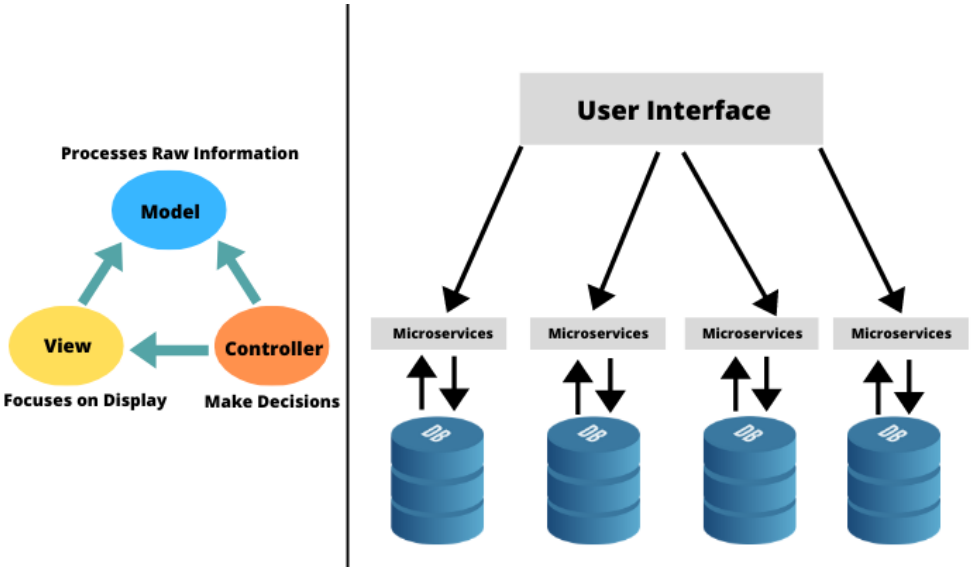
The purpose of this word file is to understand how we could implement MVC project (server-side rendering) with a web API backend that follows the microservice architecture and communicates with a SQL database on Azure. The hosting of this demo will be on Azure kubernetes managed service "AKS" on Azure. Kindly note that I am also a new learner for this track. So, if you are more familiar with Azure cloud provider and you see that there's a better approach than the one demonstrated here please reach me out on ksaad@ejada.com

Microservice introduction

1- Microservices architecture (often shortened to microservices) refers to an architectural style for developing applications. Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility. To serve a single user request, a microservices-based application can call on many internal microservices to compose its response. Containers are a well-suited microservices architecture example, since they let you focus on developing the services without worrying about the dependencies. Modern cloud-native applications are usually built as microservices using containers. (What Is Microservices Architecture? | Google Cloud)

2- In the following image we have divided our system into 4 separates parts (microservices) where each service communicates with the database and once the request is resolved, the response is being passed by to the user.



3-

4- After we have seen a brief introduction on microservice, we will introduce our approach used in this demo.

5- Our project is divided into:

   a. MVC project (frontend server-side rendering)

      i. View: represents the native HTML&CSS that used to render the page that being passed to the user (after being filled with the data from the backend)

      ii. Controller: used to communicate with the backend (2 microservices) through HTTP requests to retrieve the needed information and embedded this info inside the view before returns it to the user.

      iii. Model: represents the entities that being used in our application

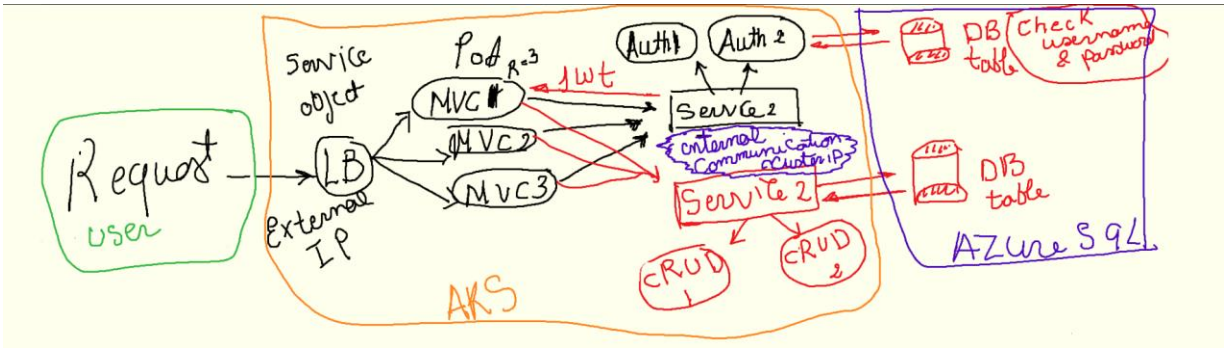
iv. This project is being containerized after being implemented and pushed to docker.io registry. Finally, this image is used inside kubernetes cluster to run a container inside the pod (we will have 3 replicas of this pod as shown in the next image)

b. ASP.Net Core Web API (backend)
   i. This is based on the MC without the need of V in the MVC approach.
   ii. We will have 2 microservices
      1. Authentication service that is responsible for determining whether the username and password are valid or not when the user needs to login to the system. If the credentials are valid, a JWT (JSON Web Token: JSON Web Token - Wikipedia)
      2. CRUD service that is responsible for the operations that the user could execute on the database after being sure that the user is allowed to perform this action using the JWT generated from the first microservice.
   iii. These projects are being containerized after being implemented and pushed to docker.io registry. Finally, the images are used inside kubernetes cluster to run containers inside the pods (we will have 2 replicas of each service as shown in the next image)

c. SQL database hosted on the cloud that is configured to communicate with the backend microservices using Entity framework following the code first approach (read the difference: Code First vs Database First. Know the difference - Chubby Developer)

d. Azure kubernetes cluster hosted on the cloud to manage this infrastructure

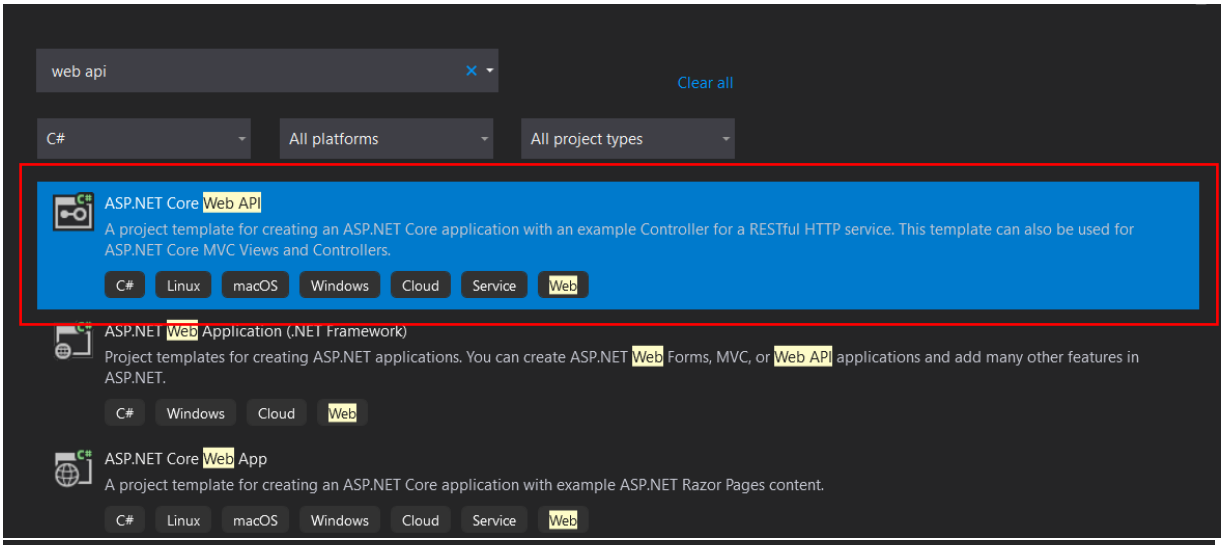
6-

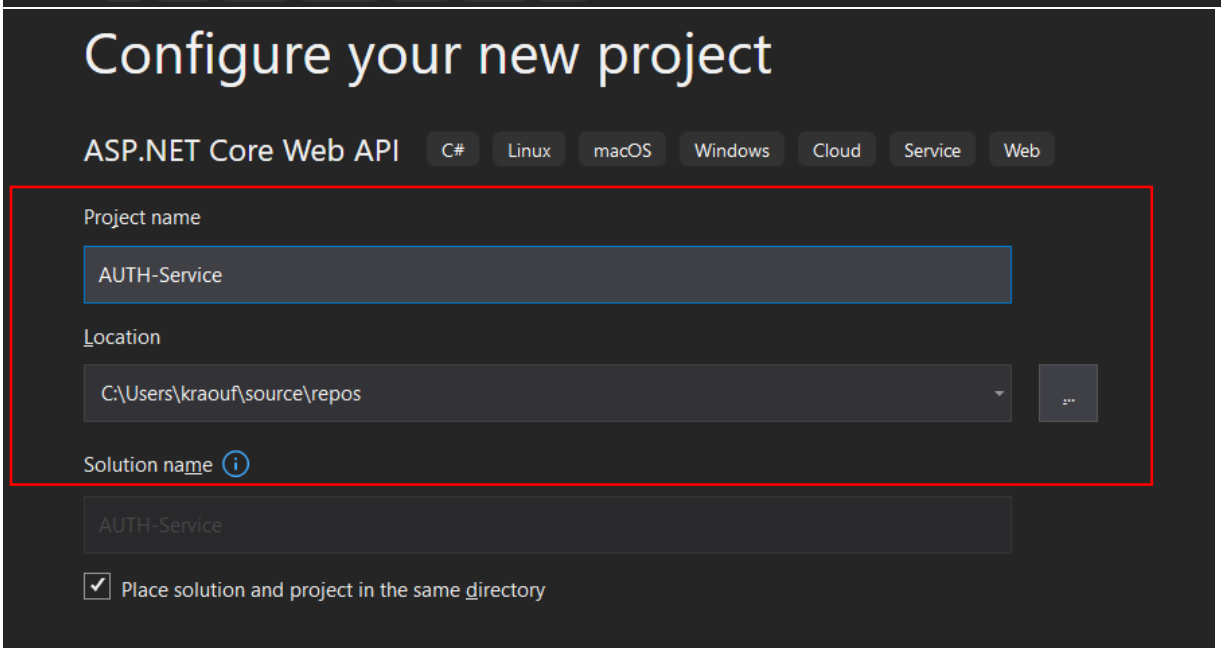


Authentication service

Setting working environment

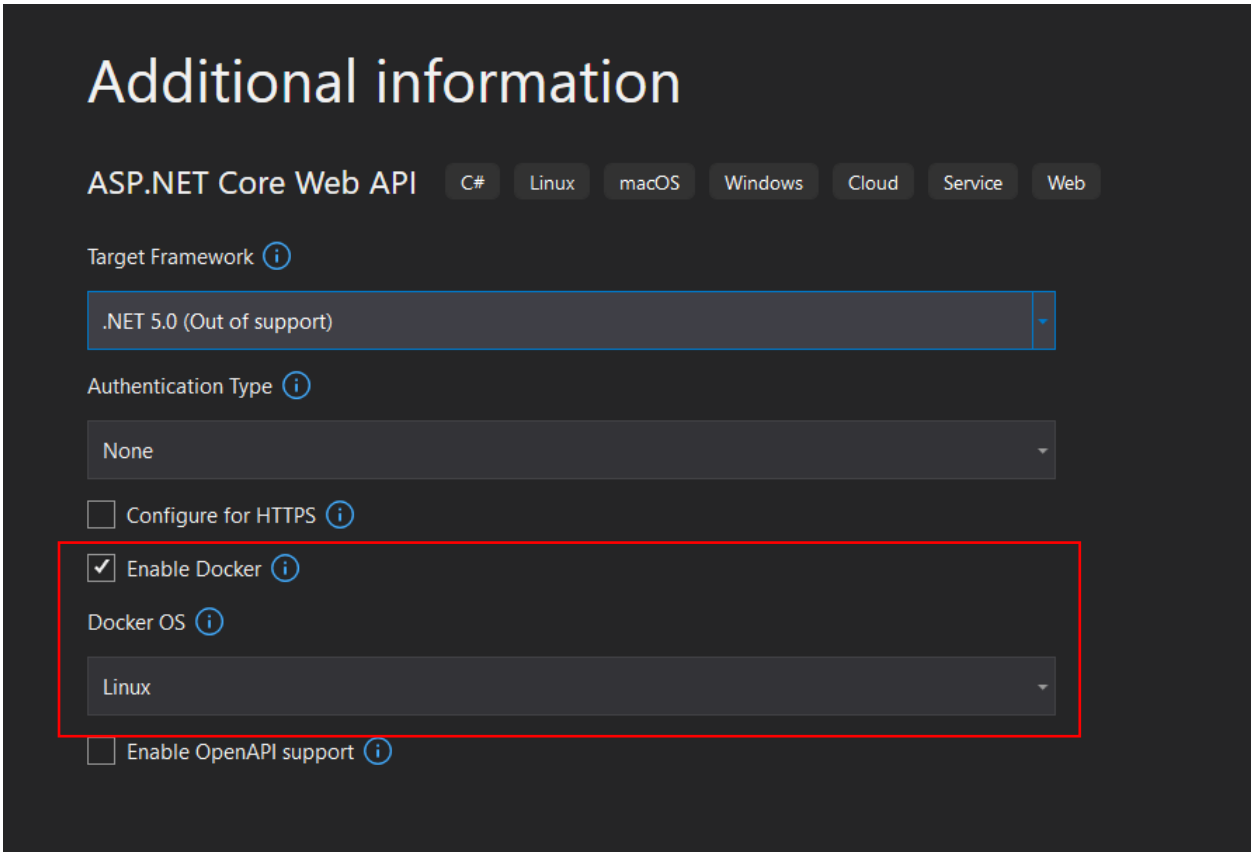
1- We will open our visual studio 2019 and create w new project as shown in the next image.

2- 

3- 

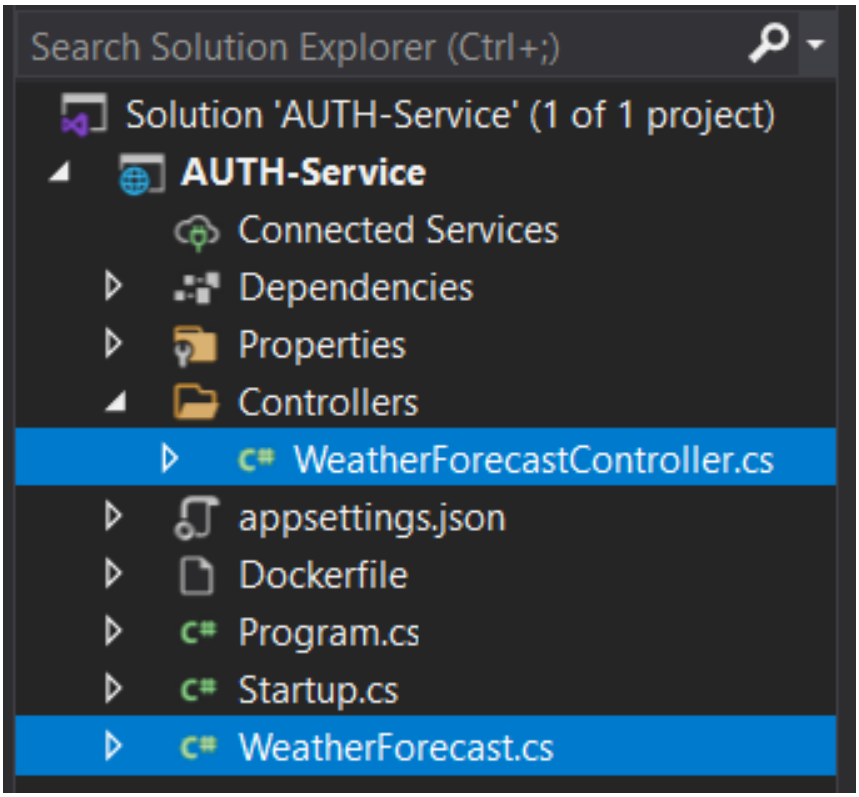
4- Make sure you have selected the option for docker in order to generate for us a docker file that could be used later to create our image without the need to create this file by our own.

5-

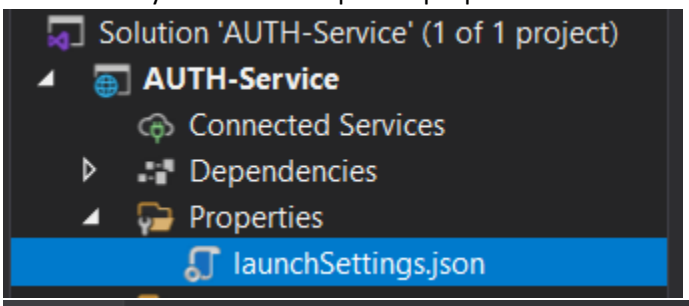


6- Once our project is created we will see the following image. Remove the “WeatherForecast.cs” file and “WeatherForecastController.cs”.

7-



8- Open the “launchSettings.json” file and apply the following configurations. These configurations are used only for the development purpose not related to the kubernetes deployment.

9- 

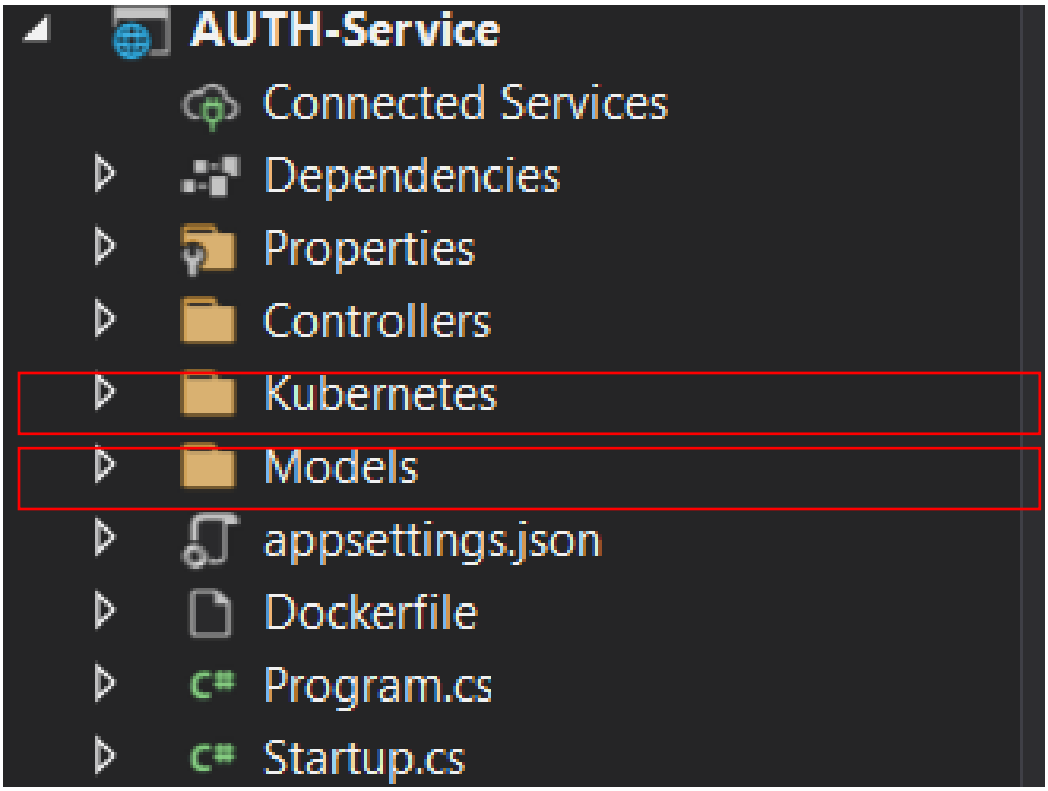
10- Schema: http://json.schemastore.org/launchsettings.json

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:28424",
      "sslPort": 44364
    }
  },
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": false,
      "launchUrl": "",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "AUTH_Service": {
      "commandName": "Project",
      "launchBrowser": false,
      "launchUrl": "",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "dotnetRunMessages": "true",
      "applicationUrl": "http://localhost:5000"
    },
    "Docker": {
      "commandName": "Docker",
      "launchBrowser": true,
      "launchUrl": "{Scheme}://{ServiceHost}:{ServicePort}/weatherforecast",
```

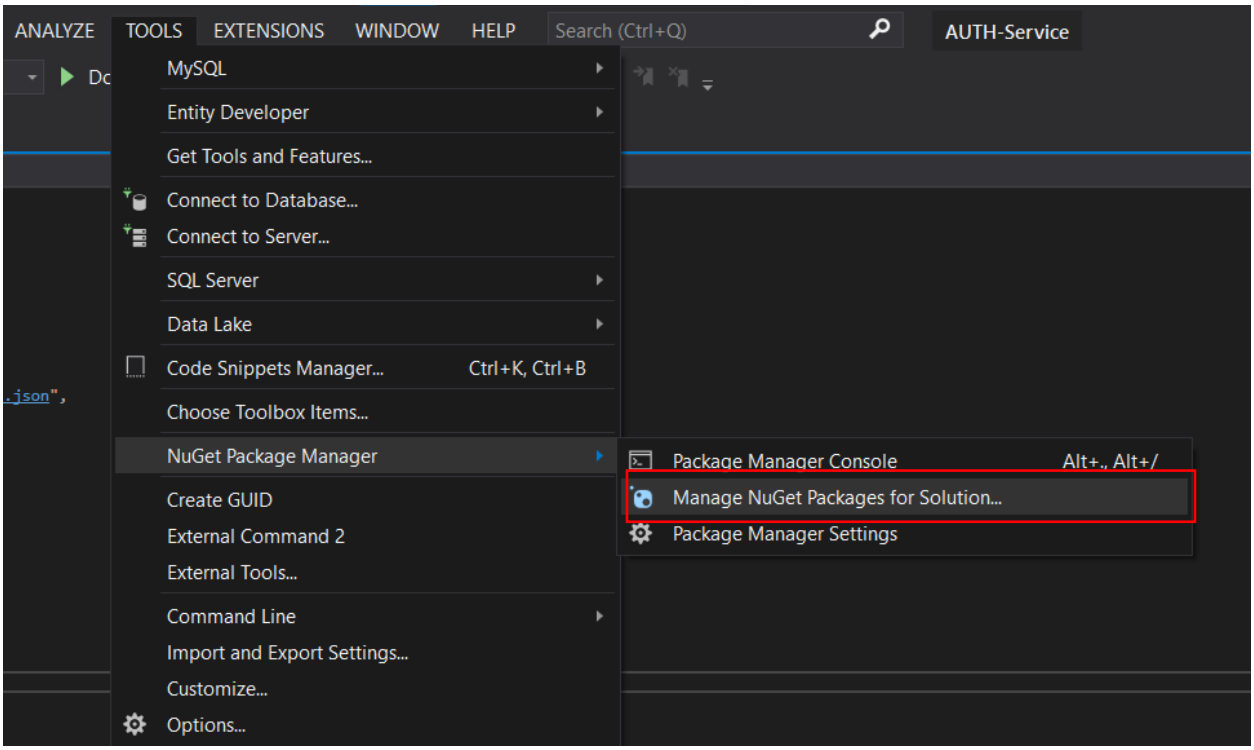
11- Create a folder and name it “Kubernetes”. This folder will hold another file that shall be used in our deployment section later.

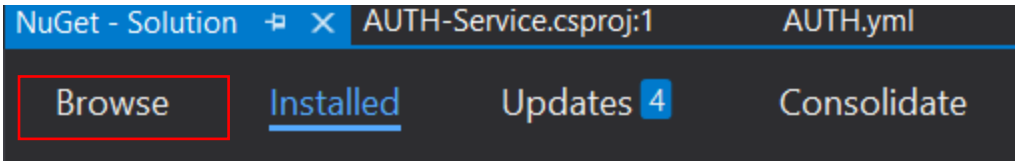
12- Create a folder and name it “Models”. This folder will hold our entities and the schema classes used to create our database.

13- The following image shows the final view.

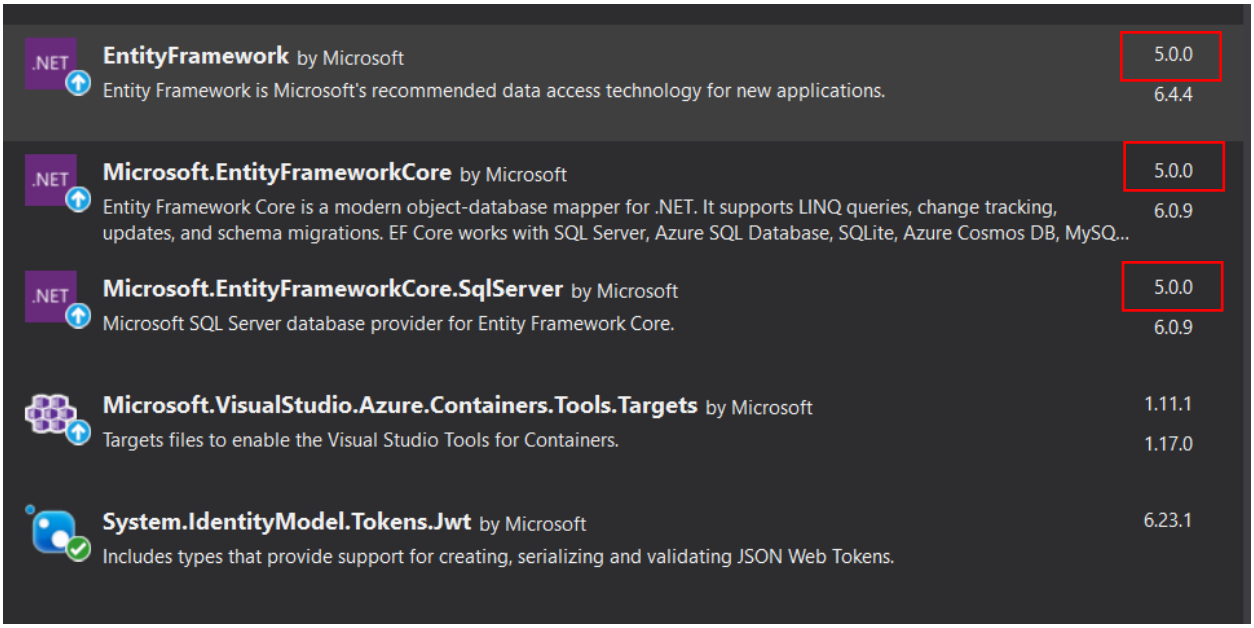
14- 

15- Before we start, we need to install some dependencies and libraries like Entity framework and some tools used by this library.

16- 

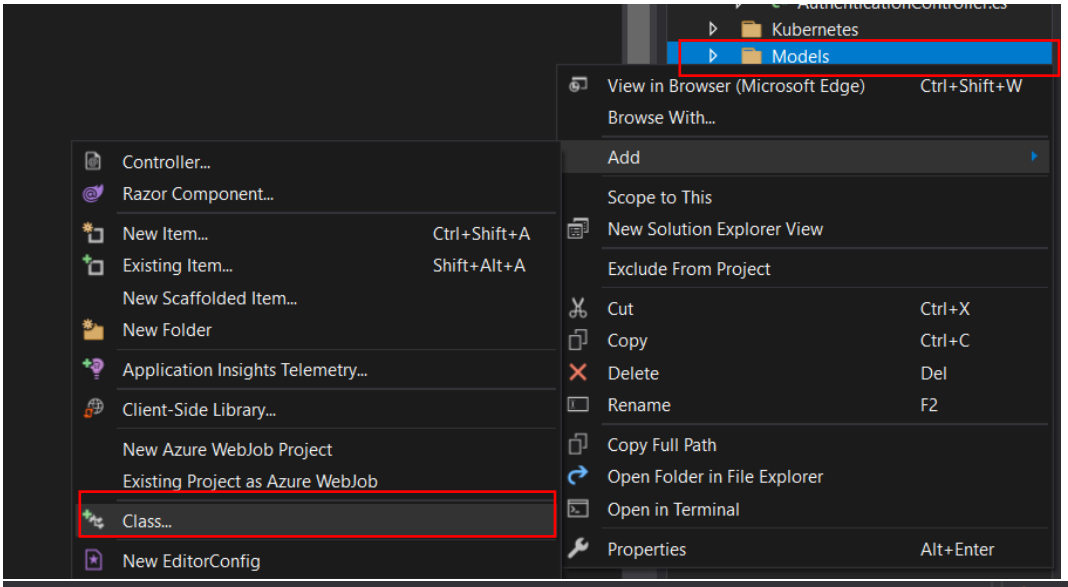
17- 

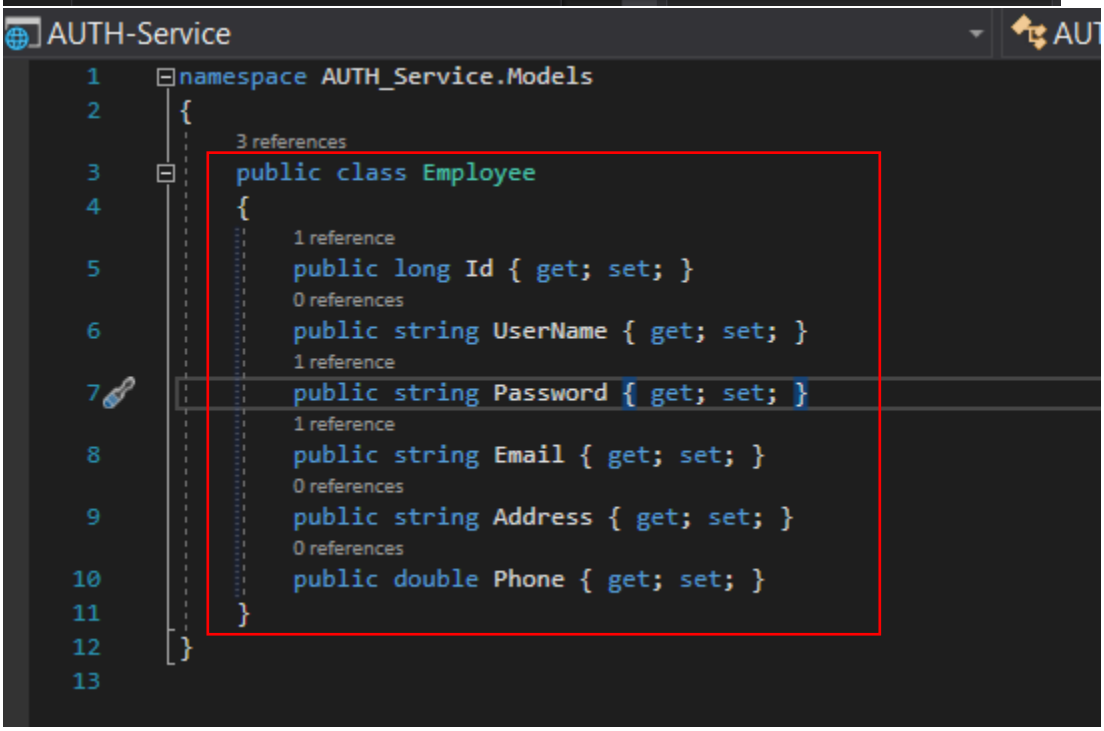
18- Search for the following packages and install them (make sure the version used is the same as the .Net version selected when you create the project)

19- 

Configuring Entity framework

1- First, we define create a class model named “Employee” that represents our database schema used by EF.

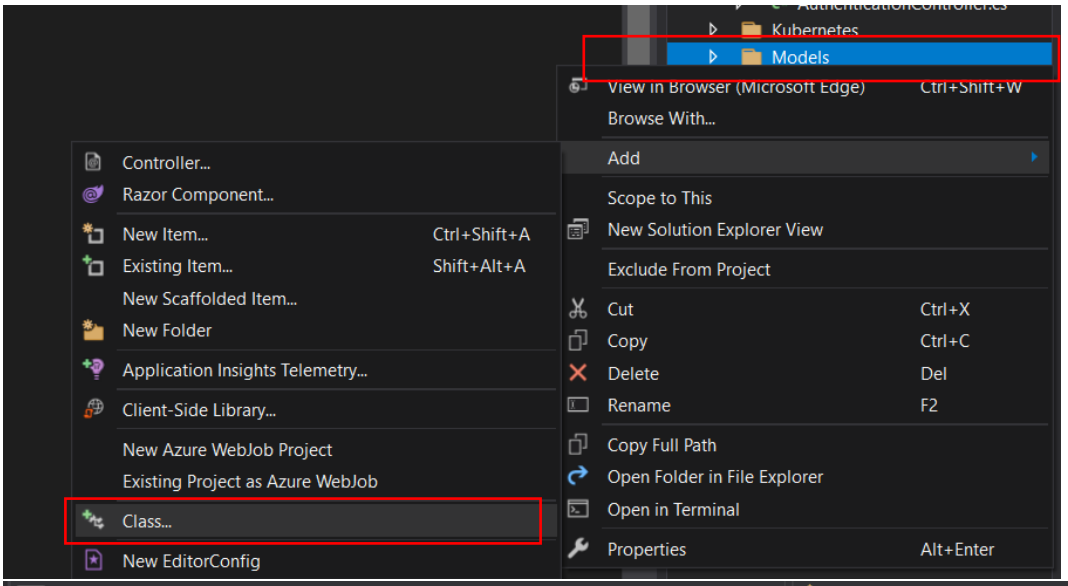
2- 

3- 

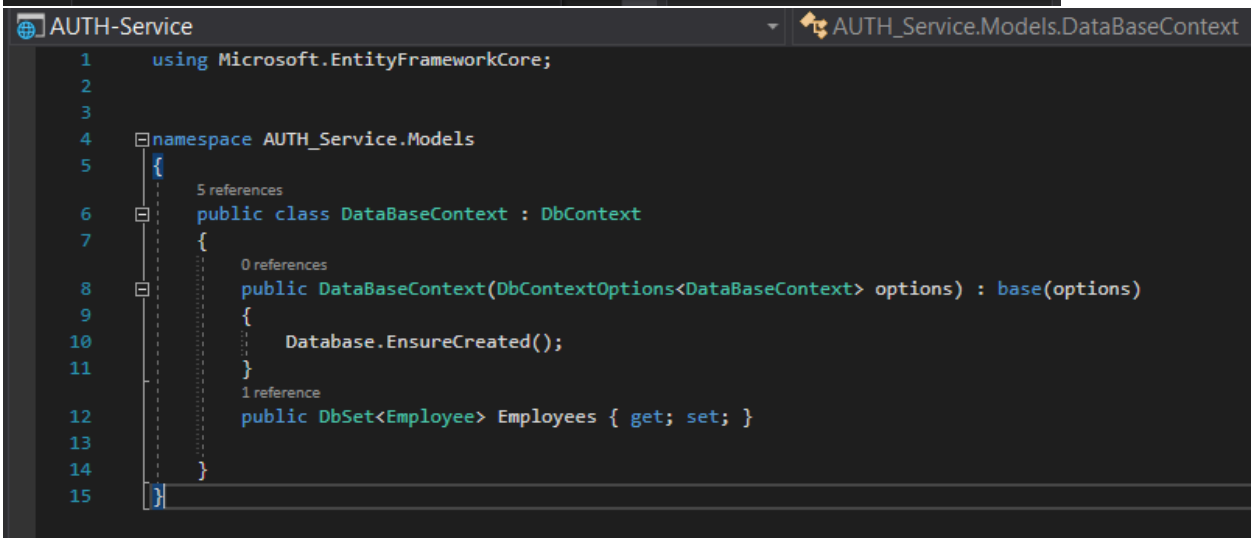
4- Next, we create our database context class. this class will do two primary things

   a. Use our “Employee” class and map this to a real database table inside the database
   b. A constructor that accepts the connection string to connect (dependency injection) to the cloud database and creates the database. This constructor also checks if the database is already created (won’t create a new database) before create it.

5-

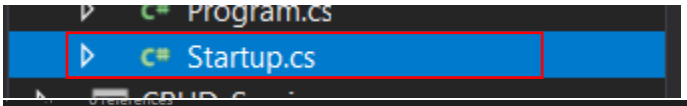


6-

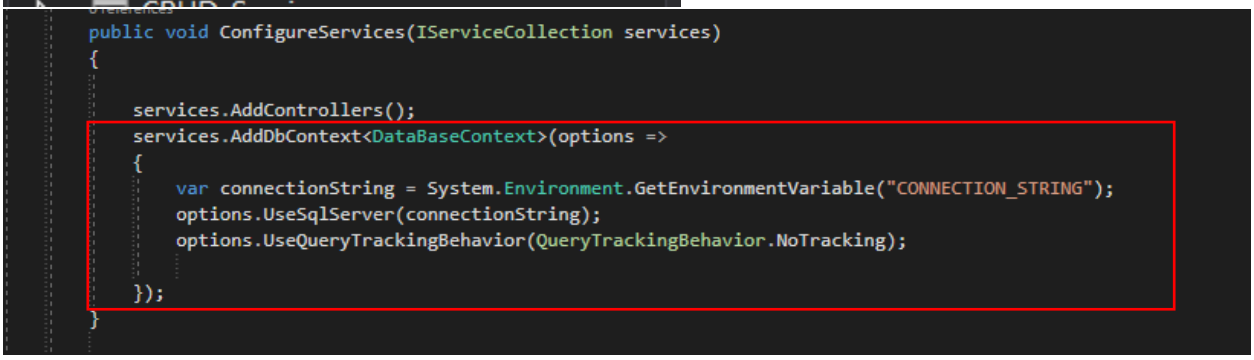


7- The final step is to pass the connection string and configure the EF service to create the database context. Kindly note that we are using environment variable to pass the connection string. We will see in the kubernetes deployment section the real value of this string after creating Azure SQL database.

8-

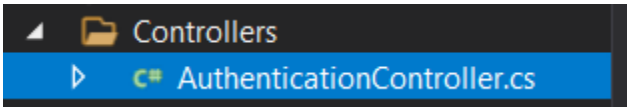


9-

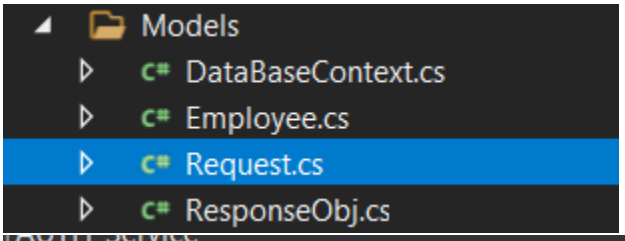


Creating authentication service controller

1- We will create a controller class named “AuthenticationController.cs”

2- 

3- We will create 2 model classes (one class represents the request object that our authentication will accept and another class represents the response)

4- 

5-
```
namespace AUTH_Service.Models
{
    1 reference
    public class Request
    {
        1 reference
        public string Email { get; set; }
        1 reference
        public string Password { get; set; }
    }
}
```

6-
```
namespace AUTH_Service.Models
{
    2 references
    public class ResponseObj
    {
        1 reference
        public string jwt { get; set; }
    }
}
```

7- The next image shows the authentication controller class itself.

   a. The constructor accepts a database context object (injected using dependency injection) that represents our database

b.

```
namespace AUTH_Service.Controllers
{
    [ApiController]
    [Route("api2")]
    1 reference
    public class AuthenticationController : ControllerBase
    {
        private readonly DataBaseContext _db;

        0 references
        public AuthenticationController(DataBaseContext db)
        {
            _db = db;
        }
```

c. The login endpoint (API), this one will check the passed request (username and password) against the cloud SQL database. If an entry is found, it will create a JWT where the payload will be the user ID inside the database. The following image shows this logic

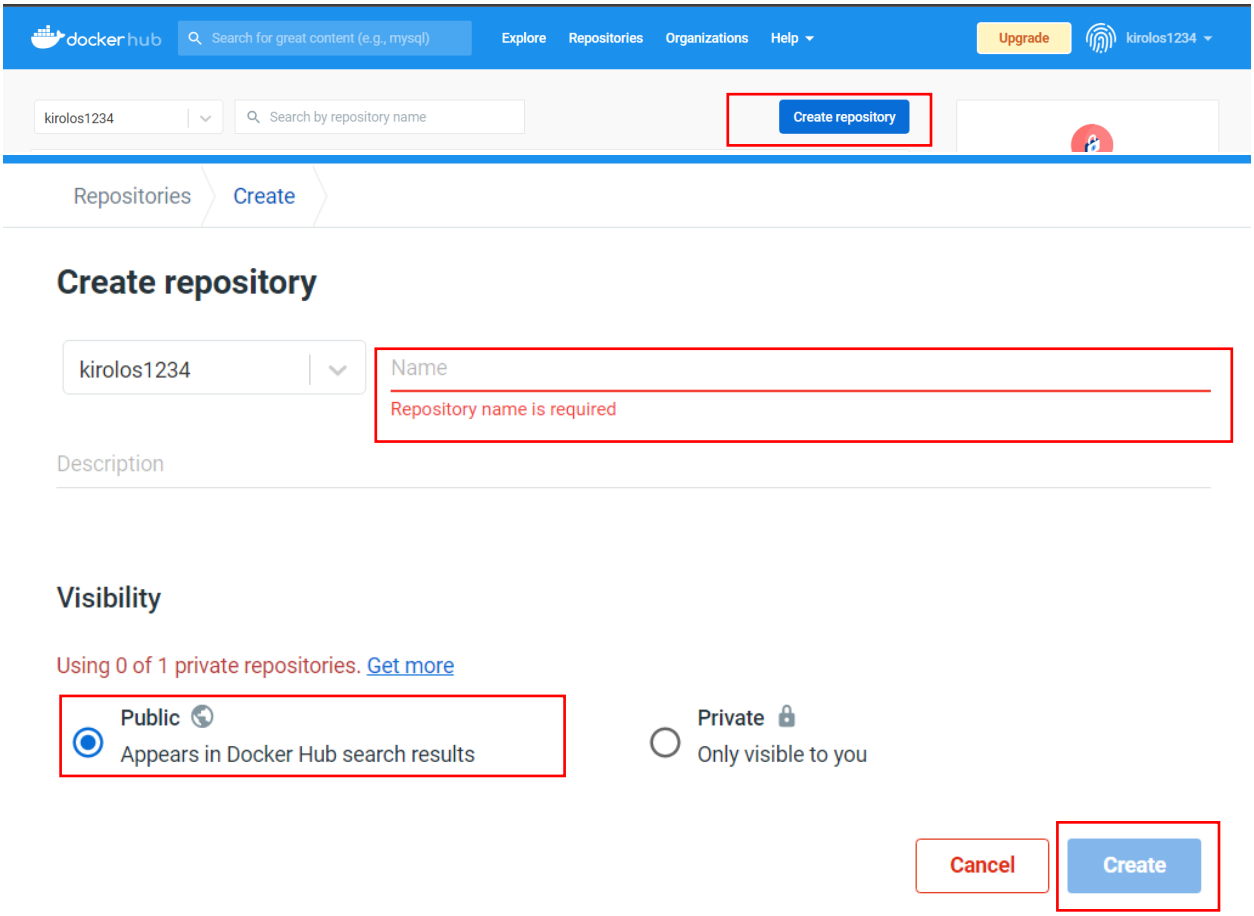
d.

```
public IActionResult LoginUser([FromBody] Request req)
{
    string email = req.Email;
    string password = req.Password;
    Employee emp = _db.Employees.Where(m => m.Email == email && m.Password == password).FirstOrDefault<Employee>();

    if (emp == null)
    {
        return NoContent();
    }
    // Define const Key this should be private secret key  stored in some safe place
    string key = System.Environment.GetEnvironmentVariable("JWT_SECRET_KEY");

    // Create Security key  using private key above:
    // not that latest version of JWT using Microsoft namespace instead of System
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));

    // Also note that securityKey length should be >256b
    // so you have to make sure that your private key has a proper length
    //
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256Signature);

    //  Finally create a Token
    var header = new JwtHeader(credentials);

    //Some PayLoad that contain information about the  customer
    var payload = new JwtPayload { { "id ", emp.Id } };

    //
    var secToken = new JwtSecurityToken(header, payload);
    var handler = new JwtSecurityTokenHandler();

    // Token to String so you can use it in your client
    ResponseObj res = new ResponseObj();
    res.jwt = handler.WriteToken(secToken);

    return Ok(res);
}
```

Containerization

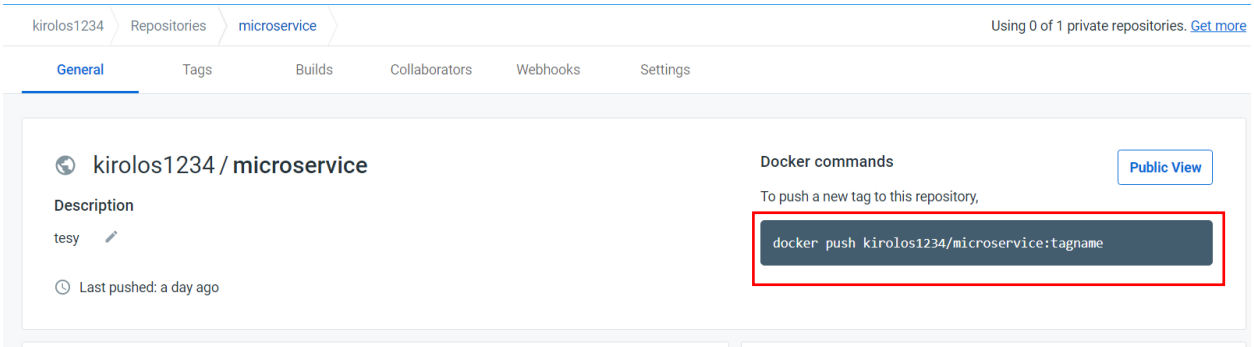
1- Create a new account on docker.io https://docker.io and a public repository

2-



3-

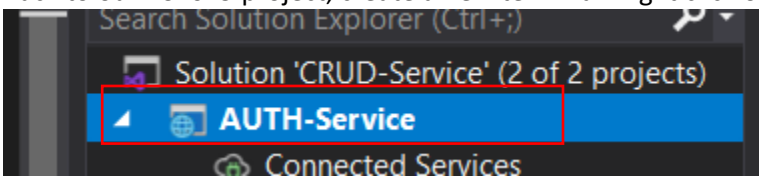
4- Copy the following command as we will use it later



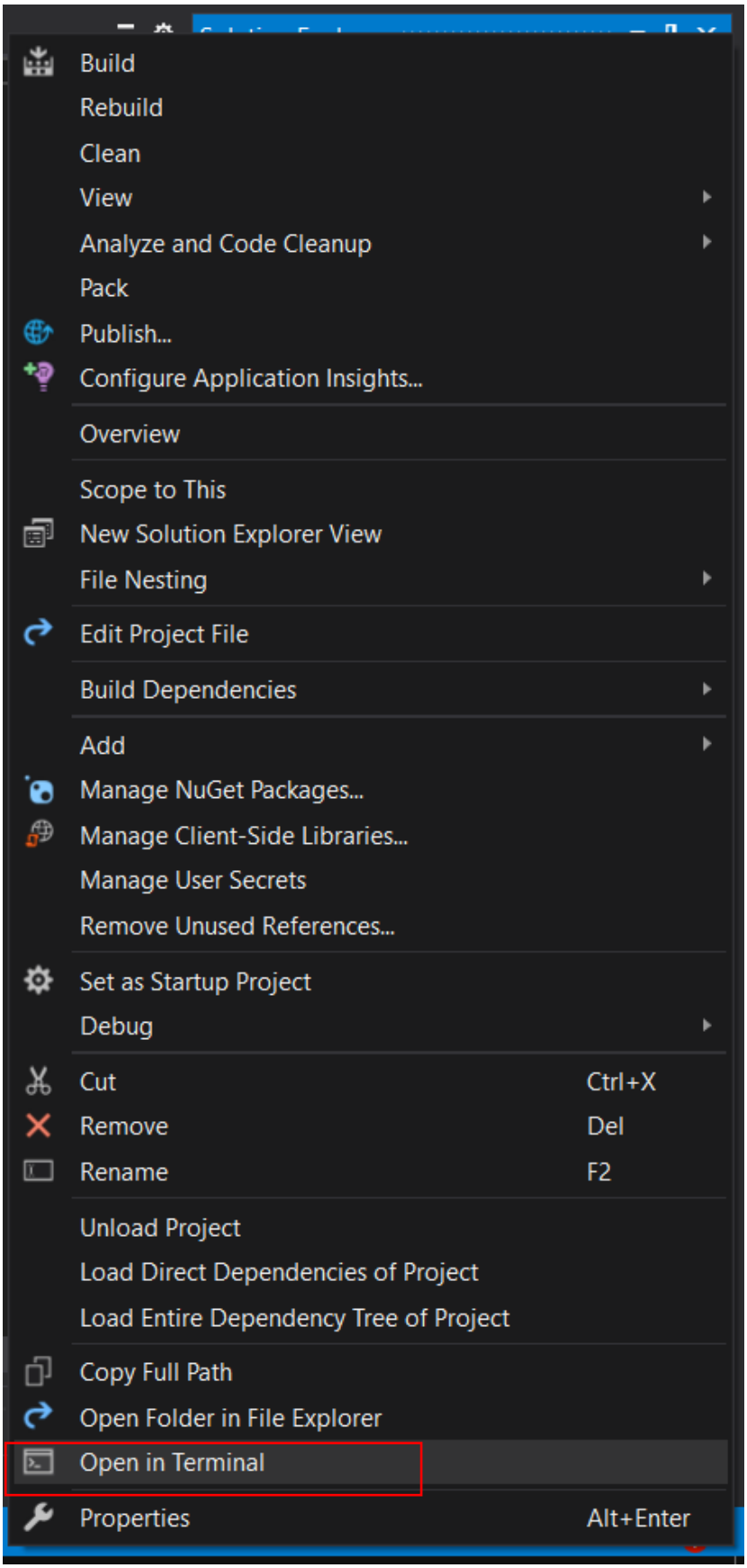
5-

6- Back to our VS2019 project, create a new terminal. Right click of the project

7-



8-


Build
Rebuild
Clean
View
Analyze and Code Cleanup
Pack
Publish...
Configure Application Insights...
Overview
Scope to This
New Solution Explorer View
File Nesting
Edit Project File
Build Dependencies
Add
Manage NuGet Packages...
Manage Client-Side Libraries...
Manage User Secrets
Remove Unused References...
Set as Startup Project
Debug
Cut Ctrl+X
Remove Del
Rename F2
Unload Project
Load Direct Dependencies of Project
Load Entire Dependency Tree of Project
Copy Full Path
Open Folder in File Explorer
Open in Terminal
Properties Alt+Enter

9- Create the image using the following command

```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service> docker build -t kirolos1234/microservice:auth-service .
```

10-

11- Push image to docker.io repo

```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service> docker push kirolos1234/microservice:auth-service
```

12-

13- **Note**: if you get an error with the previous push command, you need to login to your account from the terminal as shown next

```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service> docker login
```

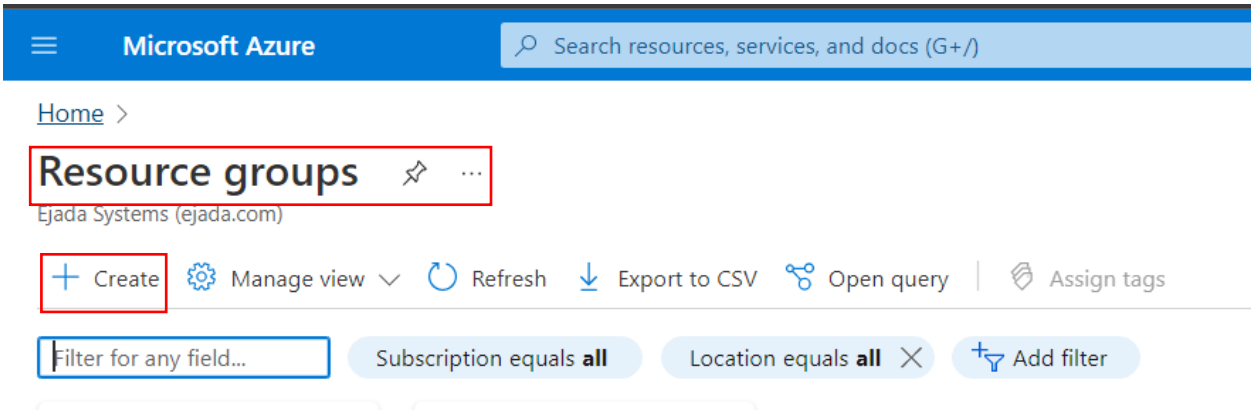
14-

15- Back to our docker.io repo we will examine the following

16-

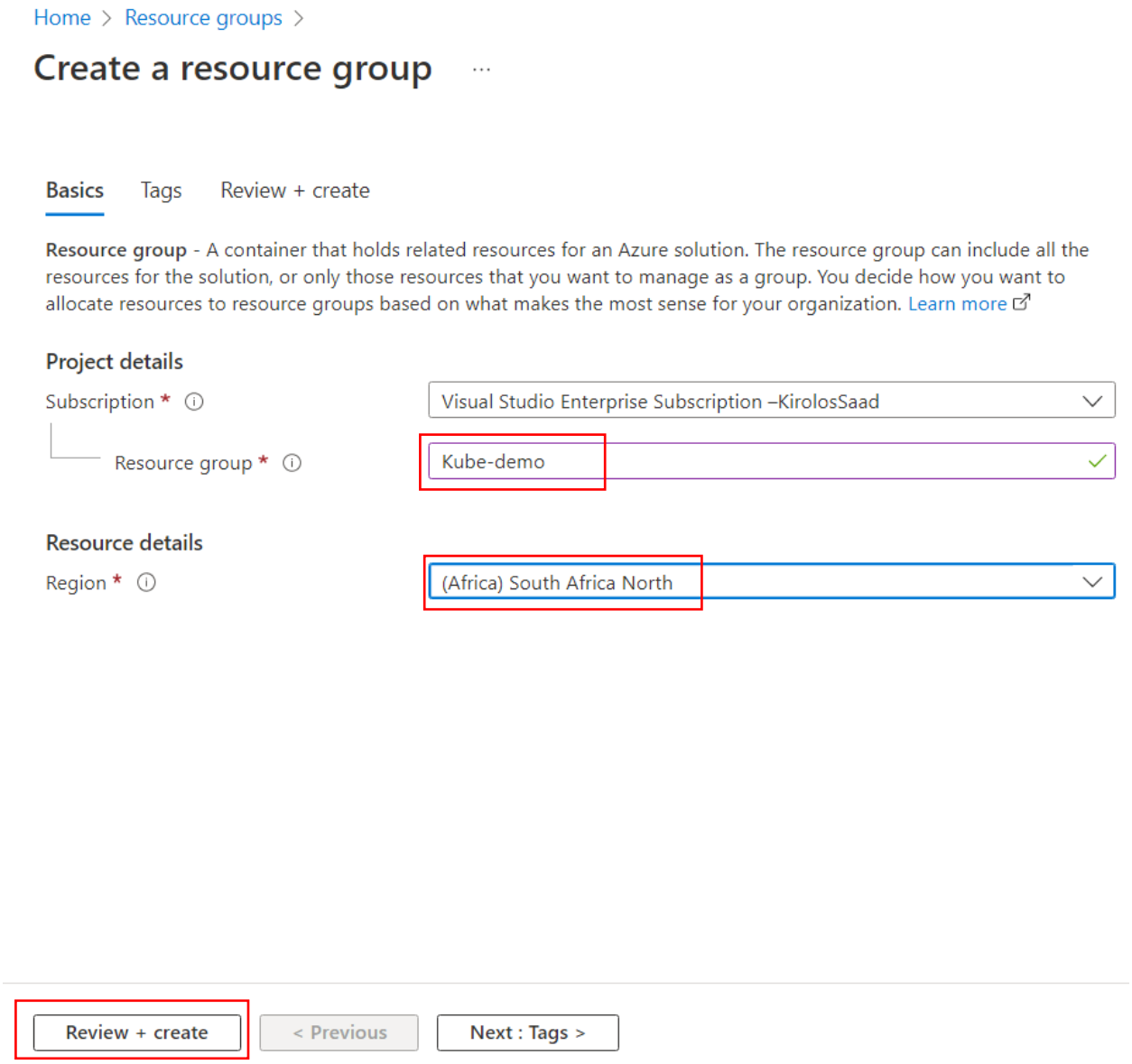
Creating Azure SQL database

1- Login to you Azure account and create a new resource group for your work



2-

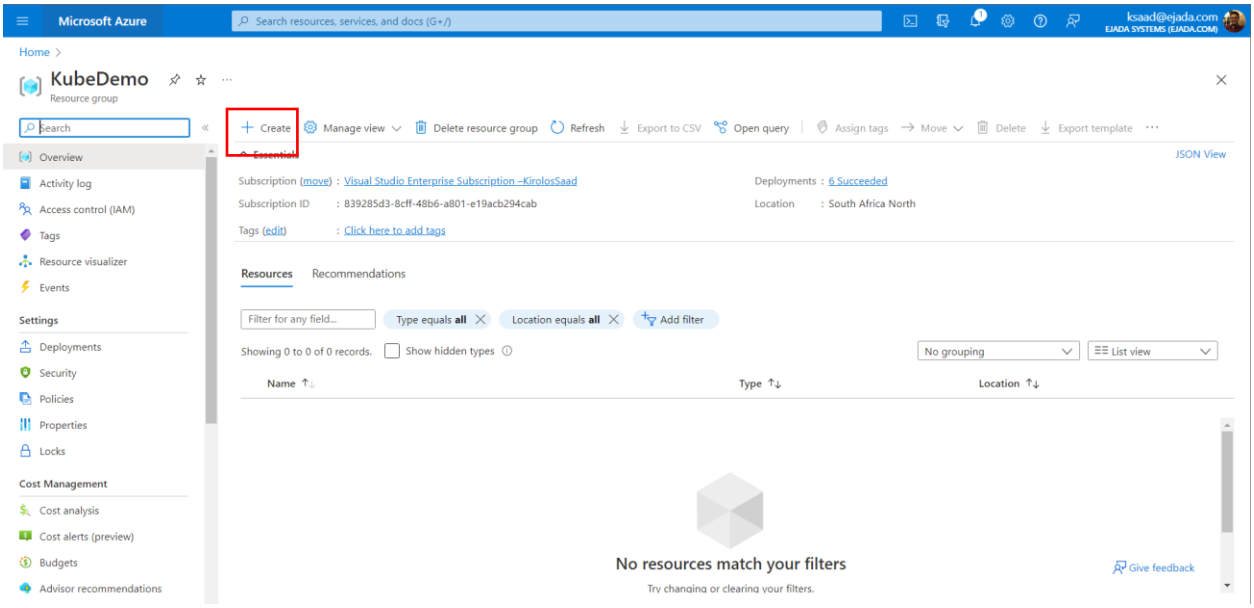
3- Give it any name you want and select the region you want



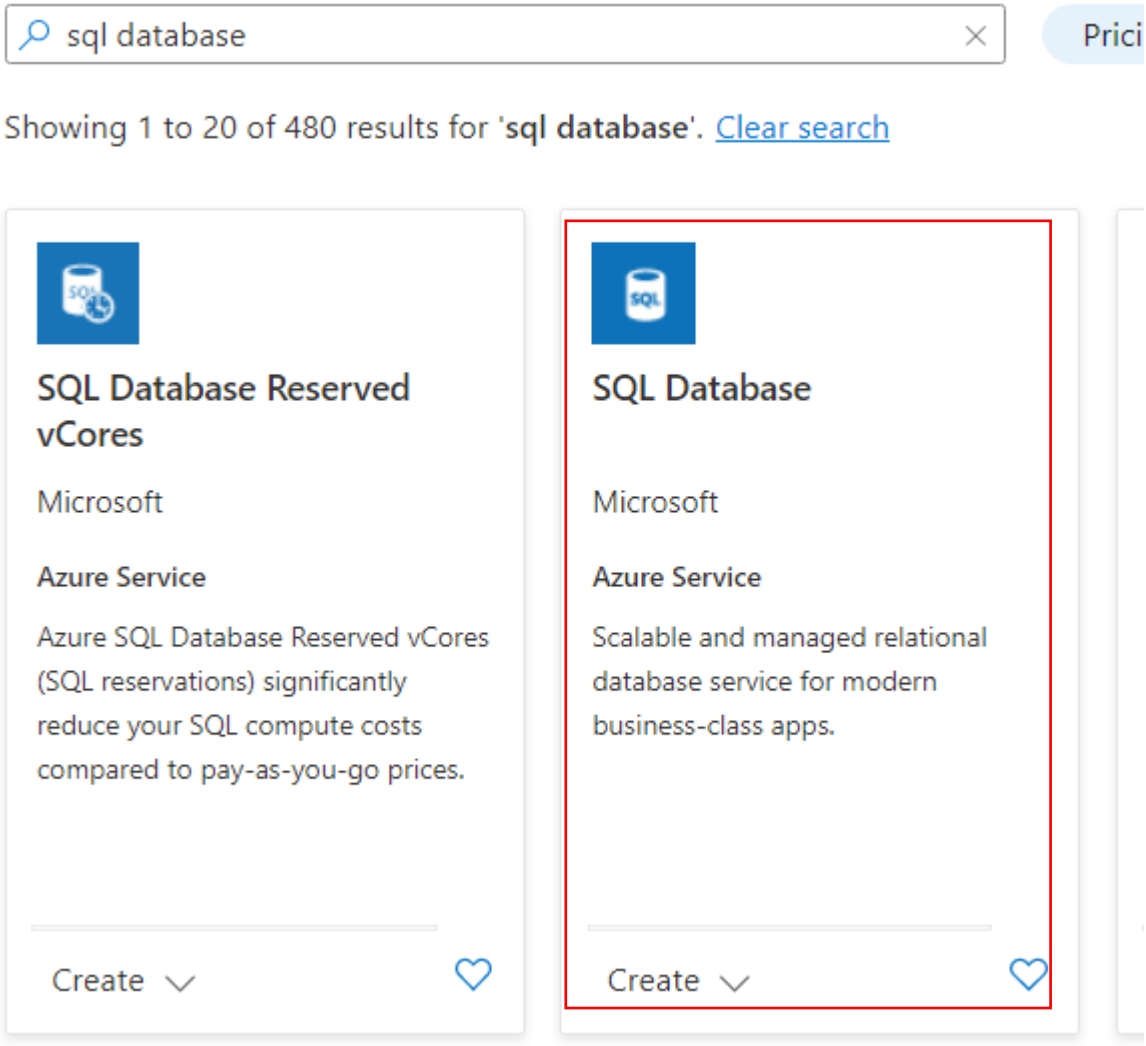
4-

5- Open the resource group after being created

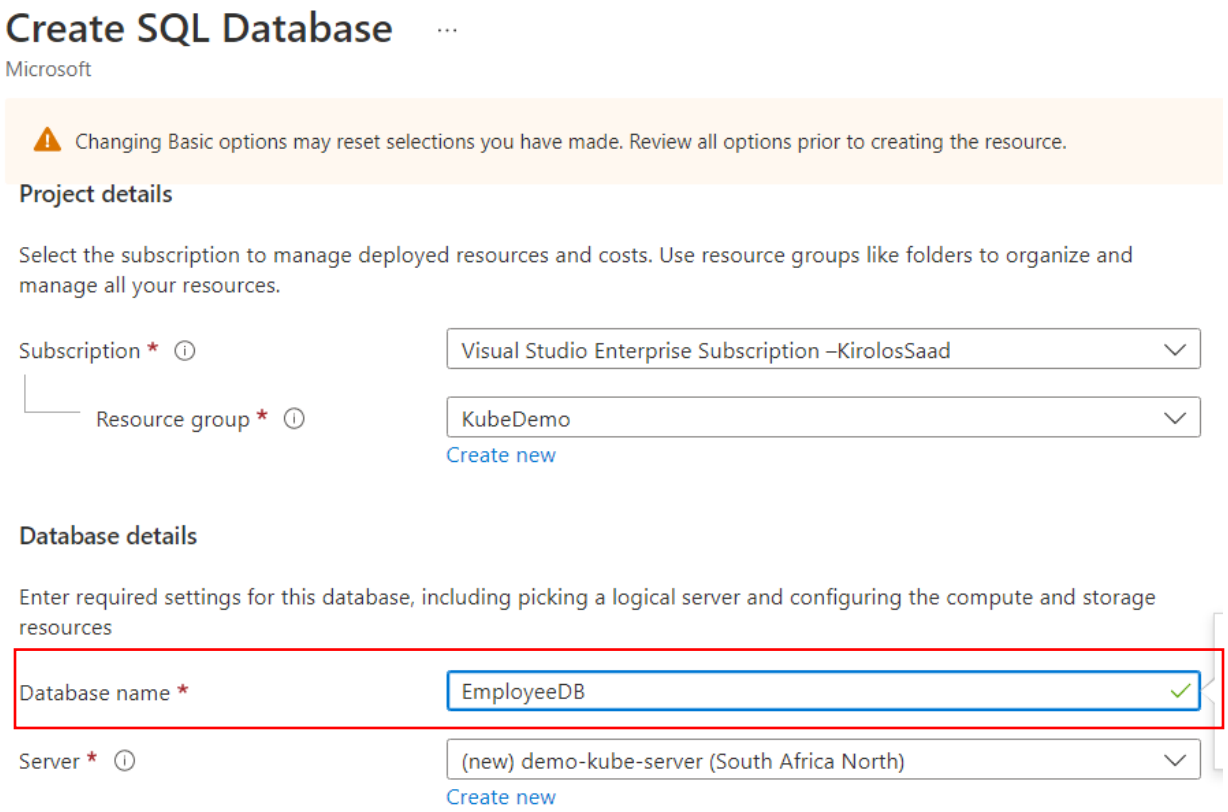
6-


Microsoft Azure
Search resources, services, and docs (G+/)
ksaad@ejada.com
EJADA SYSTEMS (EJADA.COM)
Home
KubeDemo
Resource group
Search
Create
Manage view
Delete resource group
Refresh
Export to CSV
Open query
Assign tags
Move
Delete
Export template
JSON View
Overview
Activity log
Access control (IAM)
Tags
Resource visualizer
Events
Settings
Deployments
Security
Policies
Properties
Locks
Cost Management
Cost analysis
Cost alerts (preview)
Budgets
Advisor recommendations
Subscription (move) : Visual Studio Enterprise Subscription –KirolosSaad
Subscription ID : 839285d3-8cff-48b6-a801-e19acb294cab
Tags (edit) : Click here to add tags
Deployments : 6 Succeeded
Location : South Africa North
Resources
Recommendations
Filter for any field...
Type equals all
Location equals all
Add filter
Showing 0 to 0 of 0 records.
Show hidden types
No grouping
List view
Name
Type
Location
No resources match your filters
Try changing or clearing your filters.
Give feedback

7-

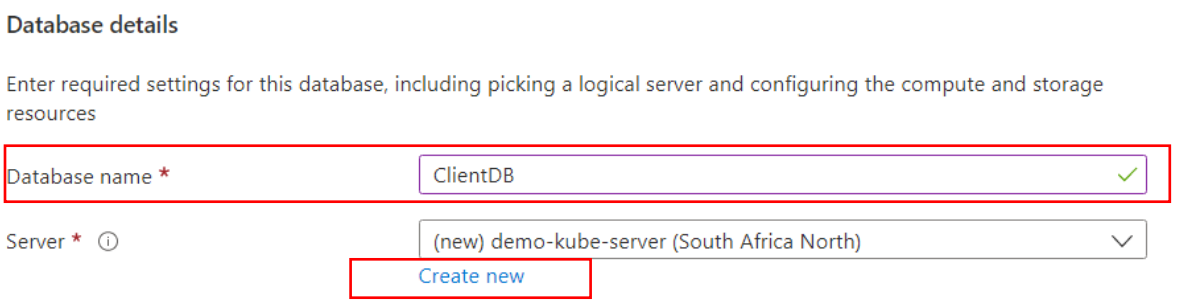

sql database
Prici
Showing 1 to 20 of 480 results for 'sql database'. Clear search
SQL Database Reserved vCores
Microsoft
Azure Service
Azure SQL Database Reserved vCores (SQL reservations) significantly reduce your SQL compute costs compared to pay-as-you-go prices.
Create
SQL Database
Microsoft
Azure Service
Scalable and managed relational database service for modern business-class apps.
Create

8- Select your resource group created before



9-

10- Give your database name and create a SQL server in case you don't have one



11-

12- During the creation of the SQL server, you can pick the authentication method you want. Here we are going to use username and password for our database

Create SQL Database Server

Microsoft

Enter required settings for this server, including providing a name and location. This server will be created in the same subscription and resource group as your database.

Server name * demo-kube-server .database.windows.net

Location * (Africa) South Africa North

Authentication

Select your preferred authentication methods for accessing this server. Create a server admin login and password to access your server with SQL authentication, select only Azure AD authentication Learn more using an existing Azure AD user, group, or application as Azure AD admin Learn more, or select both SQL and Azure AD authentication.

Authentication method
- Use only Azure Active Directory (Azure AD) authentication
- Use both SQL and Azure AD authentication
- Use SQL authentication

Server admin login * ServerAdmin123321

Password * ••••••••••••••

Confirm password * ••••••••••••••

13-

14- As we are working in a development demo, we will pick the lowest SQL server configuration

Want to use SQL elastic pool? Yes No

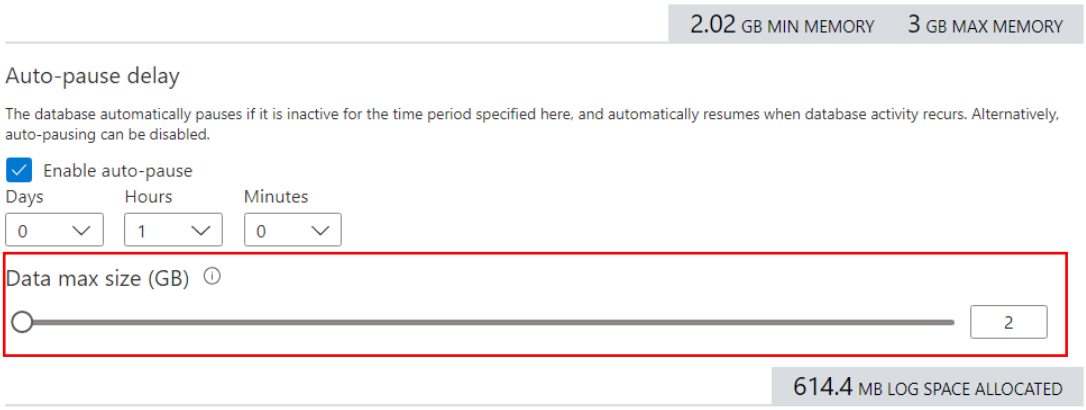
Workload environment
- Development
- Production

Default settings provided for Development workloads. Configurations can be modified as needed.

15-

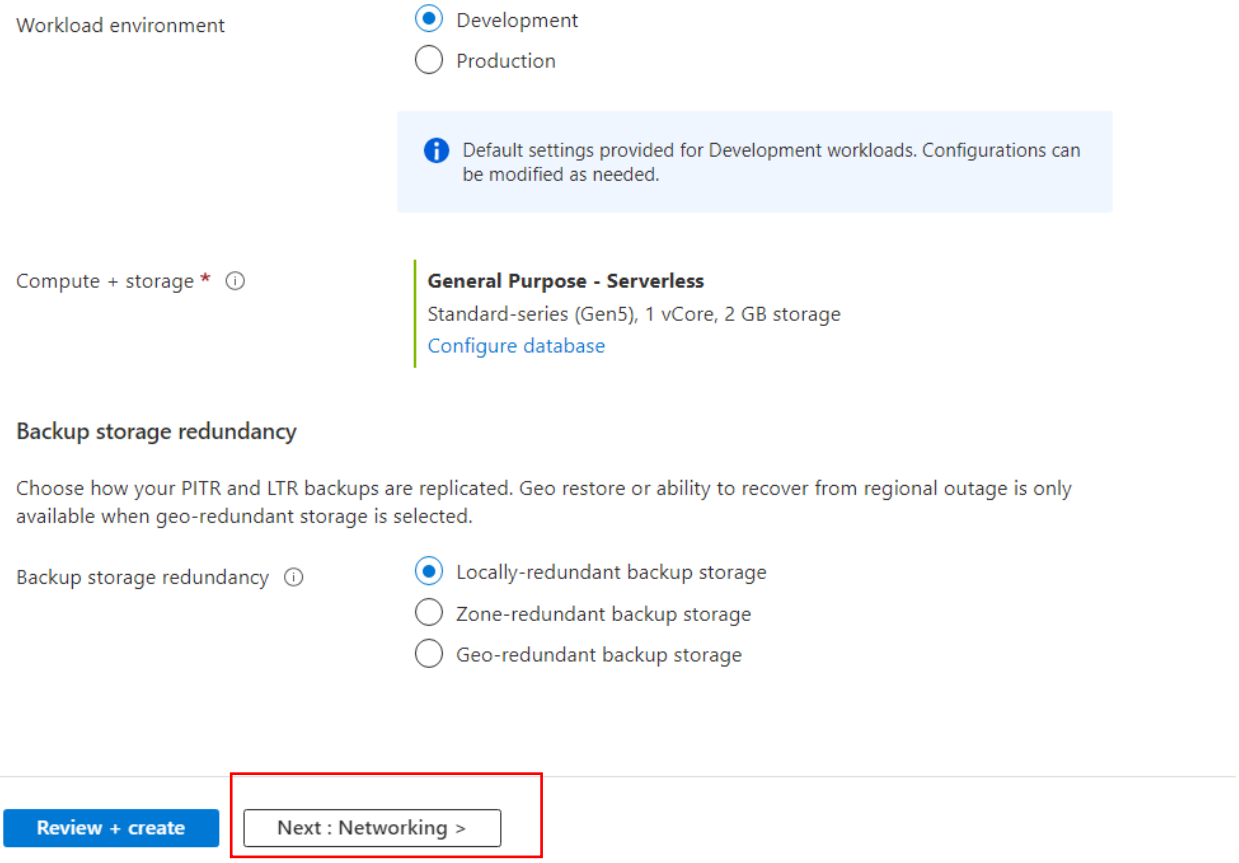
Compute + storage * **General Purpose - Serverless**
Standard-series (Gen5), 1 vCore, 2 GB storage
Configure database

16-



17-

18- After applying our configuration, we will navigate to the networking section



19-

20- In our demo, we will enable the access to our database from a certain virtual network subnet that we will create before continuing the creation of our SQL database. Later, this subnet will be used by the kubernetes cluster.

Microsoft

Basics | Networking | Security | Additional settings | Tags | Review + create

Configure network access and connectivity for your server. The configuration selected below will apply to the selected server 'demo-kube-server' and all databases it manages. Learn more

Network connectivity

Choose an option for configuring connectivity to your server via public endpoint or private endpoint. Choosing no access creates with defaults and you can configure connection method after server creation. Learn more

Connectivity method * ⓘ

- ○ No access
- ○ Public endpoint
- ◉ Private endpoint

Private endpoints

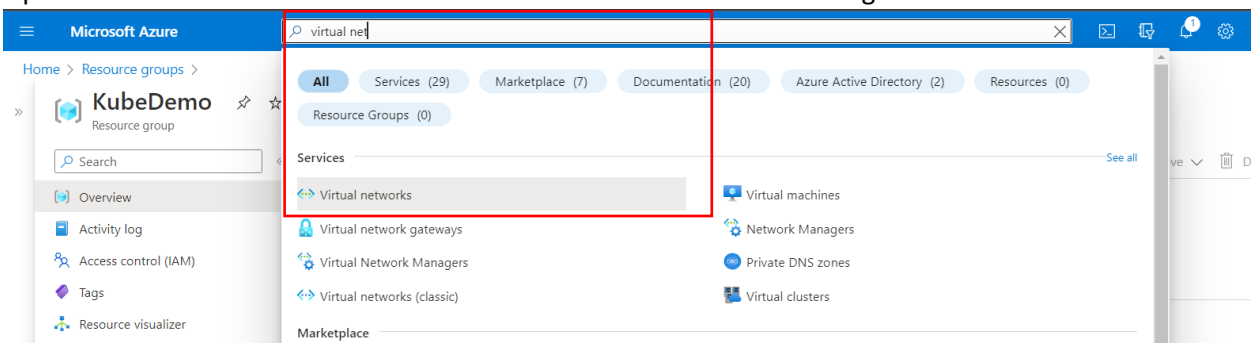
Private endpoint connections are associated with a private IP address within a Virtual Network. The list below shows all the private endpoint connections for this server. Note that private endpoint connections are defined at the server level and they provide access to all databases in the server. Learn more

+ Add private endpoint

Name	Subscription	Resource group	Region	Subnet
Click on add to create private endpoint				

21-

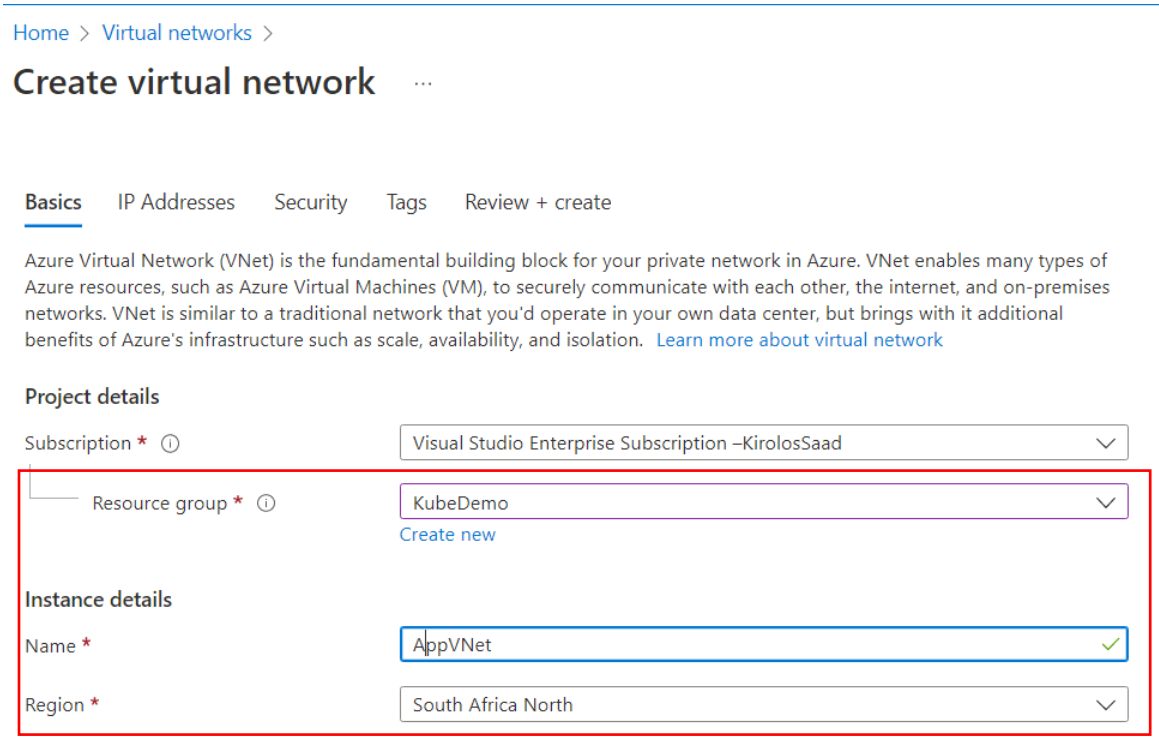
22- Open a new tab and search for virtual network as shown in the next image



23-

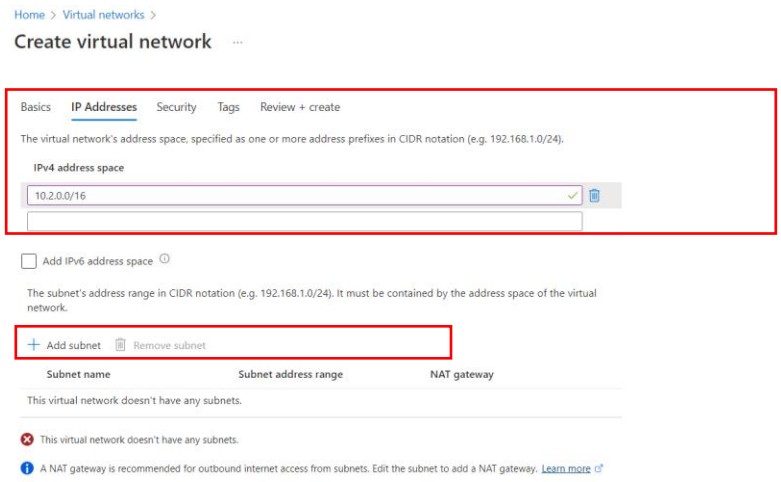
24- Pick the resource group and give it a name

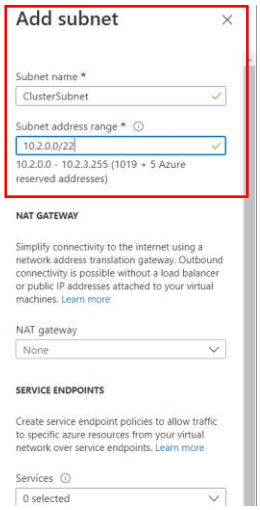
25-



26-

27- Select your IP address space and create a subnet. This subnet should have range of addresses more than 550 (requirement by kubernetes cluster which will use this subnet)

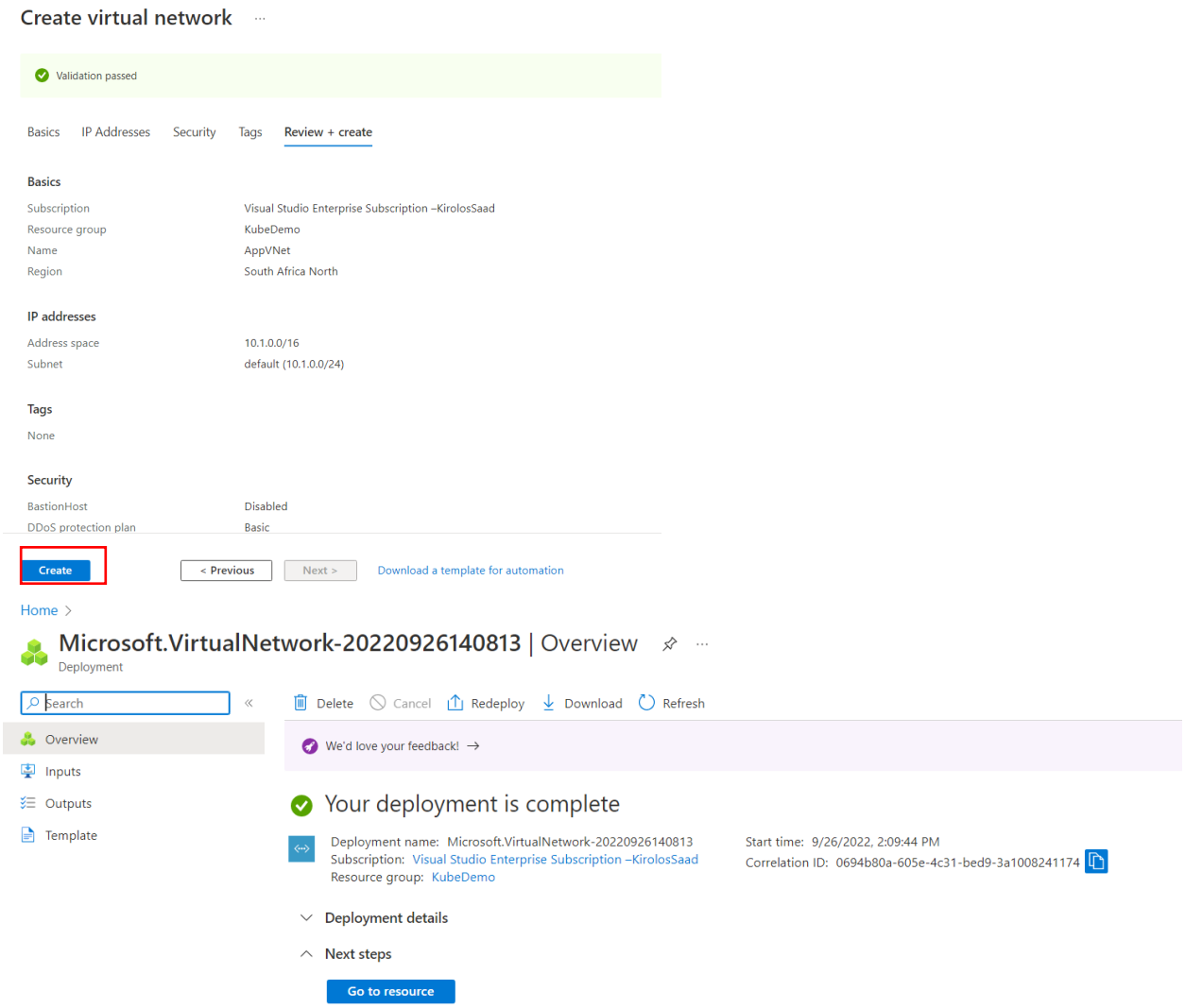




28-

29- After this create your virtual network

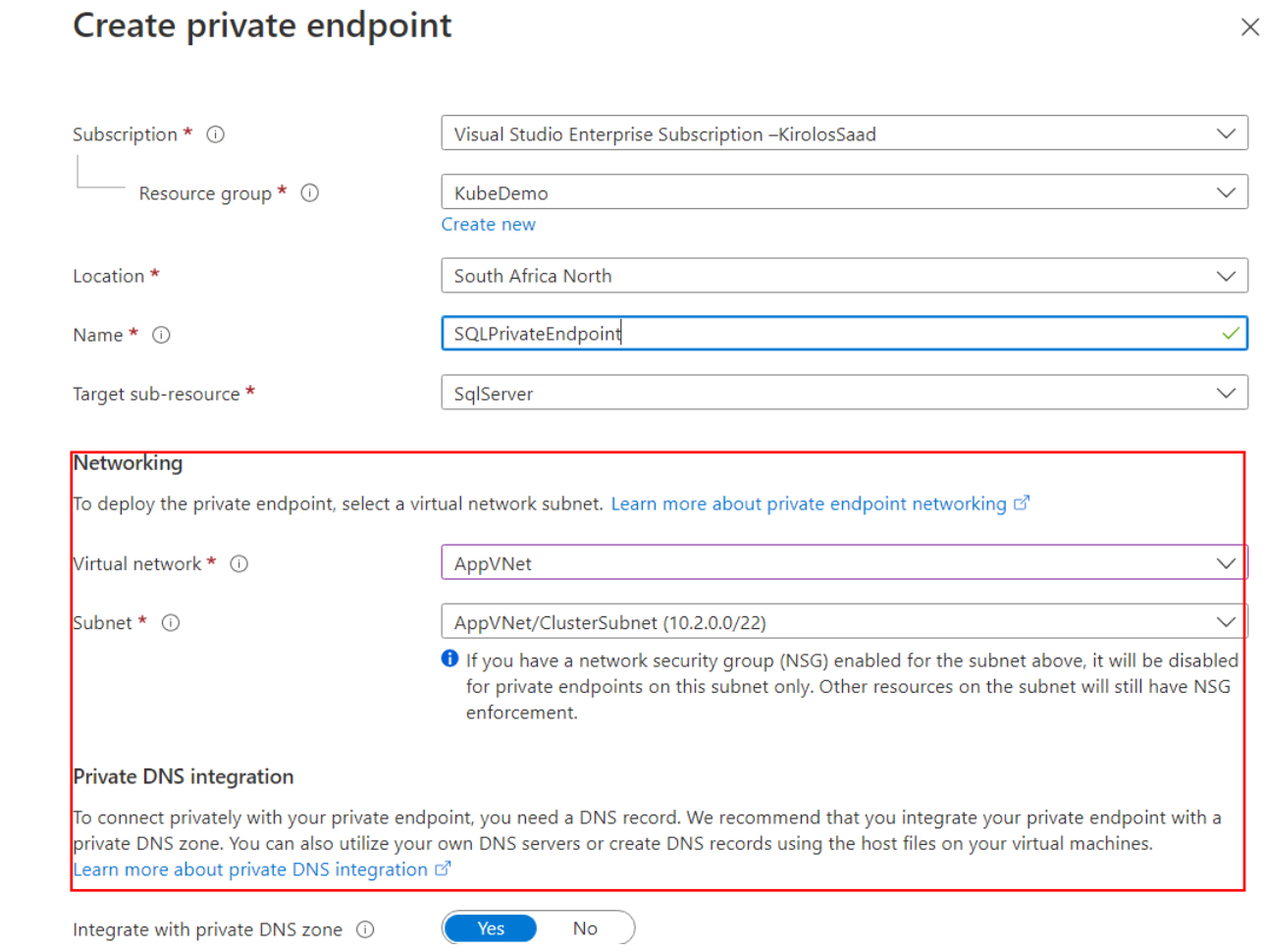
30-



31-

32- Once you are done, navigate to the tab of SQL Azure and pick this private network that we have created

33-


Create private endpoint
Subscription *
Visual Studio Enterprise Subscription –KirolosSaad
Resource group *
KubeDemo
Create new
Location *
South Africa North
Name *
SQLPrivateEndpoint
Target sub-resource *
SqlServer
Networking
To deploy the private endpoint, select a virtual network subnet. Learn more about private endpoint networking
Virtual network *
AppVNet
Subnet *
AppVNet/ClusterSubnet (10.2.0.0/22)
If you have a network security group (NSG) enabled for the subnet above, it will be disabled for private endpoints on this subnet only. Other resources on the subnet will still have NSG enforcement.
Private DNS integration
To connect privately with your private endpoint, you need a DNS record. We recommend that you integrate your private endpoint with a private DNS zone. You can also utilize your own DNS servers or create DNS records using the host files on your virtual machines.
Learn more about private DNS integration
Integrate with private DNS zone
Yes
No

Home > Marketplace >

Create SQL Database ...

Microsoft

Basics | Networking | Security | Additional settings | Tags | Review + create

Configure network access and connectivity for your server. The configuration selected below will apply to the selected server 'demo-kube-server' and all databases it manages. Learn more

Network connectivity

Choose an option for configuring connectivity to your server via public endpoint or private endpoint. Choosing no access creates with defaults and you can configure connection method after server creation. Learn more

Connectivity method * ⓘ

- ○ No access
- ○ Public endpoint
- ◉ Private endpoint

Private endpoints

Private endpoint connections are associated with a private IP address within a Virtual Network. The list below shows all the private endpoint connections for this server. Note that private endpoint connections are defined at the server level and they provide access to all databases in the server. Learn more

+ Add private endpoint

Name	Subscription	Resource group	Region	Subnet
SQLPrivateEndpoint	Visual Studio Enterpris...	KubeDemo	southafricanorth	AppVNet / defaul

Review + create | < Previous | Next : Security >

34-
35-

Home > Marketplace >

Create SQL Database ...

Microsoft

Basics Networking Security Additional settings Tags **Review + create**

Product details

SQL database
by Microsoft
Terms of use | Privacy policy

Estimated cost
Storage cost 0.40 USD / month + Compute cost 0.000194 USD / vCore / second

Terms

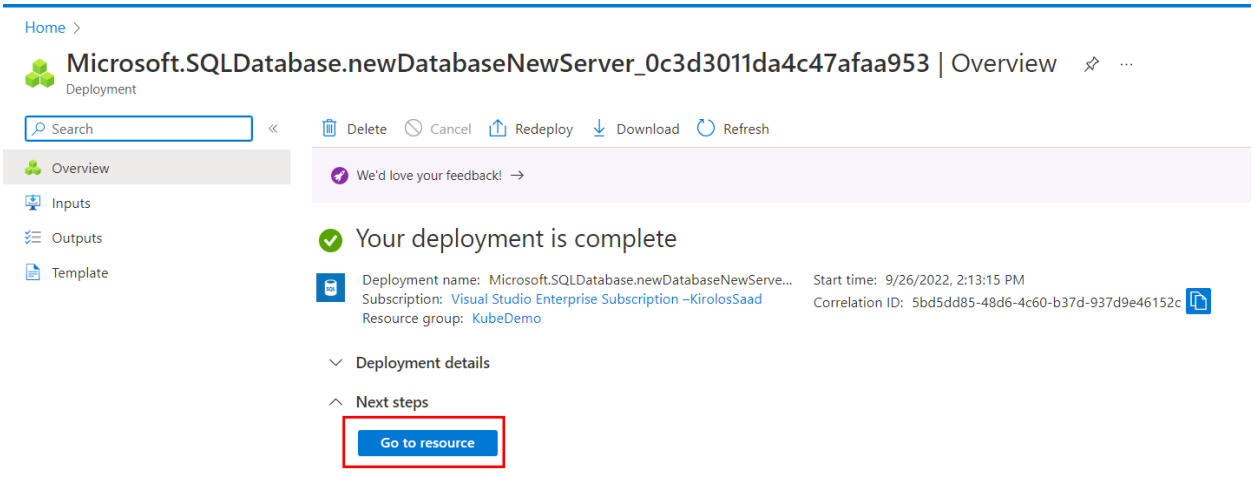
By clicking "Create", I (a) agree to the legal terms and privacy statement(s) associated with the Marketplace offering(s) listed above; (b) authorize Microsoft to bill my current payment method for the fees associated with the offering(s), with the same billing frequency as my Azure subscription; and (c) agree that Microsoft may share my contact, usage and transactional information with the provider(s) of the offering(s) for support, billing and other transactional activities. Microsoft does not provide rights for third-party offerings. For additional details see Azure Marketplace Terms.

Basics

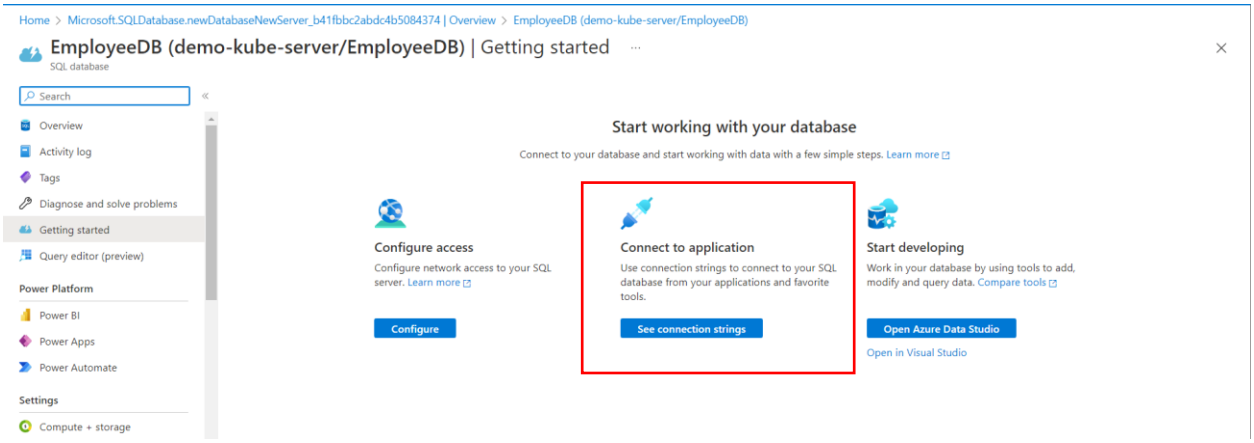
Subscription	Visual Studio Enterprise Subscription –KirolosSaad
Resource group	KubeDemo
Region	South Africa North
Database name	ClientDB
Server	(new) demo-kube-server
Authentication method	Azure Active Directory authentication only

Create < Previous Download a template for automation

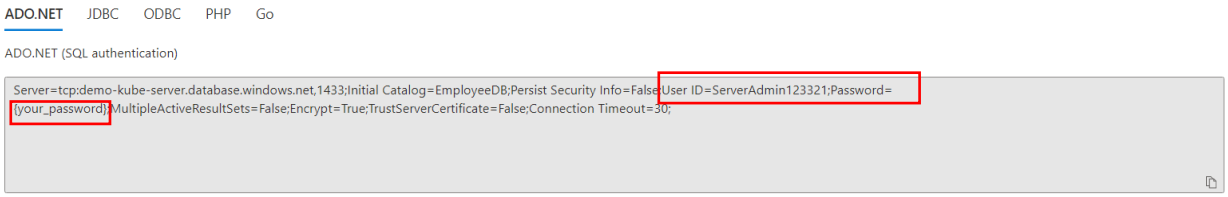
36-



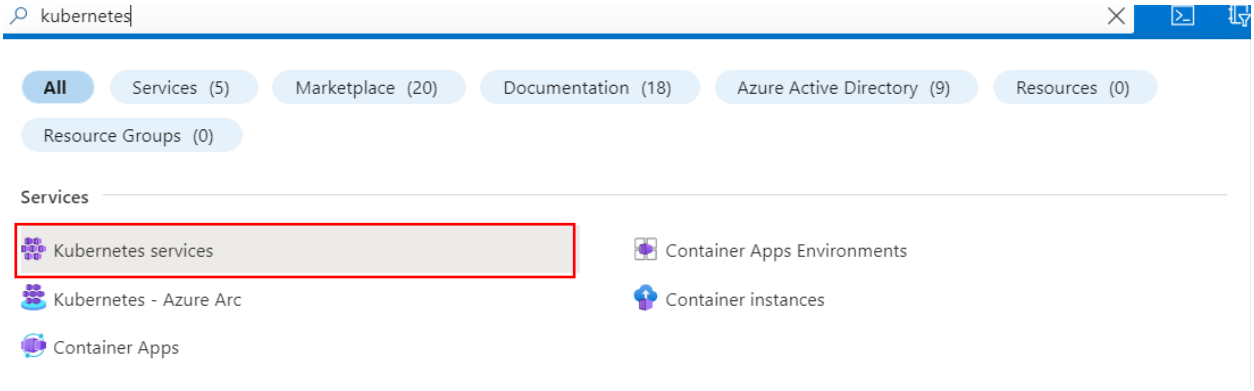
37-

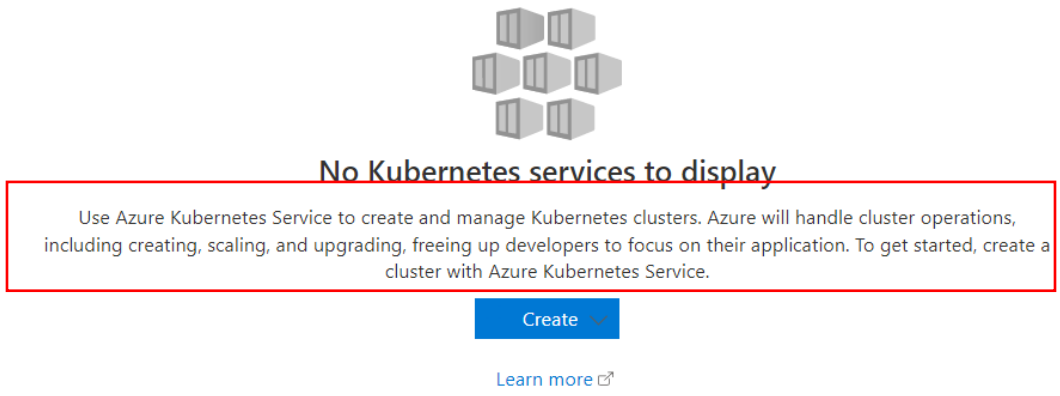
38- 

39- Copy the following connection string as this will be used with our kubernetes deployment section. Kindly note that the following string uses the username and password that we have set before.

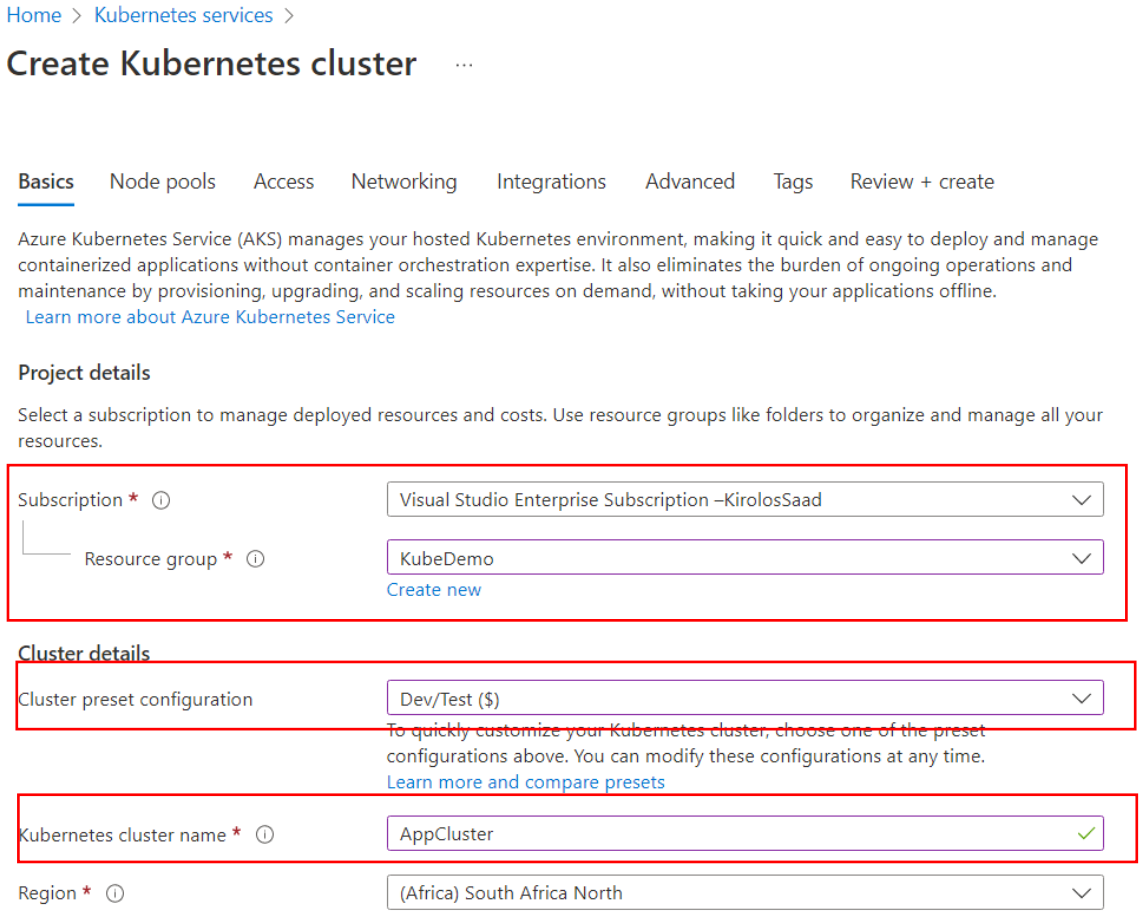
40- 

Creating Azure Kubernetes service

1- 

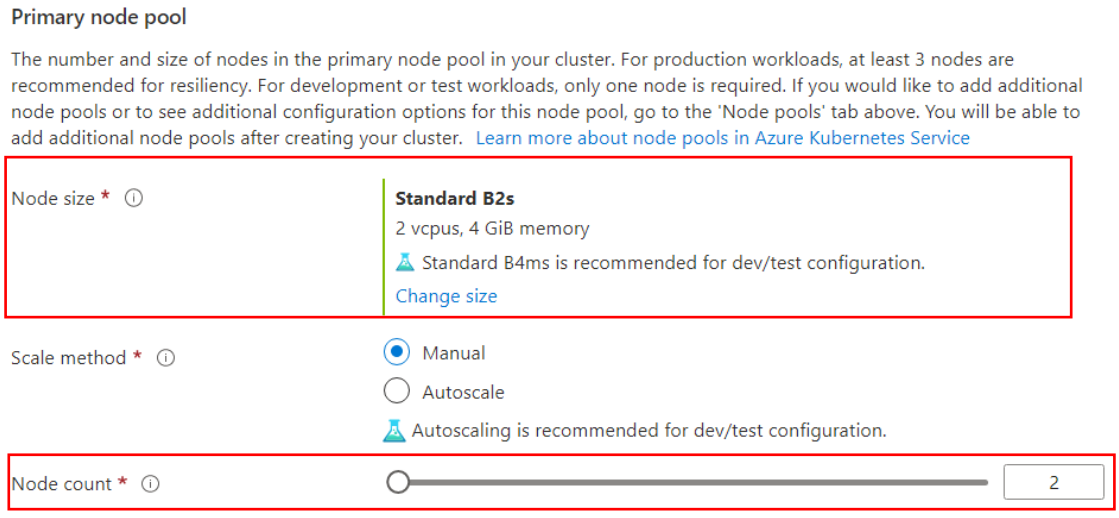
2- 

3- Give your cluster a name, resource group, pick the dev/test configuration as we are working on a demo.



4-

5- Make sure to select these configurations

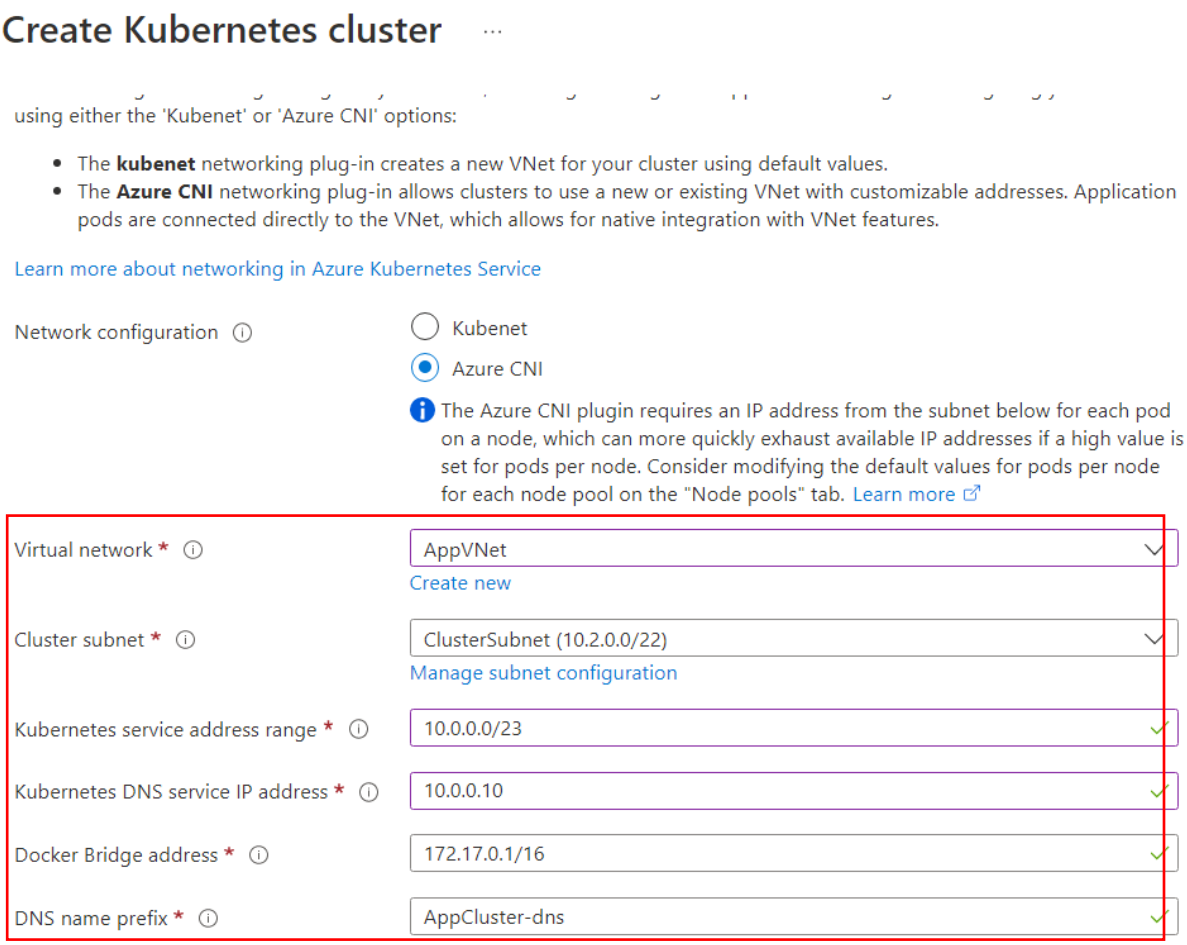


6-

7- Make sure to select your networking to be Azure CNI (to allow our cluster to utilize our virtual network subnet that we have created before)

8-

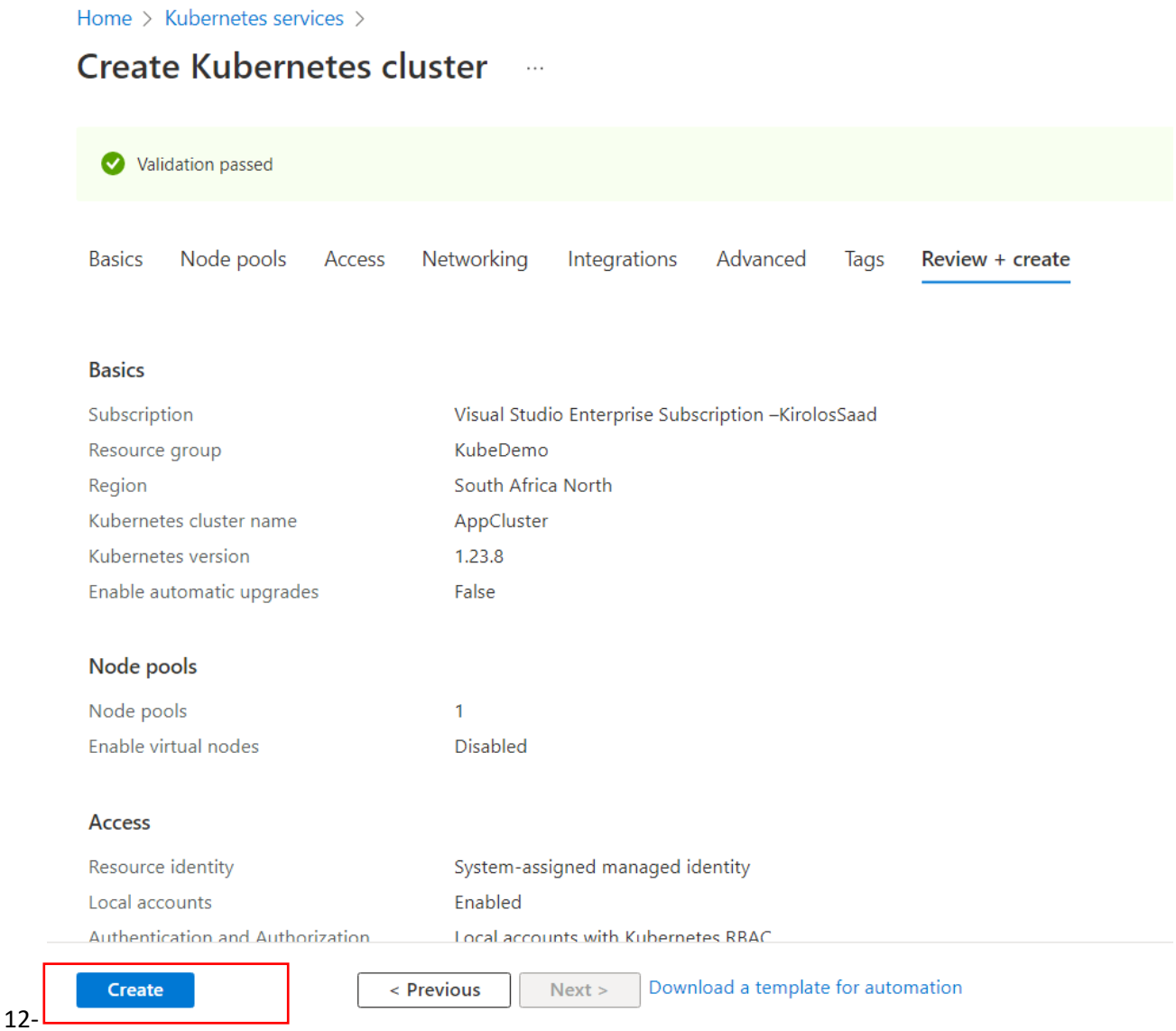
9-



10- Kindly note the following points

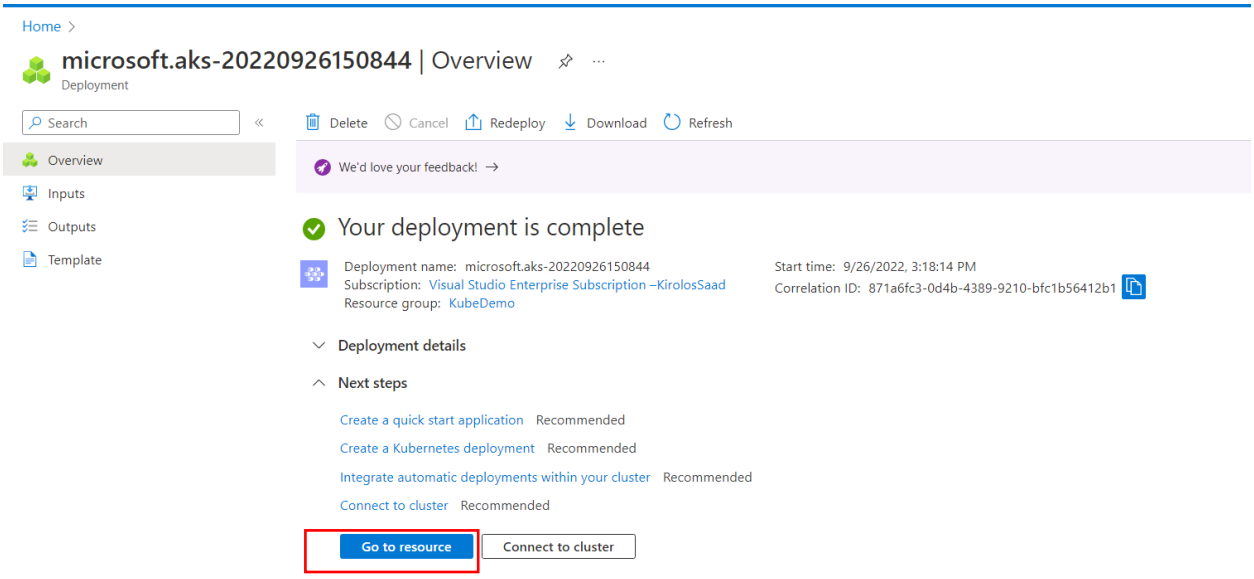
a. **Cluster subnet**: The subnet into which both the nodes and containers in the cluster will be placed (this is the subnet that our database will access a connection from)
b. **Kubernetes service address range:** A CIDR notation IP range from which to assign service cluster Ips
c. **Kubernetes DNS service IP address**: An IP address assigned to the Kubernetes DNS service
d. **DNS name prefix**

11- After these settings, you can create your cluster

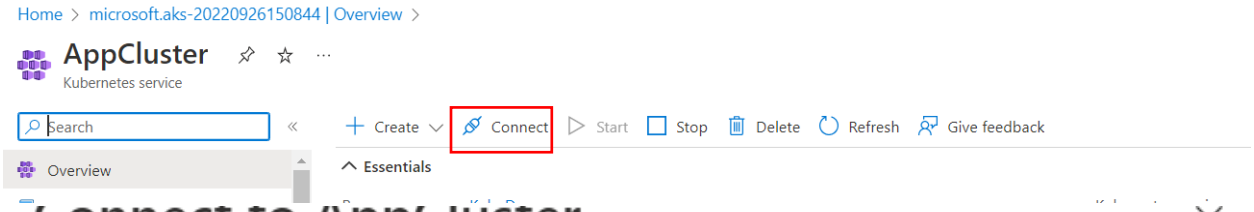

Home > Kubernetes services >
Create Kubernetes cluster ...
Validation passed
Basics Node pools Access Networking Integrations Advanced Tags Review + create
Basics
Subscription Visual Studio Enterprise Subscription –KirolosSaad
Resource group KubeDemo
Region South Africa North
Kubernetes cluster name AppCluster
Kubernetes version 1.23.8
Enable automatic upgrades False
Node pools
Node pools 1
Enable virtual nodes Disabled
Access
Resource identity System-assigned managed identity
Local accounts Enabled
Authentication and Authorization Local accounts with Kubernetes RBAC
Create
< Previous
Next >
Download a template for automation

12-

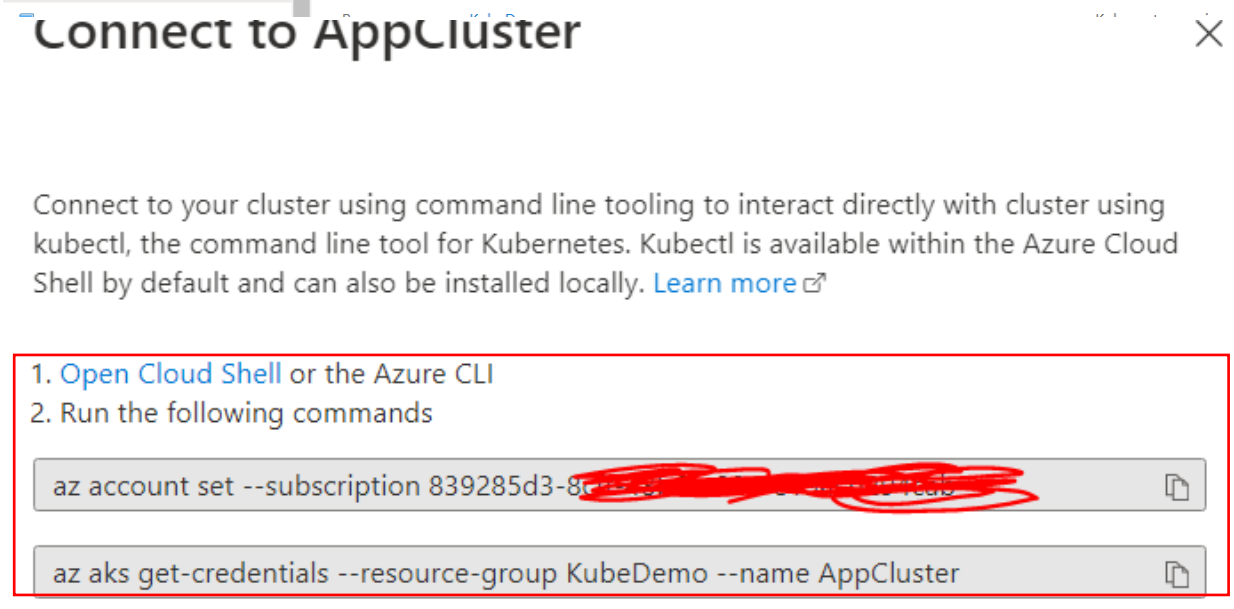
13-



14-



15-



16-

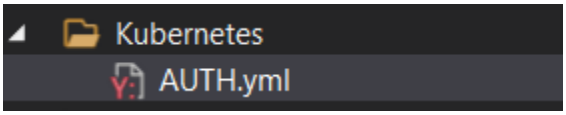
17-

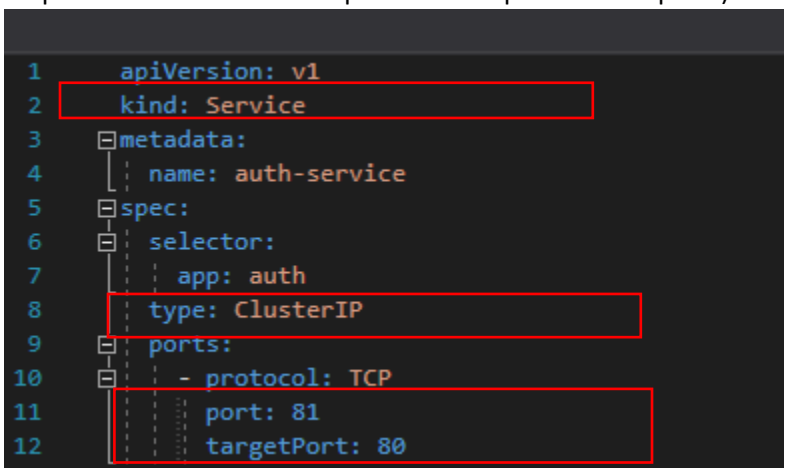
```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service> az account set --subscription 839285d3-8cff
PS C:\Users\kraouf\Desktop\Services\AUTH-Service> az aks get-credentials --resource-group KubeDemo --name AppCluster
```

18- At this point we have configured our kubectl to talk to our AKS.

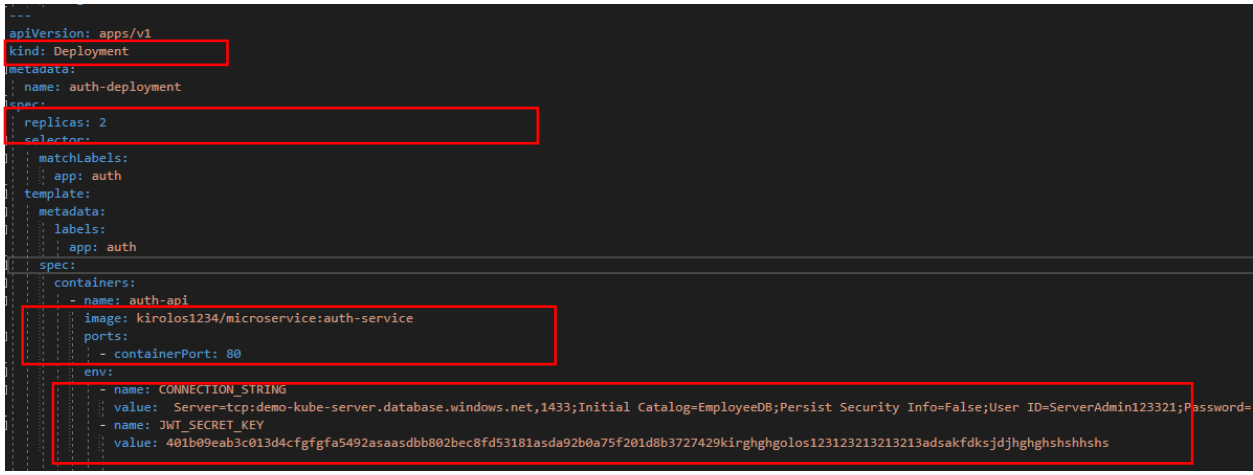
Deployment to our cluster

1- Back to our Visual studio 2019, we will create a file named AUTH.yml. this file will hold 2 kubernetes objects

2- 

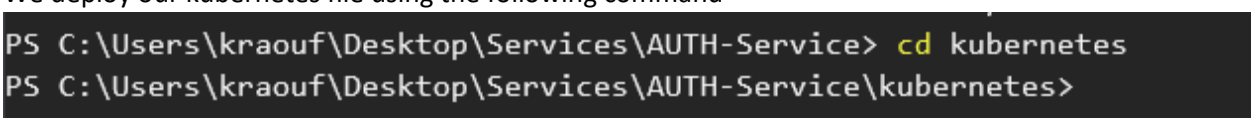
   a. Service object with type ClusterIP.
   b. Will pick port 81 to reach this service (port that accepts requests and forward the request on port 80 which will be the port that the pod will accept on)
   c. 

```yaml
apiVersion: v1
kind: Service
metadata:
  name: auth-service
spec:
  selector:
    app: auth
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 81
      targetPort: 80
```

   d. Deployment object that will controls our pods
   e. We have 2 replications of this pod
   f. We will pass the name of the container image that we push to docker.io
   g. We will allow the container to accept requests on port 80 also.
   h. We will pass 2 environments variables
      i. Connection string that we get from Azure SQL database
      ii. JWT secret key used in encryption process
   i. 

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth-api
          image: kirolos1234/microservice:auth-service
          ports:
            - containerPort: 80
          env:
            - name: CONNECTION_STRING
              value:  Server=tcp:demo-kube-server.database.windows.net,1433;Initial Catalog=EmployeeDB;Persist Security Info=False;User ID=ServerAdmin123321;Password=
            - name: JWT_SECRET_KEY
              value: 401b09eab3c013d4cfgfgfa5492asaasdbb802bec8fd53181asda92b0a75f201d8b3727429kirghghgolos123123213213213adsakfdksjdjhghghshshhshs
```

3- We deploy our kubernetes file using the following command

4- 

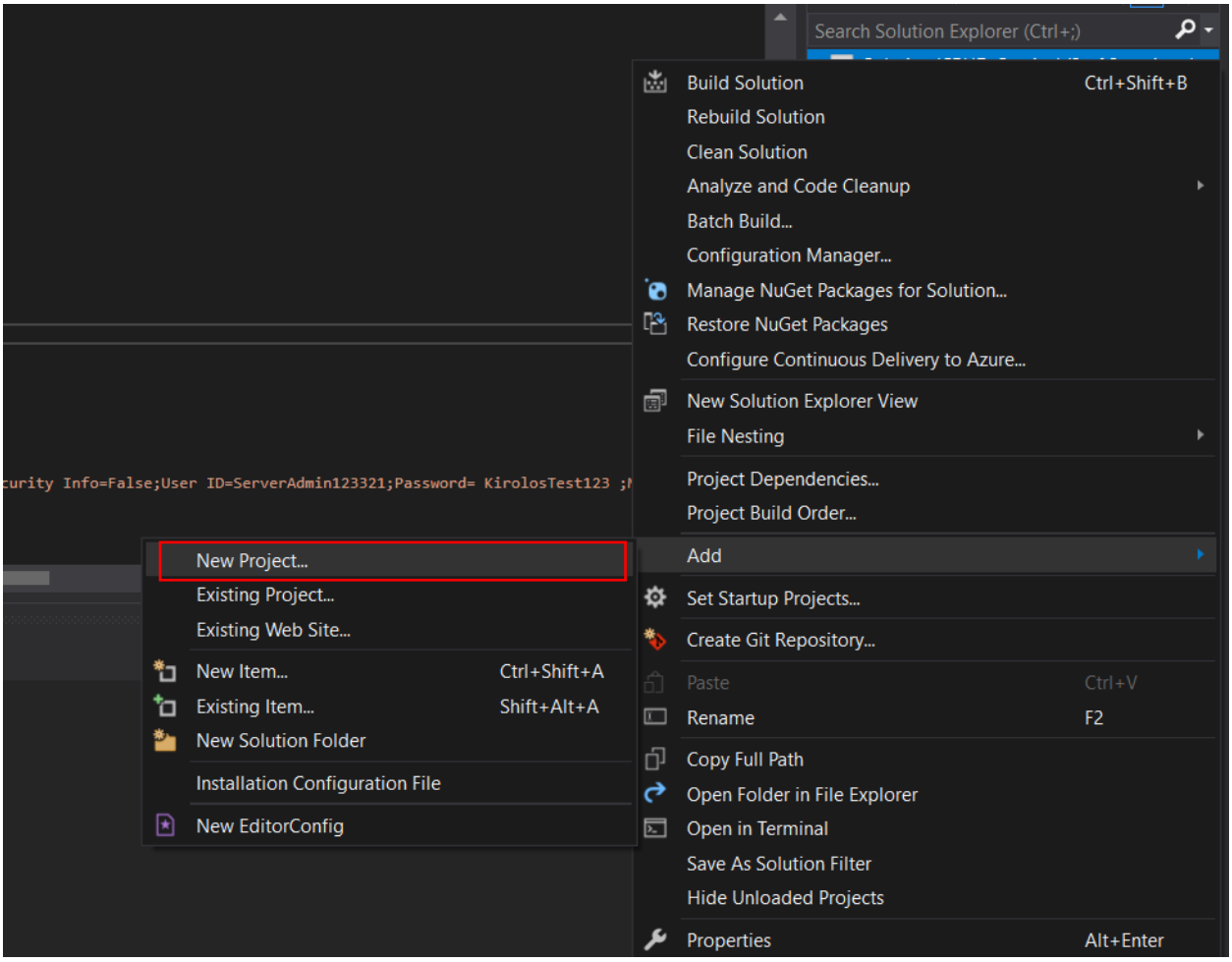
```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service> cd kubernetes
PS C:\Users\kraouf\Desktop\Services\AUTH-Service\kubernetes>
```

5-
```
> kubectl apply -f AUTH.yml
```
6- This is the output
7-
```
service/auth-service created
deployment.apps/auth-deployment created
```
8- To make sure everything is running
9-
```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service\kubernetes> kubectl get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
auth-service   ClusterIP   10.0.210.106   <none>        81/TCP    2m18s
```
10-
```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service\kubernetes> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
auth-deployment-6ddf689c8c-g68v2    1/1     Running   0          111s
auth-deployment-6ddf689c8c-w2x6x    1/1     Running   0          115s
```
11- At this point, our first microservice is configured and deployed on our kubernetes cluster.

CRUD service

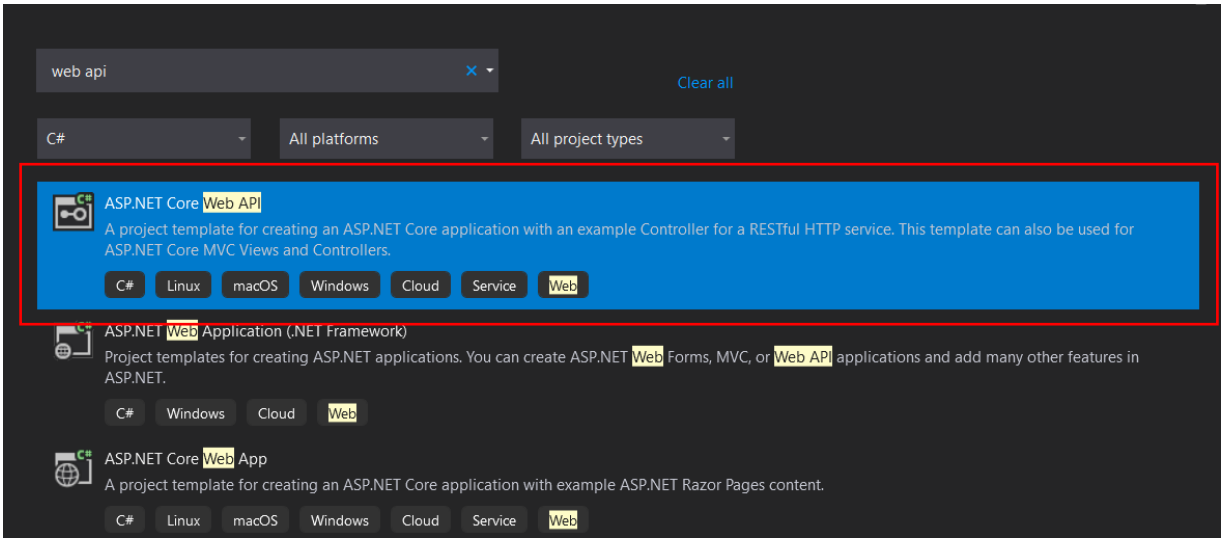
Setting working environment

1- We will right click on our solution to create a new web api project

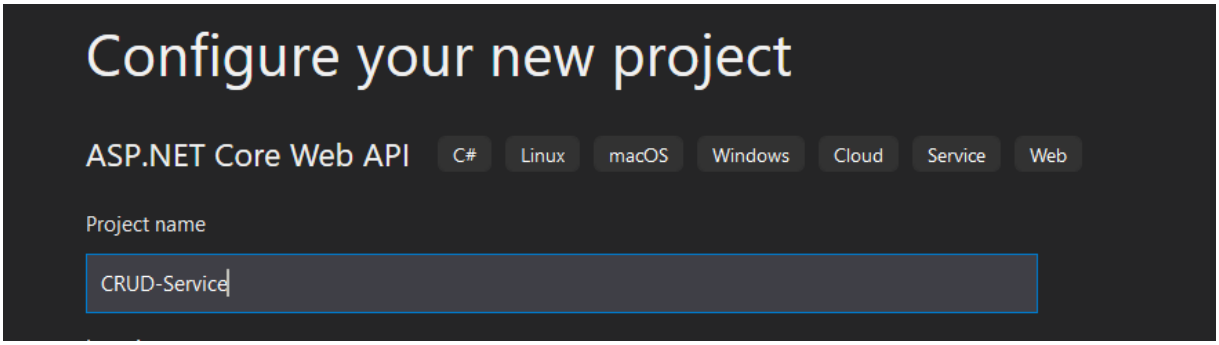


2-

3-

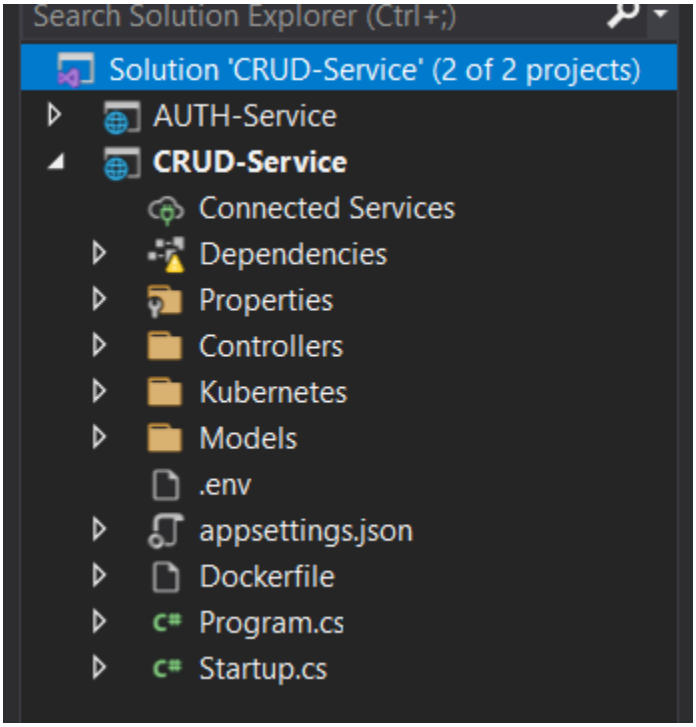


4-

5- 

6- The rest of this section will be same as what we have seen in “Setting working environment” under the authentication service

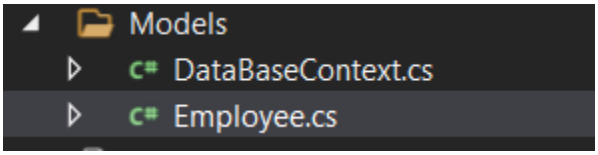
7- This is the final view

8- 

Configuring Entity framework

1- Same as we did before in the authentication service

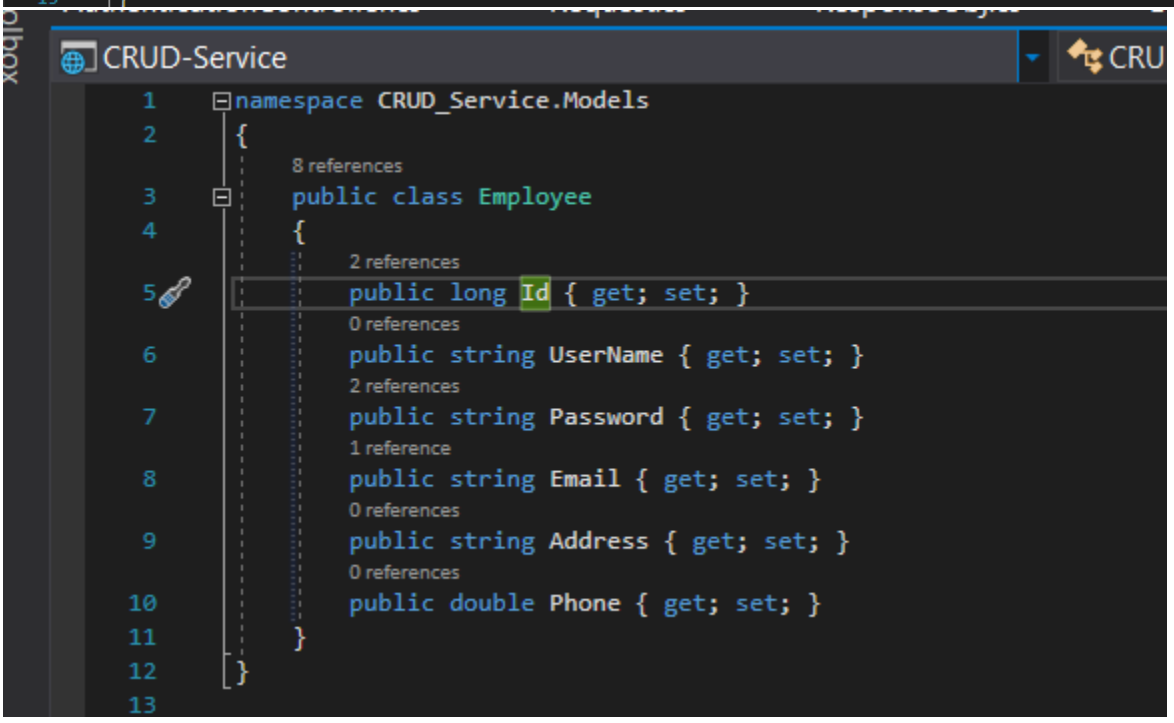
2- The following image shows the model folder



3-

```
using Microsoft.EntityFrameworkCore;


namespace CRUD_Service.Models
{
    6 references
    public class DataBaseContext : DbContext
    {
        0 references
        public DataBaseContext(DbContextOptions<DataBaseContext> options) : base(options)
        {
            Database.EnsureCreated();
        }
        6 references
        public DbSet<Employee> Employees { get; set; }

    }
}
```

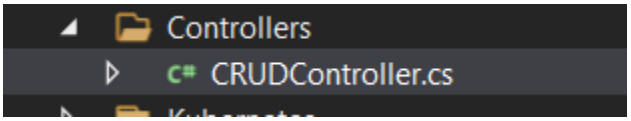
4-



5-

Creating authentication service controller

1- We will create a controller class named “CRUDController.cs”



2-

3- The next image shows the CRUD controller class itself.

a. The constructor accepts a database context object (injected using dependency injection) that represents our database

b.
```
{
    [ApiController]
    [Route("api1")]

    public class CRUDController : ControllerBase
    {
        private readonly DataBaseContext _db;


        public CRUDController(DataBaseContext db)
        {
            _db = db;
        }
```

c. The create endpoint that accept an employee object passed inside the body of the request and process this object

d.
```
// THIS IS FOR C
[HttpPost("emp")]
0 references
public IActionResult createEmployee([FromBody] Employee emp)
{
    _db.Employees.Add(emp);
    _db.SaveChanges();
    return Ok(emp);

}
```

e. The read endpoint that accepts a user ID and search for him

f.
```
}
// THIS IS FOR R
[HttpGet("emp/{id}")]
0 references
public IActionResult getEmployee(long id)
{
    Employee emp = _db.Employees.Find(id);
    emp.Password = null;

    return Ok(emp);
}
```

g. The update endpoints that accept user ID and the new user data to perform update operation. Kindly note we prevent any update action on the password, email, id.

h.

```
// THIS IS FOR U
[HttpPost("emp/{id}")]
0 references
public async Task<IActionResult> updateEmployee(long id, [FromBody] Employee emp)
{
    if (id != emp.Id)
    {
        return BadRequest();
    }
    if (_db.Employees.Find(id) == null)
    {
        return NotFound();
    }
    try
    {
        _db.Entry<Employee>(emp).State = EntityState.Modified;

        _db.Entry<Employee>(emp).Property(p => p.Password).IsModified = false;
        _db.Entry<Employee>(emp).Property(p => p.Email).IsModified = false;
        _db.Entry<Employee>(emp).Property(p => p.Id).IsModified = false;

        await _db.SaveChangesAsync();
    }
    catch (System.Data.Entity.Infrastructure.DbUpdateConcurrencyException)
    {
        if (_db.Employees.Find(id) == null)
        {
            return NotFound();
        }
        throw;
    }
    return NoContent();

}
```

i. The delete endpoint as follows

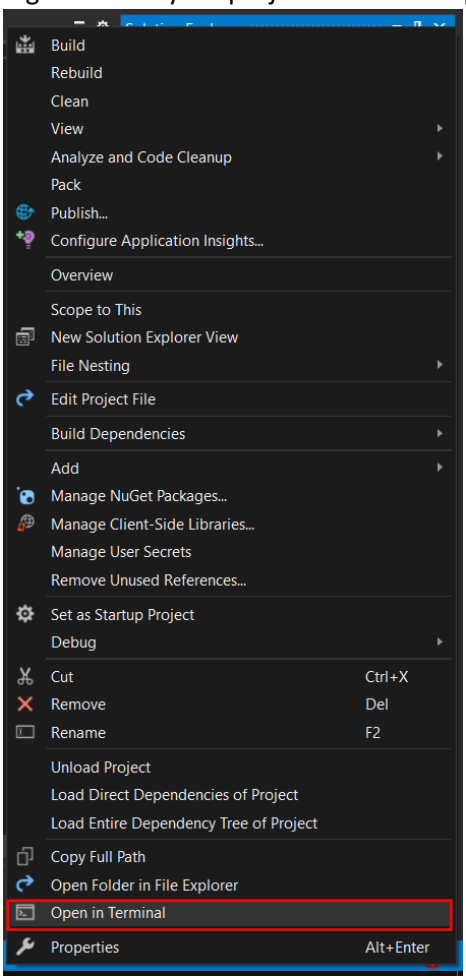
j.

```
71
72          // THIS IS FOR D
73          [HttpDelete("emp/{id}")]
            0 references
74          public async Task<IActionResult> deleteEmployee(long id)
75          {
76              var emp = await _db.Employees.FindAsync(id);
77              if (emp == null)
78              {
79                  return NotFound();
80              }
81              _db.Employees.Remove(emp);
82              await _db.SaveChangesAsync();
83
84              return Ok(emp);
85
86          }
87      }
```

Containerization

1- We will utilize the repo that we have created before to push our image for this microservice with a different image tag

2- Right click on your project and select open in terminal



3-

4- Create the image using the following command

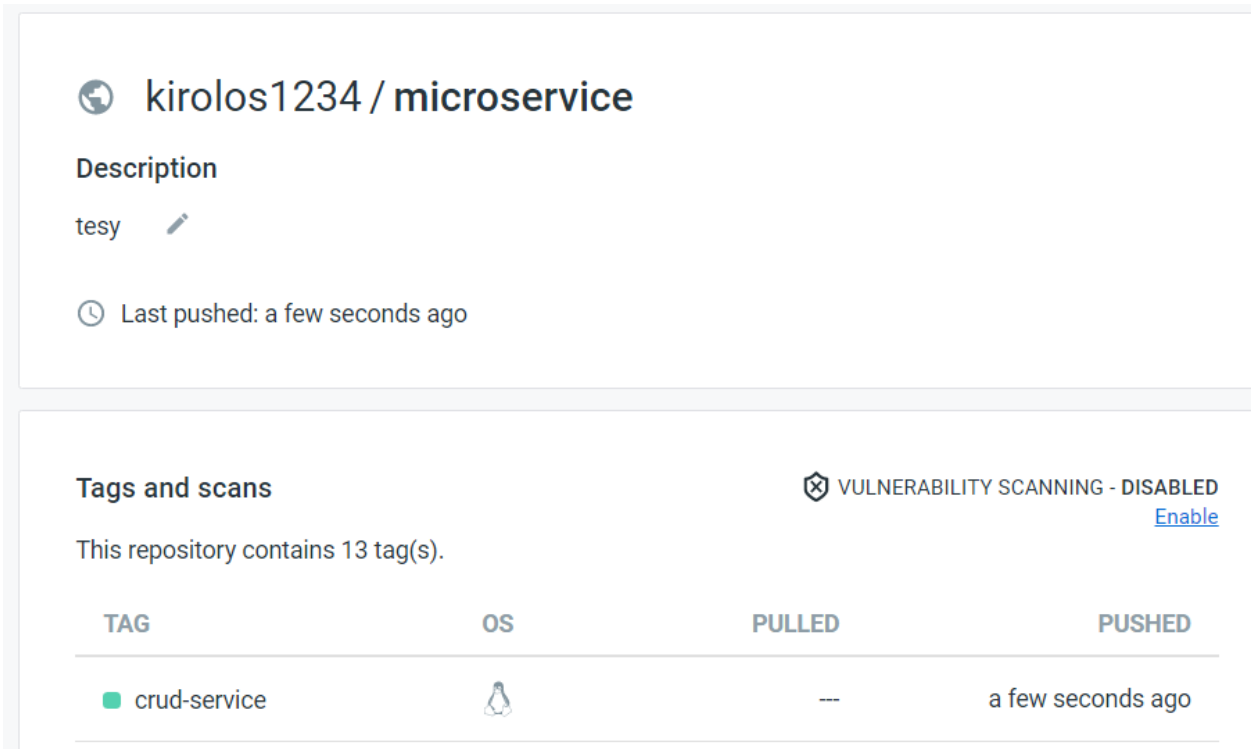
5-
```
** Copyright (c) 2021 Microsoft Corporation
**********************************************************************
PS C:\Users\kraouf\Desktop\Services\CRUD-Service> docker build -t kirolos1234/microservice:crud-service .
```

6- Push image to docker.io repo

7-
```
PS C:\Users\kraouf\Desktop\Services\AUTH-Service> docker push kirolos1234/microservice:auth-service
```

8- **Note**: if you get an error with the previous push command, you need to login to your account from the terminal as shown next

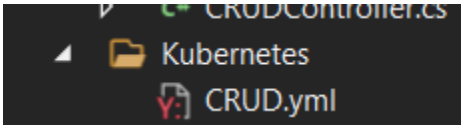
9-
```
PS C:\Users\kraouf\Desktop\Services\CRUD-Service> docker push kirolos1234/microservice:crud-service
```

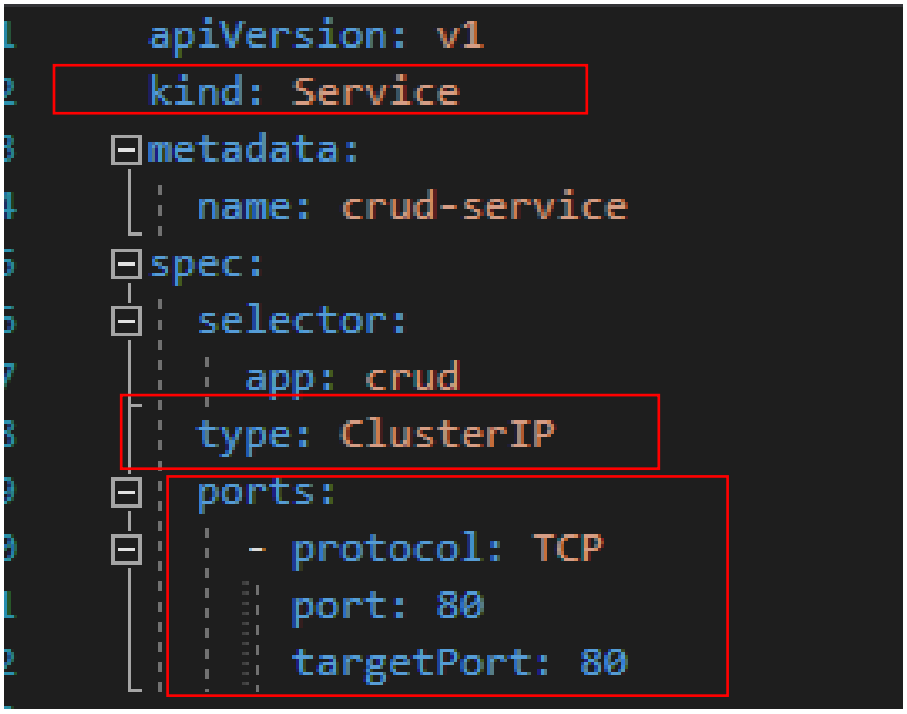

kirolos1234 / microservice
Description
tesy
Last pushed: a few seconds ago
Tags and scans
VULNERABILITY SCANNING - DISABLED
Enable
This repository contains 13 tag(s).
TAG
OS
PULLED
PUSHED
crud-service
---
a few seconds ago

10-

Deployment to our cluster

1- Back to our Visual studio 2019, we will create a file named CRUD.yml. this file will hold 2 kubernetes objects

2- 

   a. Service object with type ClusterIP.
   b. Will pick port 80 to reach this service (port that accepts requests and forward the request on port 80 which will be the port that the pod will accept on)
   c. 

   d. Deployment object that will controls our pods
   e. We have 2 replications of this pod
   f. We will pass the name of the container image that we push to docker.io
   g. We will allow the container to accept requests on port 80 also.
   h. We will pass 1 environments variables
      i. Connection string that we get from Azure SQL database

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: crud-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: crud
  template:
    metadata:
      labels:
        app: crud
    spec:
      containers:
        - name: crud-api
          image: kirolos1234/microservice:crud-service
          ports:
            - containerPort: 80
          env:
            - name: CONNECTION_STRING
              value:  Server=tcp:kube-demo-server.database.windows.net,1433;Initial Catalog=EmployeeDB;Persist Security Info=False;User ID=ServerAdmin123321;P
```

3- We deploy our kubernetes file using the following command

4-
```
PS C:\Users\kraouf\Desktop\Services\CRUD-Service\kubernetes> kubectl apply -f CRUD.yml
```

5- To make sure everything is running

6-
```
PS C:\Users\kraouf\Desktop\Services\CRUD-Service\kubernetes> kubectl get services
NAME                TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
auth-service        ClusterIP      10.0.1.48     <none>          81/TCP         33m
crud-service        ClusterIP      10.0.1.167    <none>          80/TCP         33m
kubernetes          ClusterIP      10.0.0.1      <none>          443/TCP        46m
```

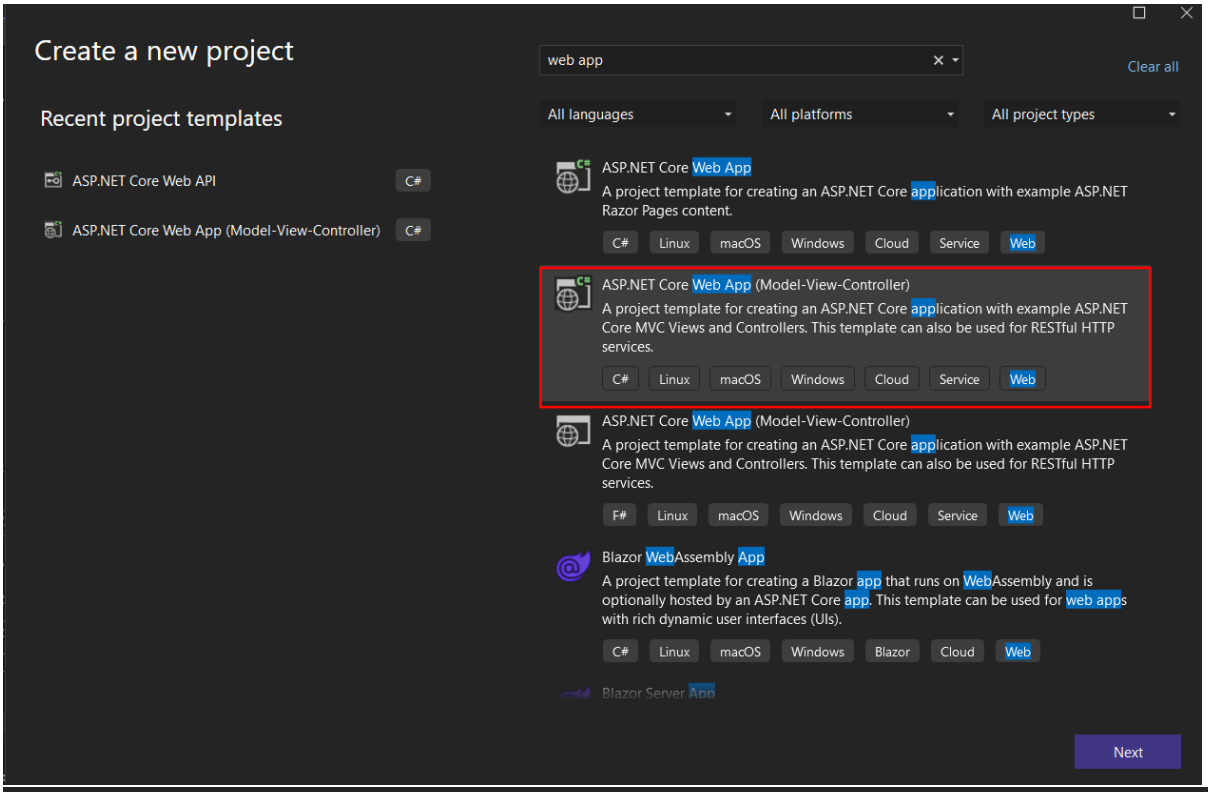
7-
```
PS C:\Users\kraouf\Desktop\Services\CRUD-Service\kubernetes> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
auth-deployment-9bb9bff-4h9tm       1/1     Running   0          33m
auth-deployment-9bb9bff-qkg9h       1/1     Running   0          33m
crud-deployment-748f469857-clfc6    1/1     Running   0          2m13s
crud-deployment-748f469857-ggn4j    1/1     Running   0          2m2s
```

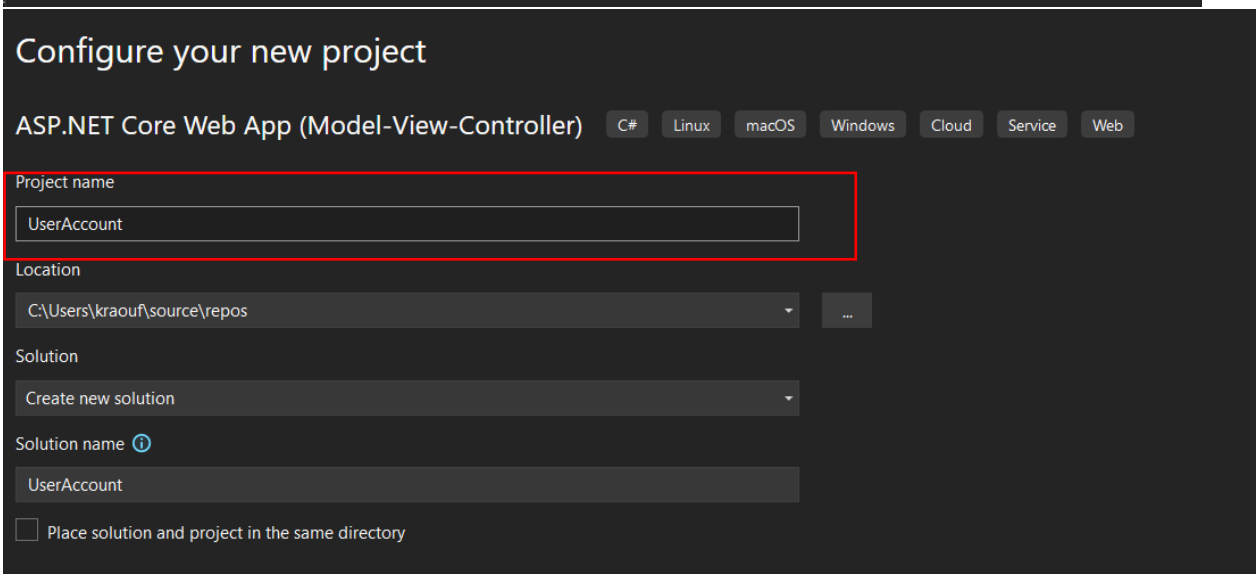
8- At this point, our second microservice is configured and deployed on our kubernetes cluster.

MVC Project

Setting working environment

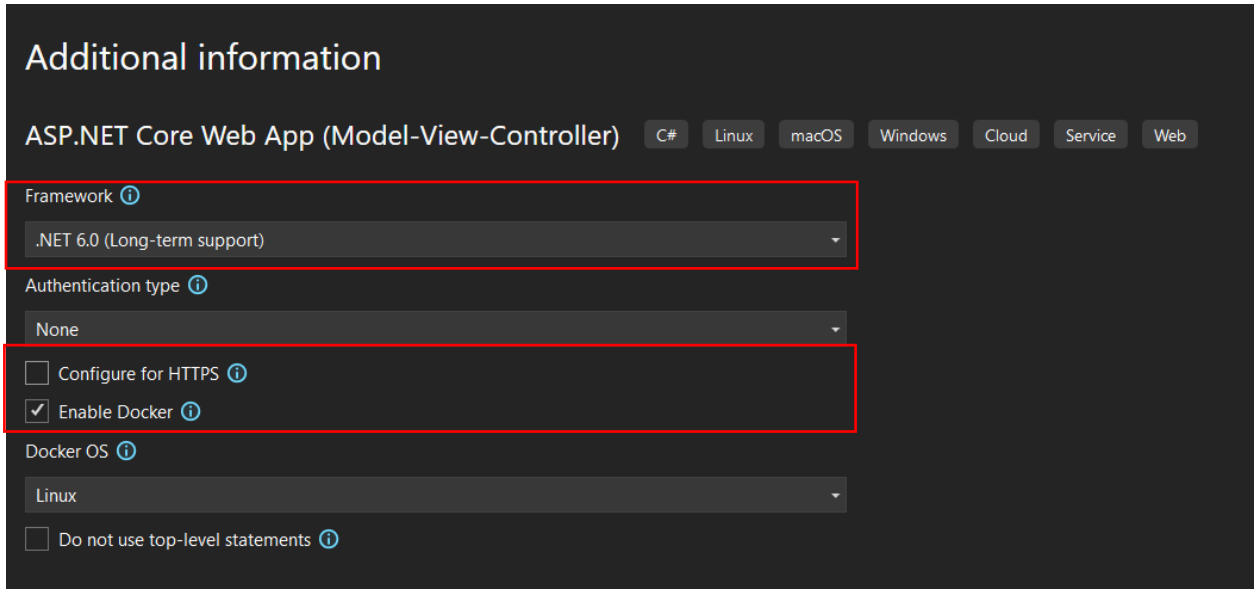
1- Create a new MVC project as shown next

2- 

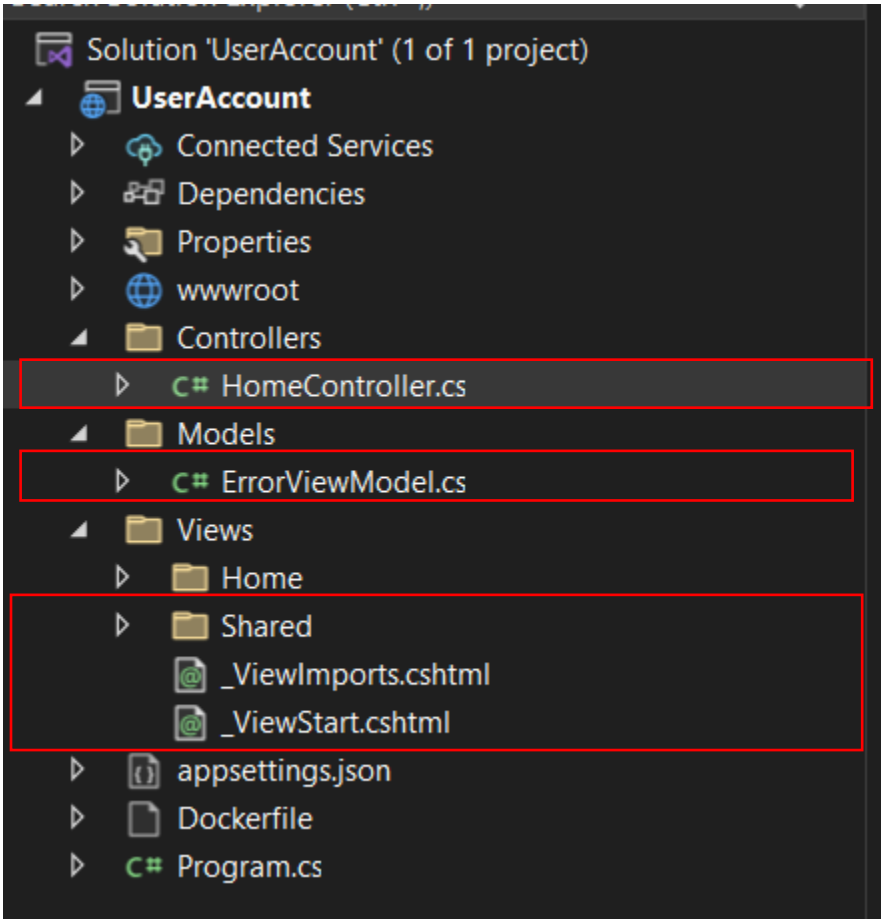
3- 

4- Don’t’ forget to pick these options

5-



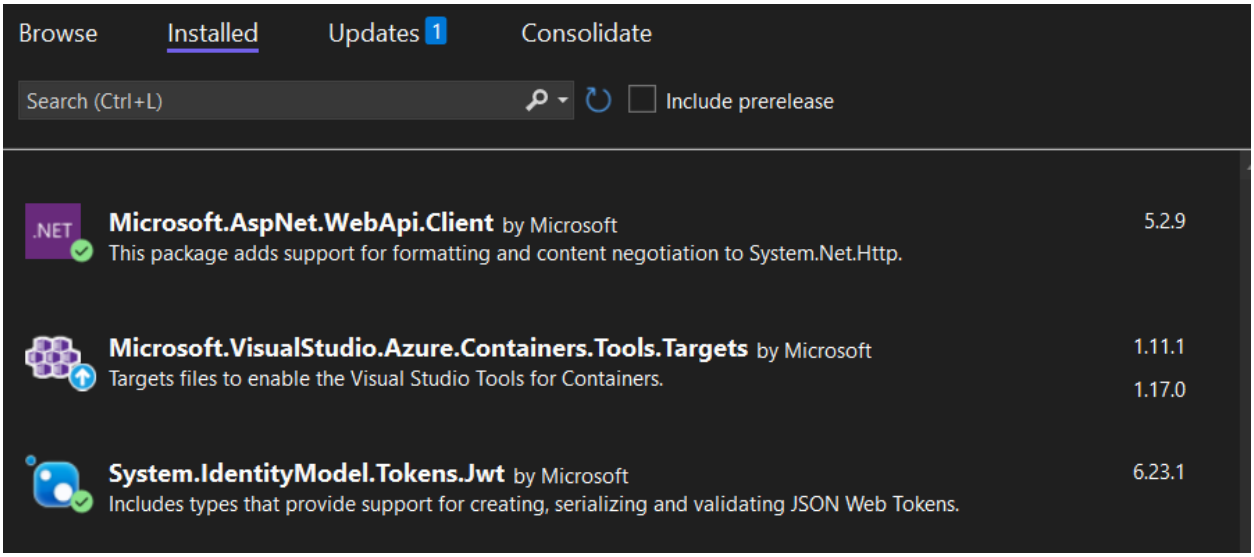
6- Remove the following items as we won't use them for this demo



7-

8- One final step, is to download some libraries that we will use as we did before.

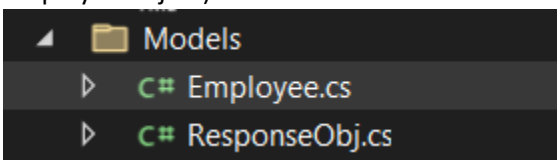
9-


Browse
Installed
Updates 1
Consolidate
Search (Ctrl+L)
Include prerelease
Microsoft.AspNet.WebApi.Client by Microsoft
This package adds support for formatting and content negotiation to System.Net.Http.
5.2.9
Microsoft.VisualStudio.Azure.Containers.Tools.Targets by Microsoft
Targets files to enable the Visual Studio Tools for Containers.
1.11.1
1.17.0
System.IdentityModel.Tokens.Jwt by Microsoft
Includes types that provide support for creating, serializing and validating JSON Web Tokens.
6.23.1

Creating the models

1- As our MVC will deal mainly with 2 responses (one from the authentication that returns to us a JWT in case our user is authenticated and another from the CRUD microservice that returns an employee object)

2-



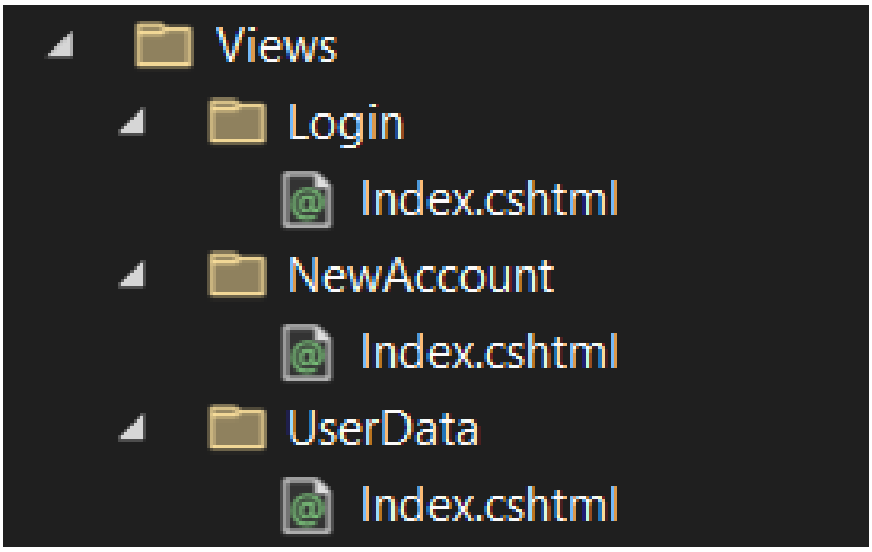
3-

```
namespace UserAccount.Models
{
    14 references
    public class Employee
    {
        3 references
        public long Id { get; set; }
        4 references
        public string UserName { get; set; }
        2 references
        public string Password { get; set; }
        3 references
        public string Email { get; set; }
        3 references
        public string Address { get; set; }
        3 references
        public double Phone { get; set; }
    }
}
```

4-

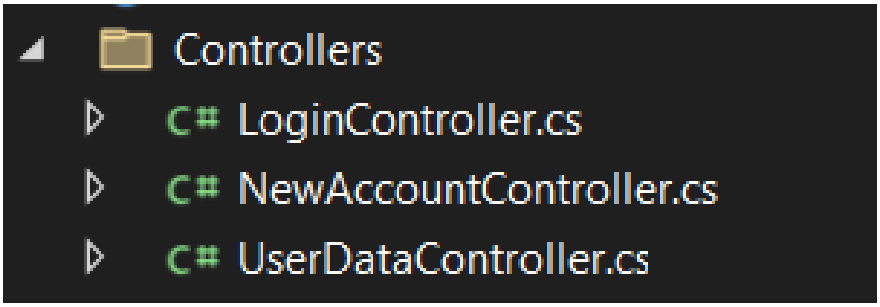
```
namespace UserAccount.Models
{
    2 references
    public class ResponseObj
    {
        2 references
        public string jwt { get; set; }
    }
}
```

Creating the views

1- We will have 3 views (3 folders)
   a. Login page
   b. Create a new user page
   c. Update, Read, Delete user data in the same page
2- I won't dig into details as these files are simple to understand (examine the code for these files)
3- 

Creating the controllers

1- For each view, we will have a controller



2-

3- We will pick the login controller

   a. This file contains 2 functions

      i. One to render the view of the login page that the user will use to enter his credentials

      ii.

```
public class LoginController : Controller
{
    string BASE_URL = Environment.GetEnvironmentVariable("BACKEND_STRING_AUTH");

    0 references
    public ActionResult Index()
    {
        return View();

    }
```

      iii. Another function that will make an HTTP request to the authentication service to check if the given email and password are correct or not. If the data is correct, the user will get a JWT as a response that will be stored inside session storage and the request will be forwarded to the second page with the payload from the JWT (UserData view)

      iv.

```
0 references
public async Task<ActionResult> Authenticate(Employee employeeData)
{
    using (var client = new HttpClient())
    {
        var uri = new Uri(BASE_URL + "/api2/login");

        StringContent jsonContent = new(System.Text.Json.JsonSerializer.Serialize(employeeData), Encoding.UTF8, "application/json");

        HttpResponseMessage response = await client.PostAsync(uri, jsonContent);


        if (response.StatusCode == System.Net.HttpStatusCode.NoContent)
        {
            return RedirectToAction("Index", "Login");

        }
        ResponseObj token = await response.Content.ReadAsAsync<ResponseObj>(new[] { new JsonMediaTypeFormatter() });

        HttpContext.Session.SetString("Token", token.jwt);


       var handler = new JwtSecurityTokenHandler();
        var tokenObj = handler.ReadJwtToken(token.jwt);

        var payLoad = tokenObj.Payload.FirstOrDefault().Value;

        return RedirectToAction("Index", "UserData", new { id = payLoad });
    }
```

4- UserData controller

   a. This file will contain the methods that make requests to the CRUD microservice.

   b. For simplicity, I will pick one method (the Read operation) to explain it. However, the rest of the methods are similar

   c. The read method will check the stored JWT and extract the user ID in case the stored JWT is a valid one.

   d. A Read operation is sent to the CRUD microservice with the user id to retrieve user's data

e. Finally, the data are being passed to the view to embed this data and returns the HTML page to the user

f.

```
// GET: UserData
0 references
public async Task<ActionResult> Index(string id)
{
    var jwtToken = HttpContext.Session.GetString("Token");

    if (jwtToken == null)
    {
        return RedirectToAction("Index", "Login");

    }

    var handler = new JwtSecurityTokenHandler();
    var tokenObj = handler.ReadJwtToken(jwtToken);

    if (tokenObj == null)
    {
        return RedirectToAction("Index", "Login");

    }
    var payloadID = tokenObj.Payload.FirstOrDefault().Value.ToString();

    if (payloadID == null && payloadID == "")
    {
        return RedirectToAction("Index", "Login");

    }

    using (var client = new HttpClient())
    {
        var uri = new Uri(BASE_URL + "api1/emp/" + payloadID);

        HttpResponseMessage response = await client.GetAsync(uri);

        if(response.StatusCode == System.Net.HttpStatusCode.NoContent)
        {
            return RedirectToAction("Index", "Login");

        }
        Employee emp =  await response.Content.ReadAsAsync<Employee>(new[] { new JsonMediaTypeFormatter() });
        return View(emp);
    }

}
```

5- New Account controller

a. This one contains 2 functions

i. One to render the page of creating a new account

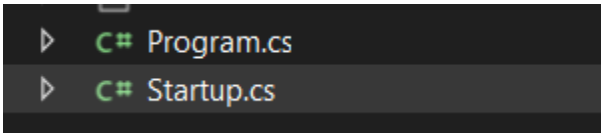
ii. A function that posts the data that the user enters in the screen to the CRUD microservice to create a new account

b.

```
0 references
public class NewAccountController : Controller
{
    string BASE_URL = Environment.GetEnvironmentVariable("BACKEND_STRING_CRUD");


    // GET: UserData
    0 references
    public ActionResult Index()
    {
        return View();
    }
    0 references
    public ActionResult Cancel()
    {
        return RedirectToAction("Index", "Login");
    }
    // POST: HomeControlhhler1/Create
    [HttpPost]
    0 references
    public async Task<ActionResult> Create(Employee employeeData)
    {
        using (var client = new HttpClient())
        {
            var uri = new Uri(BASE_URL + "api1/emp");

            StringContent jsonContent = new(JsonSerializer.Serialize(employeeData),Encoding.UTF8, "application/json");

            HttpResponseMessage response = await client.PostAsync(uri,jsonContent);

        }

        return RedirectToAction("Index", "Login");

    }
```

Extra configurations

1- Open the startup file



2-

3- Configuring the start point to be login controller/Index action

a.

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
0 references
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseSession();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Login}/{action=Index}/{id?}");
    });
}
```

4- Configuring the session storage to be used inside our project and specify a expire time for the session (forcing the user to re-login again after this time)

   a. The value of this time will be passed using environment variable in the kubernetes deployment section

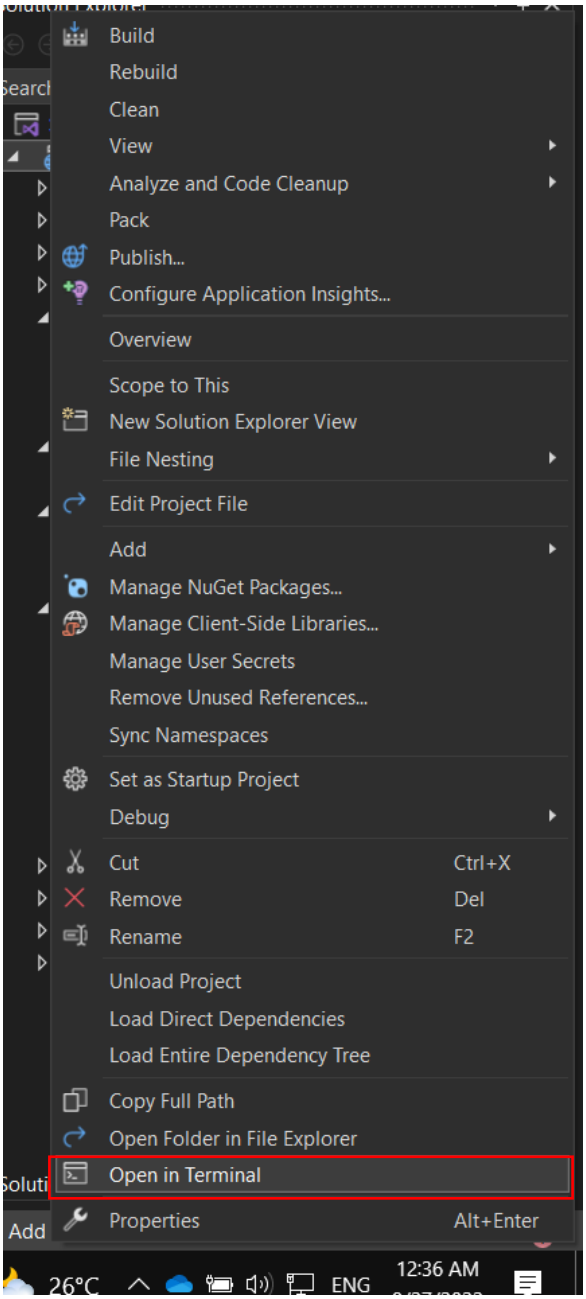
b.

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
0 references
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseSession();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Login}/{action=Index}/{id?}");
    });
}
```

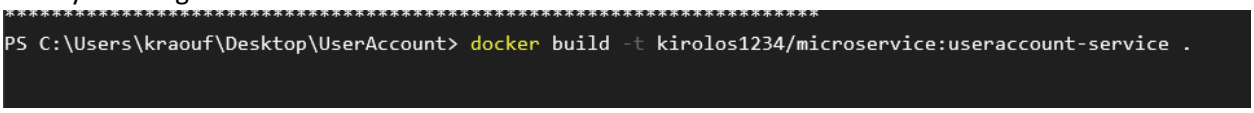
c.

```
// This method gets called by the runtime. Use this method to add services to the container.
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddDistributedMemoryCache();

    services.AddSession(options =>
    {
        options.IdleTimeout = TimeSpan.FromMinutes(int.Parse(System.Environment.GetEnvironmentVariable("JWT_LIFETIME_MIN")));
        options.Cookie.HttpOnly = true;
        options.Cookie.IsEssential = true;
    });
}
```

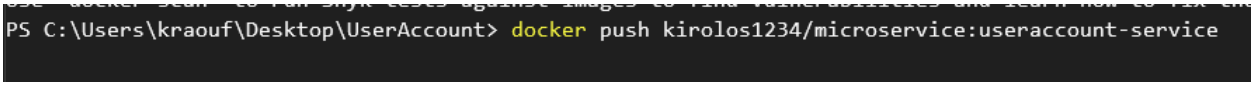
Containerization

1- 

2- Build your image

3- 

4- Push image

5- 

6- Make sure you image is pushed

kirolos1234 / **microservice**

Description

tesy

Last pushed: a few seconds ago

Tags and scans

VULNERABILITY SCANNING - **DISABLED**
Enable

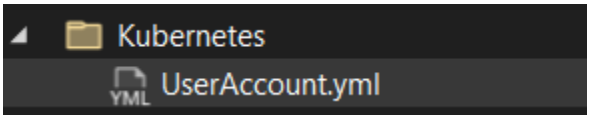
This repository contains 15 tag(s).

TAG	OS	PULLED	PUSHED
useraccount-service		---	a few seconds ago

7-

Deployment to our cluster

1- Back to our Visual studio we will create a file named UserAccount.yml. this file will hold 2 kubernetes objects

2- 

   a. Service object with type load balancer to be accessible using an external ip.
   b. Will pick port 80 to reach this service (port that accepts requests and forward the request on port 80 which will be the port that the pod will accept on)
   c.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: useraccount-service
spec:
  selector:
    app: useraccount
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
```

   d. Deployment object that will controls our pods
   e. We have 2 replications of this pod
   f. We will pass the name of the container image that we push to docker.io
   g. We will allow the container to accept requests on port 80 also.
   h. We will pass 3 environments variables
      i. The session expiration time in minute
      ii. The backend URLs of both microservices that we have in our system. Kubernetes by default has a DNS during the creation of the cluster itself. This is very efficient as we could access the service objects (of type ClusterIP) using this pattern:
         1. **http://{SERVICE_NAME}.{NAMESPACE}.svc.cluster.local:{PORT_NUMBER}**
         2. By default, we are deploying our objects in the **default** namespace. However, you could create another namespace and deploy you objects in it (read more about namespaces)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: useraccount-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: useraccount
  template:
    metadata:
      labels:
        app: useraccount
    spec:
      containers:
        - name: useraccount-api
          image: kirolos1234/microservice:useraccount-service
          ports:
            - containerPort: 80
          env:
            - name: BACKEND_STRING_CRUD
              value: http://crud-service.default.svc.cluster.local:80/
            - name: BACKEND_STRING_AUTH
              value: http://auth-service.default.svc.cluster.local:81
            - name: JWT_LIFETIME_MIN
              value: "1"
```

3- We deploy our kubernetes file using the following command

```
PS C:\Users\kraouf\Desktop\UserAccount\kubernetes> kubectl apply -f UserAccount.yml
```

4-

5- To make sure everything is running. Kindly copy the external IP of the load balancer and paste it inside any browser.

```
PS C:\Users\kraouf\Desktop\UserAccount\kubernetes> kubectl get services
NAME                  TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)
auth-service          ClusterIP      10.0.1.48     <none>          81/TCP
crud-service          ClusterIP      10.0.1.167    <none>          80/TCP
kubernetes            ClusterIP      10.0.0.1      <none>          443/TCP
useraccount-service   LoadBalancer   10.0.1.7      20.87.115.234   80:31425/TCP
```

6-

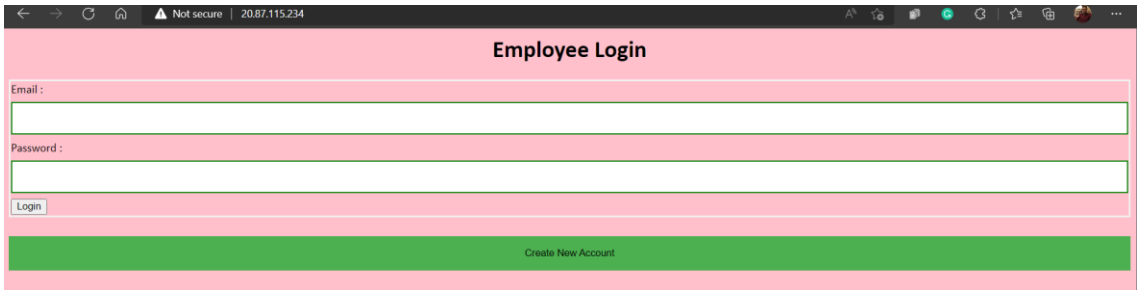
```
PS C:\Users\kraouf\Desktop\UserAccount\kubernetes> kubectl get pods
NAME                                      READY   STATUS    RESTARTS
auth-deployment-9bb9bff-4h9tm             1/1     Running   0
auth-deployment-9bb9bff-qkg9h             1/1     Running   0
crud-deployment-748f469857-clfc6          1/1     Running   0
crud-deployment-748f469857-ggn4j          1/1     Running   0
useraccount-deployment-748f957948-96gb5   1/1     Running   0
useraccount-deployment-748f957948-gx2sm   1/1     Running   0
```

7-

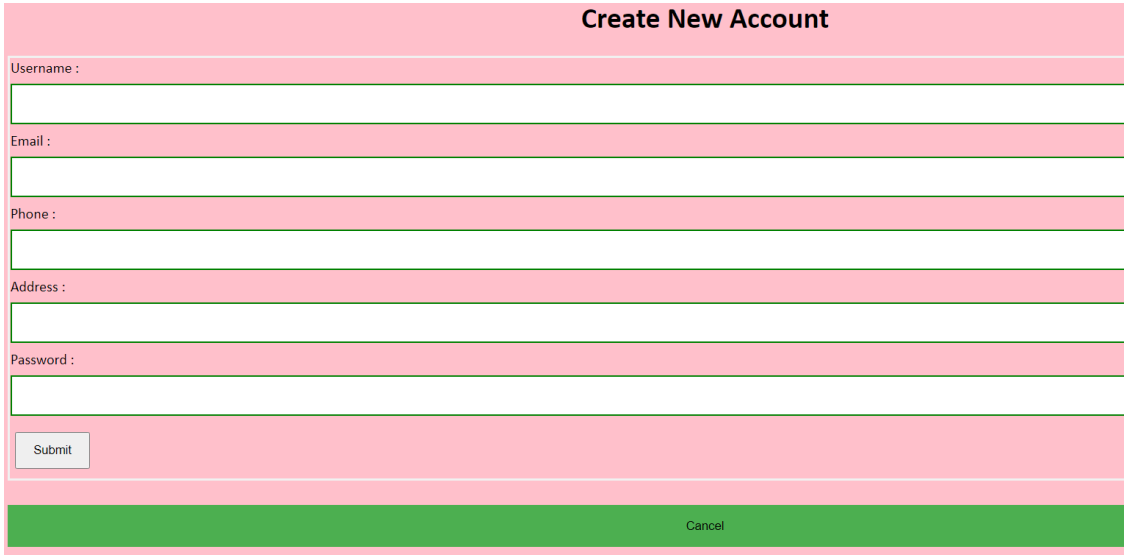
8- At this point, our project is configured and deployed on our kubernetes cluster.

9- The final picture

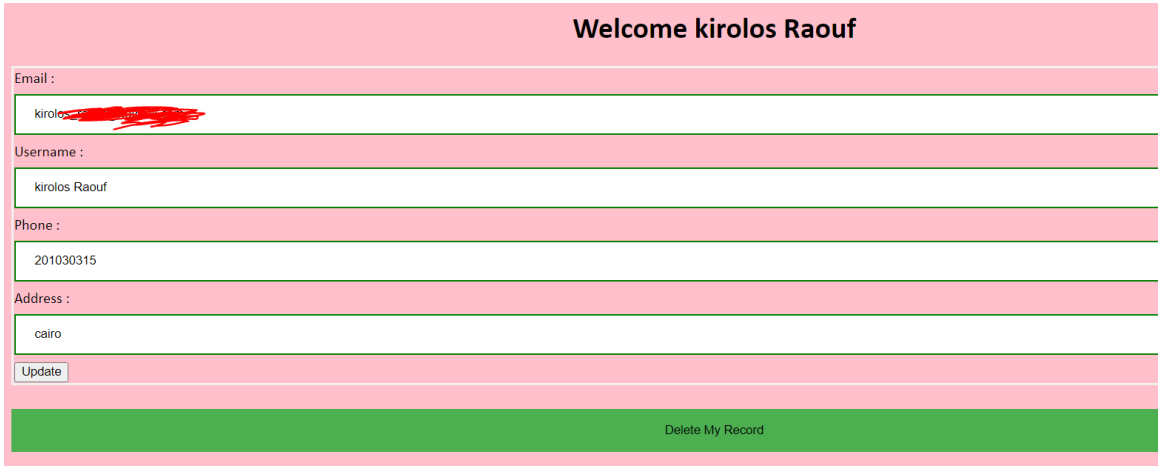
   a. Login page

b. 

c. Create a new user page

d. 

e. User data after creating a new account and login to the system

f. 

10- That's it 😊