual pods inside the cluster

2 : Service : The public endpoint that gets us the access to the pods

3 : Check under the services and ingress section in the portal to monitor the running services.

<u>Azure Functions</u>

We generally use Azure Functions inorder to run them if a particular event occurs.

- They are completely managed by Azure and have flexible pricing plans.
- They scale automatically as they are managed by azure.
- They are taken down by Azure after inactivity

We use triggers and bindings when implementing them. Triggers basically respond to events and bindings are used to pass arguments to the triggers.

Many of the Azure Services can be used as Triggers and Bindings.

eg. Run a timer trigger to run every 5 minutes and then calculate the sum of a column in DB and then send an event in EventGrid (binding)

The functions can have cold or warm start according to the hosting plan we choose. Cold start requires certain steps to run such as :

- allocation of unspecialized server
- Specialising the worker
- resetting the runtime

- loading the function into the memory
- Code runs

<u>Warm start is always preferred as the function is ready to run.</u>

There are 3 types of hosting plans in Azure Functions:

1 : Consumption Plan : We pay for what we use but there is a limit of 1.5gb ram.

we can calculate the monthly costs of Consumption plan by :

- considering executions per month
- avg memory consumed
- avg execution duration

eg: if we have a function that ran 9 million times in a month with avg execution duration of 1.5 secs and avg memory consumption of 800 mb, we can calculate the payment for execution time as follows :

function ran for : 9 mil * 1.5 secs = **13.5 mil secs**

total gb consumed/sec = 13.5 * 0.8 = 10.8 gb - 400 = **10.4 gb/sec**

400 subtracted as Azure grants 400 gb/s per month
monthly payment for execution time : 10.4 * 0.000016 gb/s = **166.4 dollars**

2 : Premium Plan : As the name suggests, it is more expensive option that helps us run the function with a warm start as opposed to cold one in consumption plan. They are useful for time-sensitive tasks.

3 : Dedicated Plan : It runs on the same server as our App Service and its only downside is it does not autoscale as the other 2.

There is a concept called Durable Functions as well that are useful for function chaining when we want to cal one function after another.

Deploying our Order App on Azure Functions

Running the Functions Locally :

We will first run the Order App locally. Inorder to do that, we will download the Azure Function Core Tools from Microsoft.

After running the application, we will have 2 URL's, viz. one for Storage and one for Cosmos DB. We can run those locally in Postman.

- these will be Post reqs with body copied from **ordersample.json**
- the output will be the req body with order details displayed on the VS code terminal.

Deploying and running it on Azure Function App :

1 : Create Azure Function App in the Portal.

- Select appropriate Plan, OS and Runtime
- Select a storage as function apps use storage to store the functions and its parameters.

2 : Run/debug the the application locally and then under Workspace tab of Azure Tools extension, select the function app icon and deploy to Azure.

- make POST requests using the 2 urls generated after running the order application in VsCode using Postman and copy the ordersamples.json contents in the req body

3 : After deploying, even select Upload settings to upload the local settings to Azure Functions App.

4 : In the portal, inside the Function App, locate the 2 urls and click on them. Then after opening, select get function url and then select any of the url's and test them in Postman.