Building and Deploying .NET Core applications on AKS and Azure Web Applications via Github Actions

Contents

[Lab: Building and Deploying .NET Core applications on AKS and Azure Web Applications via Github Actions 3](#_Toc114677955)

[Exercise 1: Building the ASP.NET Core Web API 4](#_Toc114677956)

[Exercise 2: Creating an Azure Web Application and deploying into it 9](#_Toc114677957)

[Exercise 3: Setup AKS, ACR and Azure SQL DB 22](#_Toc114677958)

[Exercise 4: Develop the Web Application 25](#_Toc114677959)

[Exercise 5: Clean up 48](#_Toc114677960)

# Lab: Building and Deploying .NET Core applications on AKS and Azure Web Applications via Github Actions

**Objectives**

After completing this lab, you will have hands-on experience with:

* Deploying an ASP.NET core application to an Azure Web App using Github.
* Setup an Azure SQL DB, connect to it.
* Setup an Azure Container Registry
* Build an ASP.NET Core Application that can communicate with an Azure Web App and an Azure SQL DB instance.
* Deploy that application to AKS. Work on the deployment and manifest files in Github.

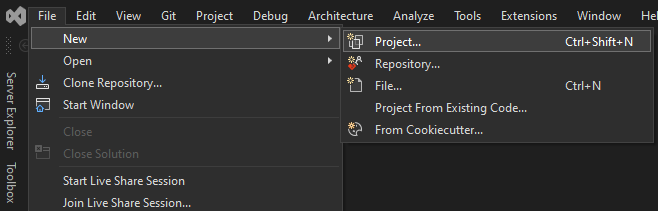
Estimated Time to complete this lab: 120 **minutes**.

## Exercise 1: Building the ASP.NET Core Web API

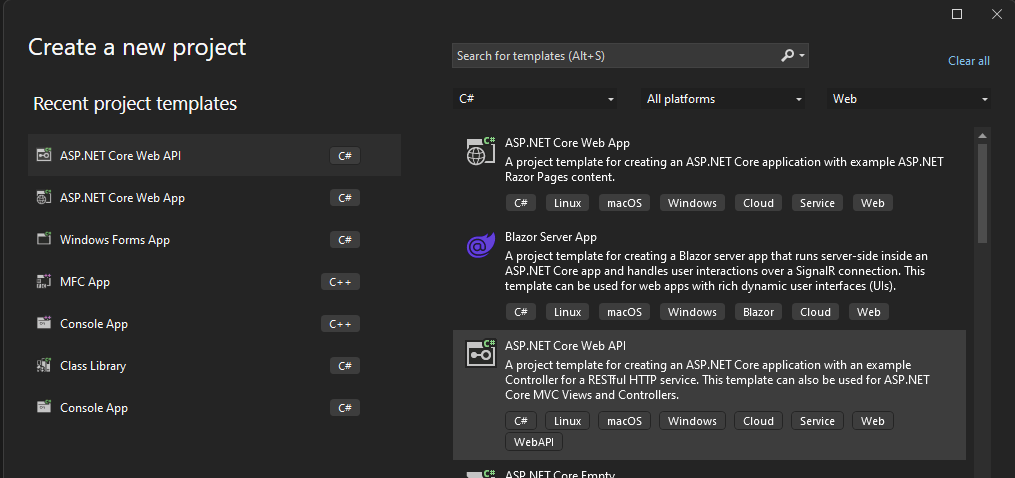
Estimated Time to complete this exercise: 30 min

**Task Description**

1. Create a folder in a drive like d:\AKSLab
2. Launch Visual Studio 2022 and create a new project:



1. In the “Create new project” window filter the project templates so that the **C#** language and **Web** type project templates are listed. Select the “**ASP.NET Core WebAPI**” template and click “Next”



1. In the “**Configure your new project**” step give your project a name, specify the folder as the one that you have created in the first step. And check the “**Place solution and project in the same directory**” option. Then, click **Next**.

Graphical user interface, application, website

Description automatically generated

1. In the “**Additional information**” window select the **Framework** that has the LTS support. E.g., .NET 6.0. Uncheck the “**Configure for HTTPS**” leave the rest of the options to their default values. Finally, click the “**Create**” button.

A screenshot of a computer

Description automatically generated

1. After the project is created, debug the application.

Graphical user interface, application

Description automatically generated

1. One console application should start, and one browser window should appear.

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

The browser defaults to the Swagger UI. Test the “**WeatherForecast API**” at the address <http://localhost:5222/WeatherForecast>

You should see some return values in JSON format like the following:

Graphical user interface, text

Description automatically generated

1. It is time to add this project into a public repository.
2. In the “**Git**” menu click the “**Create Git Repository…**” item.

Graphical user interface, application

Description automatically generated

1. In the “**Create a Git repository**” step, select “**GitHub**” as “**a new remote**” and uncheck “**Private repository**” option, fill your Github user account details. When you are done click the “Create and Push” button.

Graphical user interface, application

Description automatically generated

1. Once successful then your new repository should appear among the other ones in your Github account.

A screenshot of a computer screen

Description automatically generated with medium confidence

## Exercise 2: Creating an Azure Web Application and deploying into it

Estimated Time to complete this exercise: 20 min.

**Task Description**

You will work on the Azure and Github portals.

1. Navigate to <https://portal.azure.com/> and login to your Azure subscription with your user account.
2. On the left menu click on the “Resource groups” item.

Graphical user interface, text, application

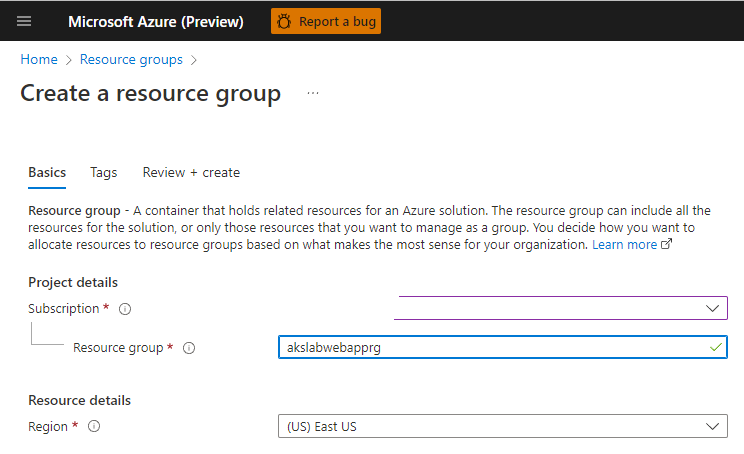
Description automatically generated

1. In the “Resource Groups” page click on the “+ Create” button

Graphical user interface, text, application

Description automatically generated

1. Select your “**Subscription**” then assign a name in “Resource group”. Specify a region for the new Resource Group. Finally, click the “**Review + create**” button on the bottom of the page. After validation is passed click the “**Create**” button.

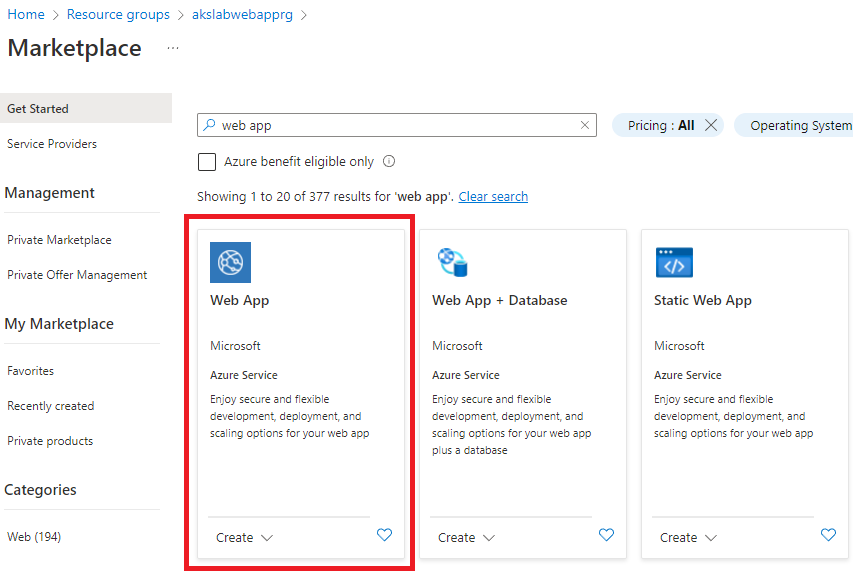


1. After the new resource group is created you will be redirected to the “**Resource groups**” page.
2. Click on the new resource group’s link so that you can switch to its “**Overview**” page. In this page click on the “**+ Create**” button.

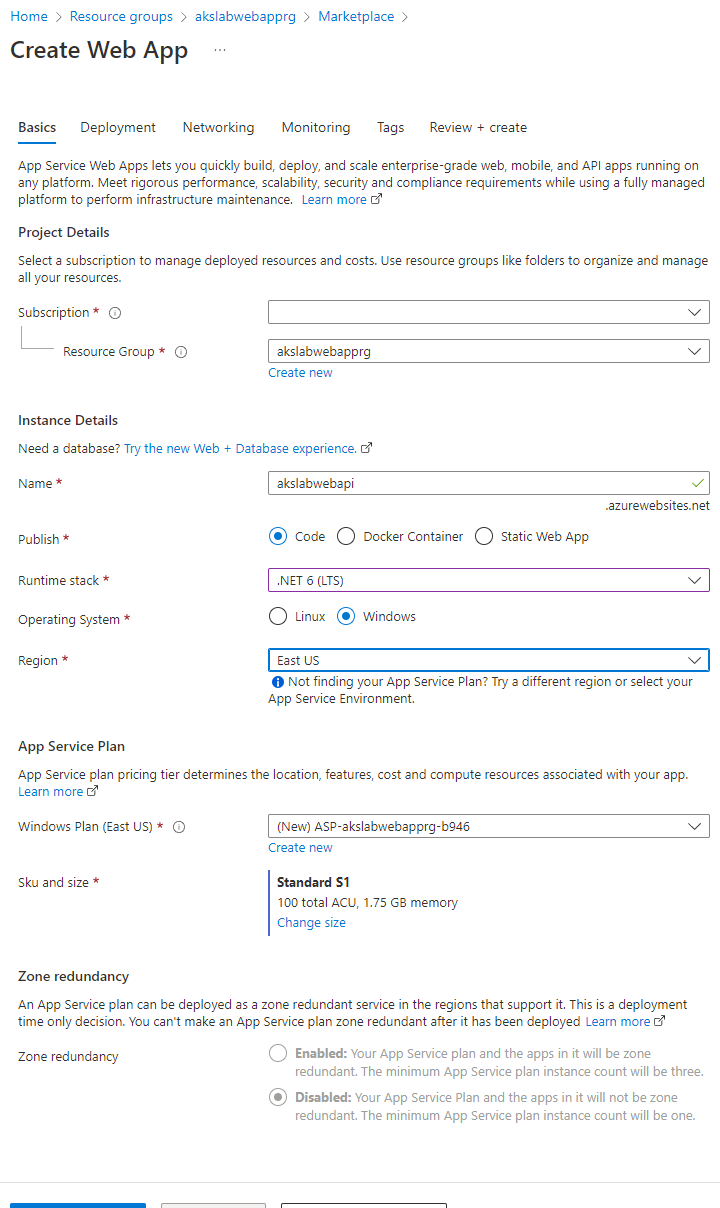
Graphical user interface, text, application, email

Description automatically generated

1. The Marketplace page will appear. In this page search for the “Web App” string.



1. Find “**Web App**” published by Microsoft. Then click on the “**Create**” link on it. “Web App” choice will appear click on it.
2. Creating a Web App is a six-step process. You will be shown the first page called “**Basics**”. In this page select your Azure Subscription, specify the resource group. This should be the one that you have created in step 4.
3. You need to come up with a unique name for your Web App in the azurewebsite.net domain.
4. You will “**publish**” “**code**”.
5. The “Runtime stack” should match the Framework that you have chosen in the Visual Studio project. Exercise 1, step 5.
6. Leave the “**Operating System**” as “Windows”.
7. Make sure that the region is the same as the Resource Group’s region. Step 4 above.



1. Leave the other settings as they are and click “**Next: Deployment**”.
2. In the “**Deployment**” page click the “**Enable**” button for “**Continuous deployment**”.

Graphical user interface, text, application, email

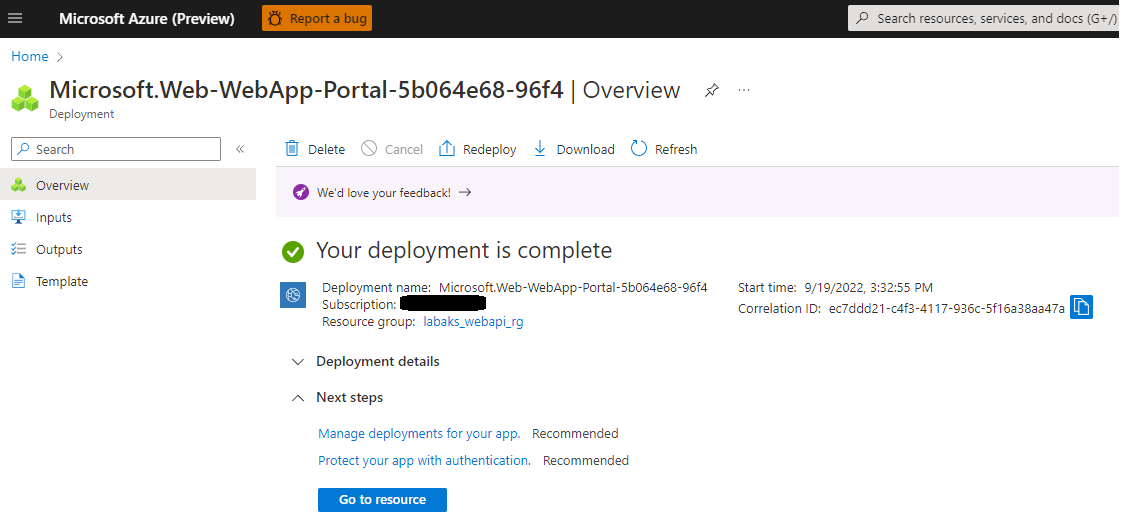
Description automatically generated

1. In the “**Github Actions details**” make the necessary selections. If you’re asked to grant permission to the “**Azure App Service Creates**” application on Github you can do so. Later you can revoke this permission in Github if you want to. See below.

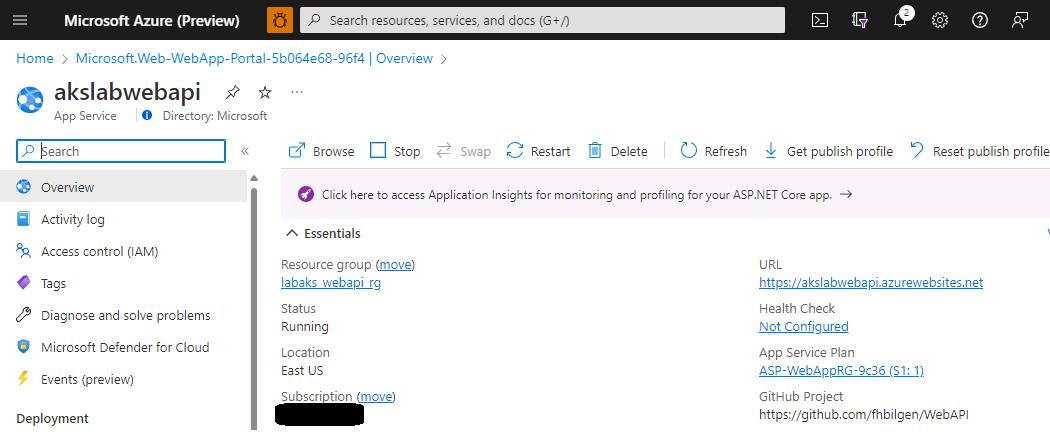
A screenshot of a computer screen

Description automatically generated

1. After you have filled out the information in the “Deployment” page then click on “**Next: Networking**”.
2. Click the “**Next: Monitoring**” button. In the “**Monitoring**” step select “**No**” for “**Enable Application Insights**” choice. Then click the “**Next: Tags**” button.
3. Click the “Next: Review + create” button.
4. Finally, click the “Create” button.
5. The deployment will start. Wait until the deployment is finished.



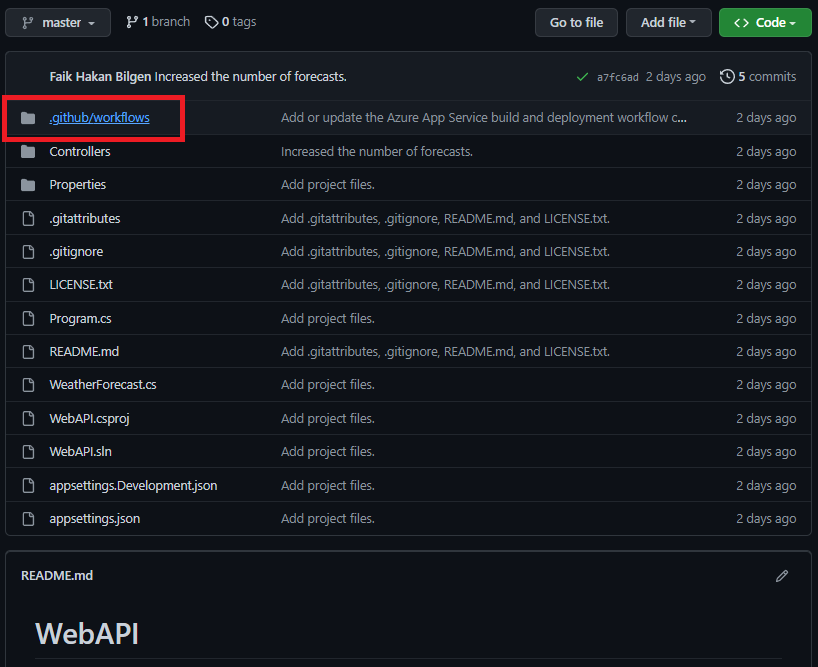
1. Click the “**Go to resource**” button. The “Overview” page for the newly created Web Application will be displayed.



1. Copy the URL on the top right-hand side. Paste it to a browser and append the /WeatherForecast string. E.g., the complete URL might be like <https://akslabwebapi.azurewebsites.net/weatherforecast>
2. Then browse to that address. You should get a JSON output like you did in Exercise 1, Step 7.



1. Make a couple of requests and observe that the count of the returned elements is five.
2. Let’s go to the Github site and open the repository for your WebAPI project and browse the “code” page. Observe that there is a new folder beneath the “.github” directory: **workflows**



1. The deployment step that we have carried out during the creation of the “Web App” caused a Github action to be created. Github action files reside beneath the “.github/workflows” directory.
2. Let’s click on the directory’s link and switch to the .github/workflows directory. There you should see the deployment YAML file. E.g., master\_akslabwebapi.yml
3. Open this file. This action is triggered when there is a push to the “**master**” branch or whenever it is run from Github’s “**Actions**“page . Also, you will see that the workflow is made from two jobs: **build** & **deploy**

name: Build and deploy ASP.Net Core app to Azure Web App - akslabwebapi

on:

push:

branches:

- master

workflow\_dispatch:

jobs:

build:

runs-on: windows-latest

steps:

- uses: actions/checkout@v2

- name: Set up .NET Core

uses: actions/setup-dotnet@v1

with:

dotnet-version: '6.0.x'

include-prerelease: true

- name: Build with dotnet

run: dotnet build --configuration Release

- name: dotnet publish

run: dotnet publish -c Release -o ${{env.DOTNET\_ROOT}}/myapp

- name: Upload artifact for deployment job

uses: actions/upload-artifact@v2

with:

name: .net-app

path: ${{env.DOTNET\_ROOT}}/myapp

deploy:

runs-on: windows-latest

needs: build

environment:

name: 'Production'

url: ${{ steps.deploy-to-webapp.outputs.webapp-url }}

steps:

- name: Download artifact from build job

uses: actions/download-artifact@v2

with:

name: .net-app

- name: Deploy to Azure Web App

id: deploy-to-webapp

uses: azure/webapps-deploy@v2

with:

app-name: 'akslabwebapi'

slot-name: 'Production'

publish-profile: ${{ secrets.AZUREAPPSERVICE\_PUBLISHPROFILE\_5D00CE231C8543289E8CF74649C5D516 }}

package: .

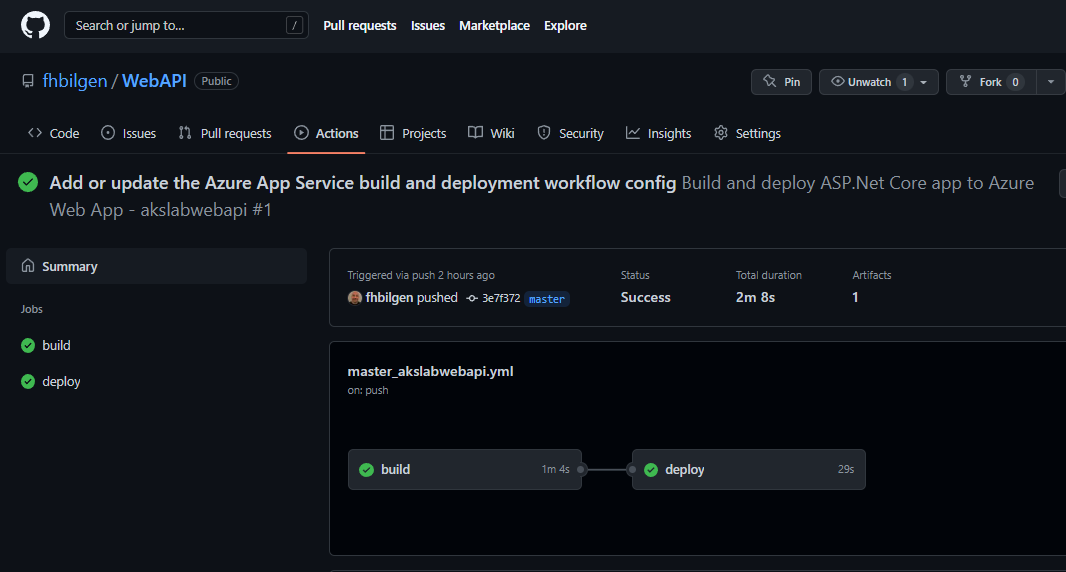
1. In the “**build**” job an agent with the latest Windows OS is spawned. .NET 6 is installed. Then, the source is built. And the artifact is stored for later access.
2. The next job “**deploy**” has been made dependent to the “build” job via the “**needs**” key. Hence, the “deploy” job has to wait for the “build” job to finish.
3. This job retrieves the artifact from the previous job and deploys it to the Web App environment in Azure.
4. Let’s switch to the “Actions” page of the repository.

A screenshot of a computer

Description automatically generated with medium confidence

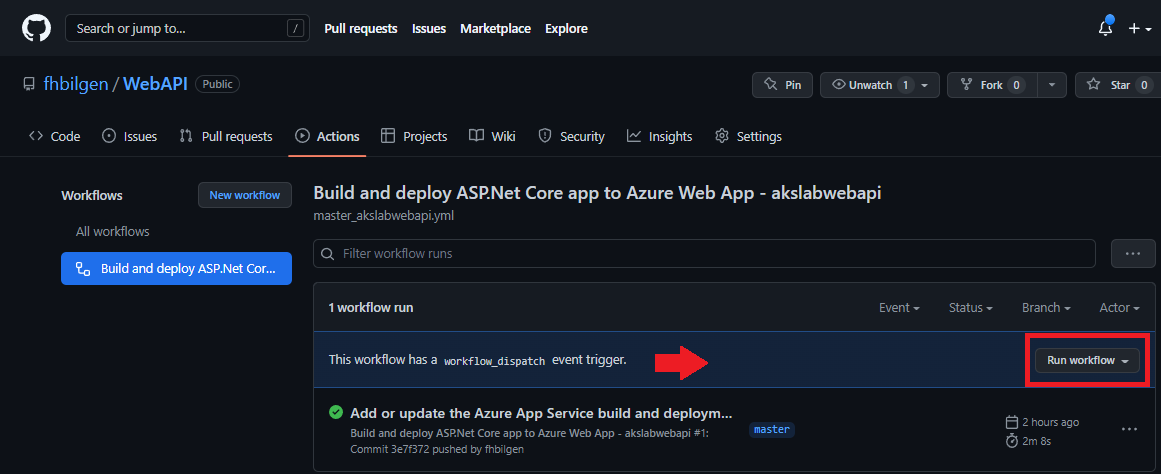
You should see one workflow is run already. This is due to the push of the workflow file to the “master” branch during the deployment of the Web App.

1. Click on the run to see the results of the jobs. E.g., the run in this sample is named as “*Add or update the Azure App Service build and deployment workflow config*”.



Here, you can see the relationship between the build and deploy jobs. If you click on either of them, you can see the detailed logs of the steps of each job. Try it out if you want to.

1. As we have seen in Step #30 above there are two ways to trigger the action’s workflow. One is manually from the Github interface. As shown in the following diagram.



1. The other is pushing some changes into the master branch. Let’s try this one.
2. Let’s go back to the Visual Studio and switch to the “Git Changes” view. Then click on the “Pull” button to get the new files after the introduction of Github Action.

Text

Description automatically generated

1. Then switch to the “**Solution Explorer**” view. Expand the “**Controllers**” folder and double click the WeatherForecastController.cs file to open it.

A screenshot of a computer

Description automatically generated with medium confidence

1. Locate the Get() function.
2. In the call to Enumerable.Range increase the upper limit from 5 to 10.

Original content:

public IEnumerable<WeatherForecast> Get()

{

return Enumerable.Range(1, 5).Select(index => new WeatherForecast

{

Date = DateTime.Now.AddDays(index),

TemperatureC = Random.Shared.Next(-20, 55),

Summary = Summaries[Random.Shared.Next(Summaries.Length)]

})

.ToArray();

}

Modified content:

public IEnumerable<WeatherForecast> Get()

{

return Enumerable.Range(1, **10**).Select(index => new WeatherForecast

{

Date = DateTime.Now.AddDays(index),

TemperatureC = Random.Shared.Next(-20, 55),

Summary = Summaries[Random.Shared.Next(Summaries.Length)]

})

.ToArray();

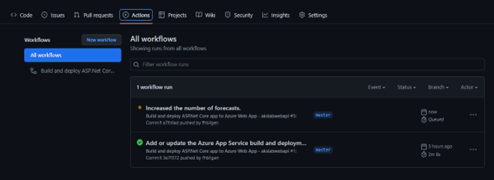
}

1. Save the solution and switch to the “Git Changes” view.
2. Enter a descriptive message for the commit.
3. Then select the “Commit all and push” option.

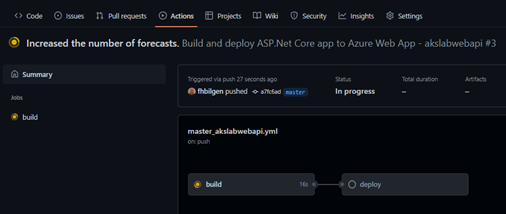
A screenshot of a computer

Description automatically generated

1. Quickly switch back to the Github site and click on the “Actions” page. You should see that the workflow has started to run.



If you click on the run, then you can see the executing job. If you click the jobs, you will see the details of the steps. This might be a good place to understand the reason if something goes wrong.



1. Once the workflow runs to completion you can browse the address from step 24. In this example’s case it is <https://akslabwebapi.azurewebsites.net/weatherforecast>
2. If the deployment was successful, then we should see ten forecasts instead of five.
3. How many forecasts did you receive?

## Exercise 3: Setup AKS, ACR and Azure SQL DB

Estimated Time to complete this exercise: 15 min

**Task Description**

You will work with Azure portal and Azure Cloud Shell.

1. Switch back to the Azure portal. Click on the Cloud Shell button.



1. If you have never run before the “Cloud Shell” then it might take some time to set up the storage resources.
2. Once the shell window is open, please check that you are in “**Bash**” and not “**PowerShell**”.

Text

Description automatically generated

1. We need to create a Resource Group for the AKS deployment. To do so let’s execute the following command. It will cause a resource group named “akslabrg” to be created in “eastus” region.

**az group create --name *akslabrg* --location *eastus***

1. After the resource group is created it is time to create the AKS. Let’s execute the following command. It will create an AKS named “aksLab” in the “aksLabrg” resource group. It will be a single node deployment with http application routing enabled.

**az aks create -g *akslabrg* -n *akslab* --enable-managed-identity --node-count 1 --generate-ssh-keys --enable-addons http\_application\_routing**

*Note: It should take around five minutes for the deployment to be finished.*

1. Then we must create the **Azure Container Registry** where we can store our docker images which will be deployed into the AKS pods. The command to execute is:

**az acr create --resource-group akslabrg --name akslabacr --sku Basic --admin-enabled "true"**

An ACR named “**akslabacr**” will be created in the “**akslabrg**” resource group.

1. Now we will create the Azure SQL DB. First, let’s create a resource group for it by executing the following command:

**az group create --name akslabsqldbrg --location eastus**

1. Then we need to create a logical server that is responsible for administrative purposes. The admin user declared here should never be used by applications to connect to the SQL DB.

**az sql server create -g akslabsqldbrg -n akslabsqlsrv --admin-user AN\_ADMIN\_ACCOUNT\_NAME --admin-password SomeVery\_STRONG\_Passw0rd!**

It should take around two-three minutes to complete.

1. Now, we can create a SQL DB named “aksdb” with the following command:

**az sql db create -g akslabsqldbrg -n aksdb -s akslabsqlsrv --service-objective GP\_S\_Gen5\_2**

1. The public IP address of your development computer should be added in the exceptions of the SQL DB Firewall. You can learn the public ip address of the dev computer by just searching “what is my ip?” in any search engine. Note that IP address. Then execute the following command:

**az sql server firewall-rule create -g akslabsqldbrg -s akslabsqlsrv -n devIP --start-ip-address 78.183.99.84 --end-ip-address 78.183.99.84**

Assuming the IP address is 78.183.99.84.

1. Also, all the IP addresses within Azure should be allowed to connect. At least for now for testing purposes.

**az sql server firewall-rule create -g akslabsqldbrg -s akslabsqlsrv -n AllowAllWindowsAzureIps --start-ip-address 0.0.0.0 --end-ip-address 0.0.0.0**

1. We need to create a SQL DB user to use in our applications. To connect to a SQL Server, you can use the sqlcmd application in the “cloud shell”. First you need to connect to SQL Server.

**sqlcmd -S akslabsqlsrv.database.windows.net -d aksdb -U AN\_ADMIN\_ACCOUNT\_NAME -P SomeVery\_STRONG\_Passw0rd!**

Note: If you cannot connect to the server with an error message like the following then you should add the cloud shell’s terminal’s IP address as an exception, too.

*“Sqlcmd: Error: Microsoft ODBC Driver 17 for SQL Server : Cannot open server 'akslabsqlsrv' requested by the login. Client with IP address '51.124.247.14' is not allowed to access the server. To enable access, use the Windows Azure Management Portal or run sp\_set\_firewall\_rule on the master database to create a firewall rule for this IP address or address range. It may take up to five minutes for this change to take effect..”*

E.g., execute the following command:

**az sql server firewall-rule create -g akslabsqldbrg -s akslabsqlsrv -n cloudshellIP --start-ip-address 51.124.247.14 --end-ip-address 51.124.247.14**

1. Once the sqlcmd tool is connected to the server execute the following command to create a SQL DB user to be used by the applications for connecting to the DB.

**create user [app-user] with password = ‘very\_strong\_password’;**

**go**

1. Let’s add the DB user to the DB owner group so that it can create/alter DB objects. Execute the following command:

**alter role [db\_owner] add member [app-user];**

**go**

1. Then execute **quit** to exit from sqlcmd. We will continue to use sqlcmd with the db user we have just created.
2. Let’s upload the DB initialization file “initSQL.sql” to “cloud shell”. To do so click on the “Upload/download” button.



After the file is uploaded then verify with running **ls -al**

The output should contain the initSQL.sql file.

1. To execute this SQL script file, execute the following command:

**sqlcmd -S akslabsqlsrv.database.windows.net -d aksdb -U sqluser -P ‘strong\_password’ -i ./initSQL.sql**

Assuming the DB user is called sqluser. If the command succeeds, then the output should be as follows:

(1 rows affected)

(1 rows affected)

(1 rows affected)

(1 rows affected)

(1 rows affected)

(1 rows affected)

(1 rows affected)

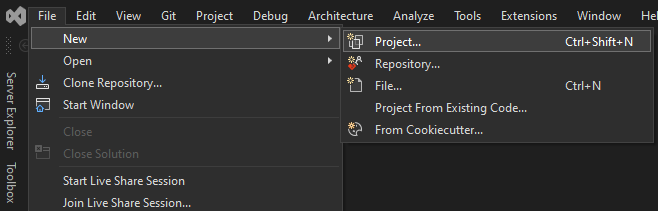
## Exercise 4: Develop the Web Application

Estimated Time to complete this exercise: 45 min

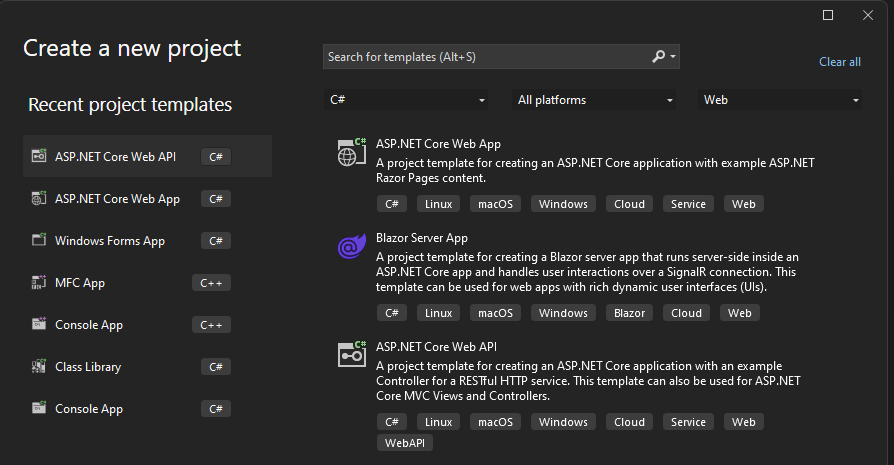
**Task Description**

You will work with Visual Studio, Github and Azure portal.

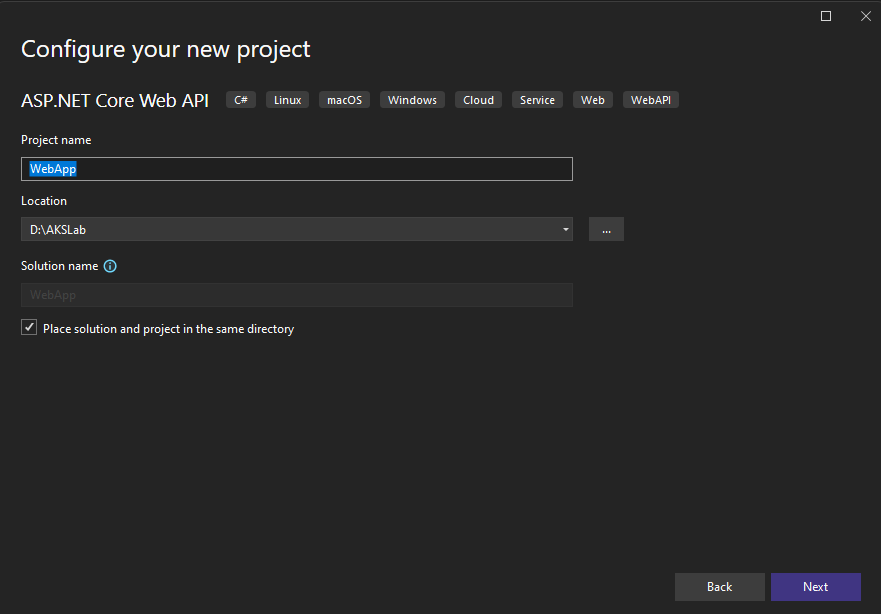
1. Launch Visual Studio 2022 and create a new project:



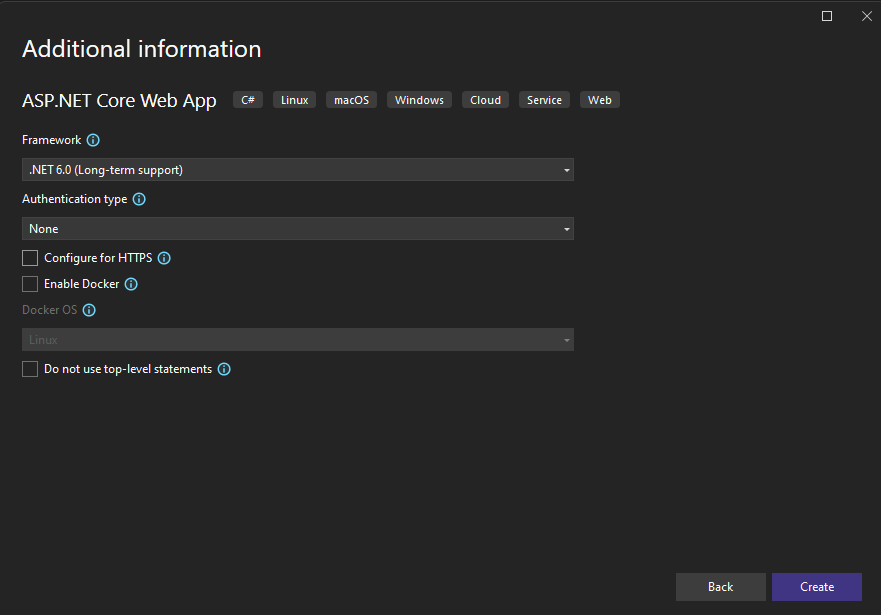
1. In the “Create new project” window filter the project templates so that the **C#** language and **Web** type project templates are listed. Select the “**ASP.NET Core Web App**” template and click “**Next**”



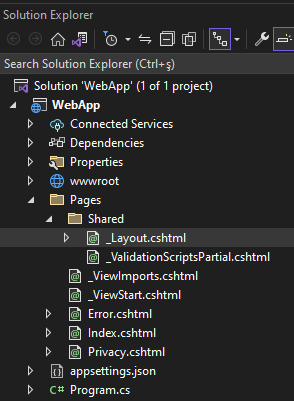
1. In the “**Configure your new project**” step give your project a name, specify the folder as the one that you have created in the first step. And check the “**Place solution and project in the same directory**” option. Then, click **Next**.



1. In the “**Additional information**” window select the **Framework** that has the LTS support. E.g., .NET 6.0. Uncheck the “**Configure for HTTPS**”, uncheck the “Enable Docker” option and leave the rest of the options to their default values. Finally, click the “**Create**” button.



1. In the “Solution Explorer” view expand the Pages and Shared folder. Then, double click the \_Layout.cshtml file to open it.



1. Find the following section

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/Index">Home</**a**>

</li>

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/Privacy">Privacy</**a**>

</li>

</ul>

1. Let’s add two more pages. One for testing the Web API invocation (the WeatherForecast service) and the other testing data retrieval from the Azure SQL DB data source. After adding the pages, the code segment should look as follows.

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/Index">Home</**a**>

</li>

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/Privacy">Privacy</**a**>

</li>

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/TestWebApp">Test Web App</**a**>

</li>

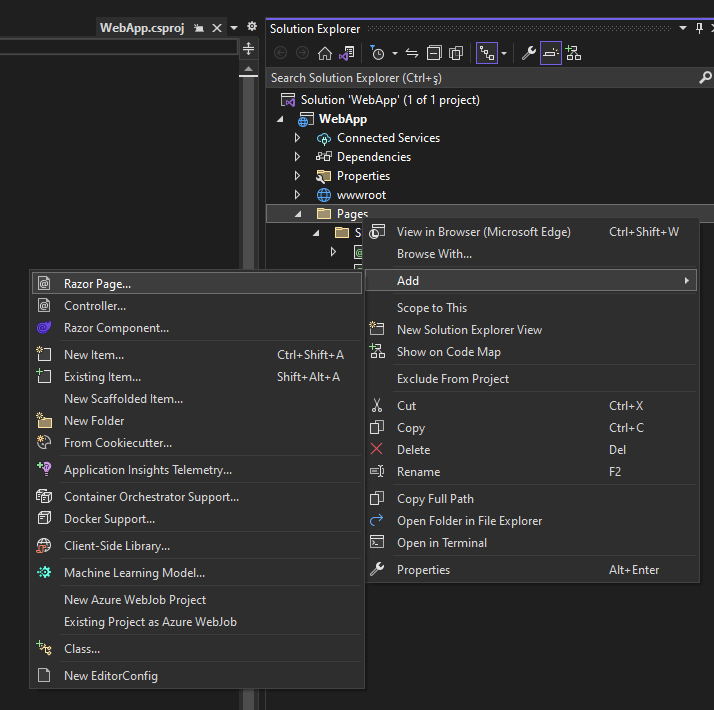
<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/TestAzSQLDB">Test Azure SQL DB</**a**>

</li>

</ul>

1. Now we need to add two razor pages. Let’s add the “**TestWebApp**” Razor Page first. Right click the “**Pages**” folder in the “**Solution Explorer**” view. Select the “**Add**” menu item and then click the “**Razor Page…**” item.

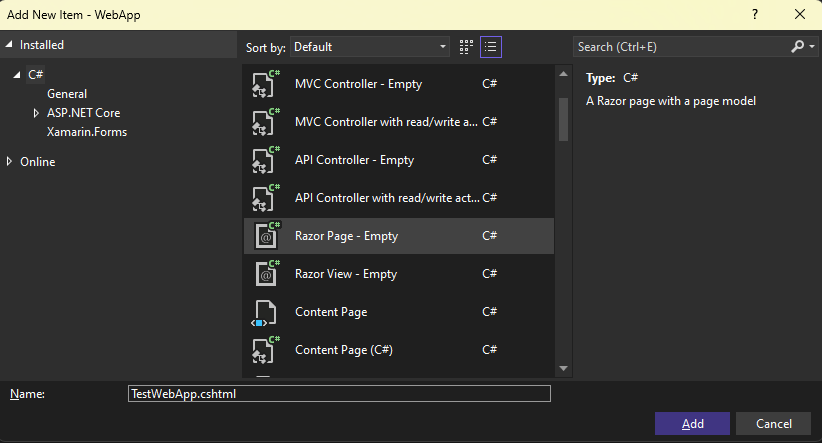


1. In the “**Add New Scaffolded Item**” step select the “**Razor Page - Empty**” option and click the “**Add**” button.

Graphical user interface

Description automatically generated

1. In the “**Add New Item**” step make sure that the “**Razor Page - Empty**” is selected and the name of the Razor Page is **TestWebApp.cshtml**. Click the “**Add**” button.

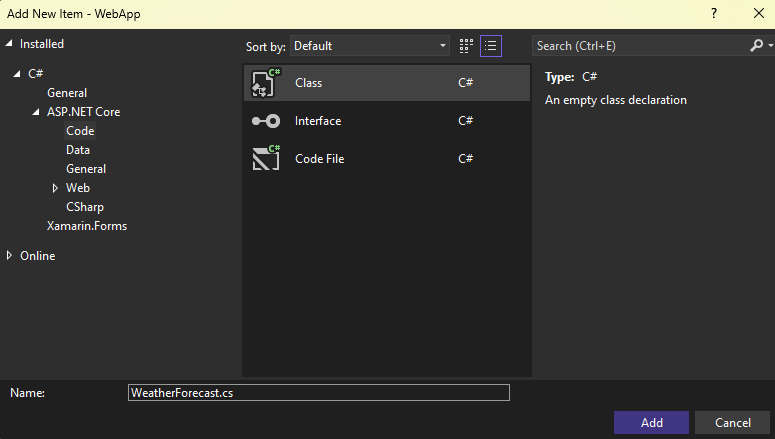


1. Before making changes to this page let’s add the class definition of the “**WeatherForecast**” class. Right click on the project and click “**Add**” and then “**Class…**”.

Text

Description automatically generated

1. In the “**Add New Item**” step select the “**Class**” template and name the class file as “**WeatherForecast.cs**”. Click the “**Add**” button.



1. In the WeatherForecast.cs file add the following highlighted properties into the “WeatherForecast” class.

public class WeatherForecast

{

public DateTime date { get; set; }

public int temperatureC { get; set; }

public int temperatureF => 32 + (int)(temperatureC / 0.5556);

public string? summary { get; set; }

}

1. Open the “TestWebApp.cshtml.cs” file.
2. Add the following:

private static readonly HttpClient client = new HttpClient();

private static string reqUrl = "https://akslabwebapi.azurewebsites.net/weatherforecast";

public IList<WeatherForecast>? Forecasts { get; set; }

**Note**: The value of reqUrl will be different in your case. Refer to the URL that you have tested in Exercise 2 Step 24.

1. Add the following two functions in the “**WeatherForecast**” class to access the Web App hosted Web API.

private static async Task<List<WeatherForecast>> GetForecasts()

{

List<WeatherForecast> fcs = new List<WeatherForecast>();

var streamTask = await client.GetStreamAsync(reqUrl);

await foreach (var fc in JsonSerializer.DeserializeAsyncEnumerable<WeatherForecast>(streamTask))

fcs.Add(fc);

return fcs;

}

public bool gfcWrapper()

{

Forecasts = GetForecasts().Result;

if (Forecasts == null)

return false;

else

return true;

}

1. Finally, make the following change to the OnGet function.

public void OnGet()

{

gfcWrapper();

}

1. Now open the “TestWebApp.cshtml” file.
2. Add the following table declaration and population code.

<table class="table">

<thead>

<tr>

<th>Date</th>

<th>Feels</th>

<th>TempF</th>

<th>TempC</th>

</tr>

</thead>

<tbody>

@if (Model.Forecasts != null)

{

foreach (var forecast in Model.Forecasts)

{

<tr>

<td> @forecast.date</td>

<td> @forecast.summary</td>

<td> @forecast.temperatureF</td>

<td> @forecast.temperatureC</td>

</tr>

}

}

</tbody>

</table>

1. It is time to test the changes. Start the application by hitting the F5 button. A browser window should appear. You should notice the added “**Test Web App**” and “**Test Azure SQL DB**” navigation choices.

Graphical user interface, application, Word

Description automatically generated

1. Click the “Test Web App” link. If everything goes well then you should see a list of 10 forecasts.

A screenshot of a computer

Description automatically generated

1. Each time you refresh the page a new set of data should be displayed. Stop debugging if everything goes well.
2. Now, let’s add the “**TestAzSQLDB**” page.
3. Right click the “**Pages**” folder in the “**Solution Explorer**” view. Select the “**Add**” menu item and then click the “**Razor Page…**” item.

A screenshot of a computer

Description automatically generated with medium confidence

1. In the “**Add New Scaffolded Item**” step select the “**Razor Page - Empty**” option and click the “**Add**” button.

Graphical user interface

Description automatically generated

1. In the “**Add New Item**” step make sure that the “**Razor Page - Empty**” is selected and the name of the Razor Page is **TestAzSQLDB.cshtml**. Click the “**Add**” button.

A screenshot of a computer

Description automatically generated with medium confidence

1. Before making changes to this page let’s add the definition of the “**City**” class. Right click on the project and click “**Add**” and then “**Class…**”.

Text

Description automatically generated

1. In the “**Add New Item**” step select the “**Class**” template and name the class file as “**City.cs**”. Click the “**Add**” button.

A screenshot of a computer

Description automatically generated with medium confidence

1. In the “**City.cs**” file add the following highlighted section into the “**City**” class.

public class City

{

public City(string cname, int ccode, int tcode)

{

Name = cname;

CallCode = ccode;

TrafficCode = tcode;

}

public string Name { get; set; }

public int CallCode { get; set; }

public int TrafficCode { get; set; }

}

1. Let’s open the “**TestAzSQLDB.cshtml.cs**” file and add the following property.

public IList<City>? Cities { get; set; }

1. Now, let’s add the following function that will retrieve data from the Azure SQL DB instance.

In the code below use the Username and Password of the Database user that you have created in Exercise 3 Step 12. For the SQL Server instance name use the name you have provided in Exercise 3 Step 8.

**IMPORTANT: It is not recommended to put any username and password information into source code. It is also against all security recommendations to store such code in Github or other source control systems. Therefore, after this lab, please delete the connection string information from the source code. Follow security guidelines for connection strings !!!**

public List<City> GetData()

{

List<City> cities = new List<City>();

try

{

SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();

builder.DataSource = "SQLSERVERNAME.database.windows.net";

builder.UserID = "app-ser";

builder.Password = “VERY\_STRONG\_PASSWORD";

builder.InitialCatalog = "aksDB";

using (SqlConnection connection = new qlConnection(builder.ConnectionString))

{

connection.Open();

string sql = "SELECT name, callcode, trafficcode FROM dbo.cities";

using (SqlCommand command = new SqlCommand(sql, connection))

{

using (SqlDataReader reader = command.ExecuteReader())

{

while (reader.Read())

{

var cname = reader["name"].ToString();

int ccode, tcode;

int.TryParse(reader["callcode"].ToString(), out ccode);

int.TryParse(reader["trafficcode"].ToString(), out tcode);

cities.Add(new City(cname, ccode, tcode));

}

}

}

}

}

catch (SqlException e)

{

return null;

}

return cities;

}

1. Then make the following change to the OnGet function

public void OnGet()

{

Cities = GetData();

}

1. You will observe that some classes are marked with a red squiggly line. This is because the “SqlClient” package is not installed.

Text

Description automatically generated

1. Right click the “**Dependencies**” folder in the “**Solution Explorer View**” and select the “**Manage NuGet Packages…**” item.

A screenshot of a computer

Description automatically generated with medium confidence

1. In the “**NuGet Package Management**” window click the “**Browse**” tab and type “**system.data.sqlclient**” and hit Enter. In the result set you should see the “**System.Data.SqlClient**” by Microsoft.

Text

Description automatically generated

1. Select it. On the right-hand side of the screen, you will see the details of the package.

Text

Description automatically generated

Click the “**Install**” button.

1. Accept the License Agreement when asked.
2. Let’s open the “**TestAzSQLDB.cshtml**” file and add the following table definition and population code.

<table class="table">

<thead>

<tr>

<th>City</th>

<th>Traffic Code</th>

<th>Call Code</th>

</tr>

</thead>

<tbody>

@if (Model.Cities != null)

{

foreach (var city in Model.Cities)

{

<tr>

<td> @city.Name</td>

<td> @city.TrafficCode</td>

<td> @city.CallCode</td>

</tr>

}

}

</tbody>

</table>

1. It is time to test the TestAzSQLDB page. Press F5 to start the project.
2. If all goes well then you should see some results when you click “**Test Azure SQL DB**” link.

Table

Description automatically generated

1. You should add “Docker” support. Right click the project and click “Add” then “Docker Support…”

Text

Description automatically generated

1. In the “**Docker File Options**” step select the “**Linux**” for the “**Target OS**” option. Click the “**OK**” button.

Graphical user interface, application

Description automatically generated

1. This will create a file named “Dockerfile” to be added into the project.

A screenshot of a computer

Description automatically generated with medium confidence

1. Open the “**Dockerfile**” and examine the content.
2. You will see that there is one container named “**build**” that is made from a .NET 6 installed image. This container is used to build the “WebApp” application. Then a container called “**publish**” is used to publish the image and based on that image the final image is built.
3. Now, it is time to add the project to Github.
4. In the “**Git**” menu click the “**Create Git Repository…**” item.

Graphical user interface, application

Description automatically generated

1. In the “**Create a Git repository**” step, select “**GitHub**” as “**a new remote**” and uncheck “**Private repository**” option, fill your Github user account details. When you are done click the “Create and Push” button.

Graphical user interface, text

Description automatically generated

1. Once successful then your new repository should appear among the other ones in your Github account.

A screenshot of a computer

Description automatically generated

1. Now, you should create a deployment on the AKS resource.
2. Go to the Azure portal and open the Resource Group of the AKS resource.

Graphical user interface, application

Description automatically generated

1. Then open the AKS resource by clicking on it.

On the “Kubernetes service”s Overview page click on the “Deployment Center (preview)” menu

Graphical user interface, text, application

Description automatically generated

1. In the “**Select the code location**” step click on the “Github” option and click the “Next” button.

Graphical user interface, text, application

Description automatically generated

1. In the “Select a repository” step Azure might ask for permission. Click on the “Authorize” button.

A picture containing graphical user interface

Description automatically generated

1. After the authorization is done select the “**WebApp**” repository and the “**master**” branch.

Timeline

Description automatically generated with low confidence

1. Click the “OK” button.
2. In the application settings confirmation page verify that the correct Dockerfile is located, and correct values are depicted. You should be familiar with the contents of the “Dockerfile” from steps 39 & 40 above.

Graphical user interface, text, application, email

Description automatically generated

1. Click the “**Next**” button.
2. In the last step, select “create new” namespace. Give a name that describes the purpose of the application. Make sure that the ACR that you have created earlier is selected. Exercise 3 Step 6.

Graphical user interface, text, application

Description automatically generated

1. Click the “Done” button.
2. After a few minutes you will see a page as follows:

Graphical user interface, text, application, email

Description automatically generated

1. It is time to switch to Github. Open the “WebApp” repository and click the “Actions” link. What Azure has done is to create a deployment workflow into the Github Actions. Therefore, we should see the outcome of a run. This will be a failure:

A screenshot of a computer

Description automatically generated

1. The error message that you will see below will be like the following one.

*“Error: error: resource mapping not found for name: "akslab-14b0" namespace: "" from "/tmp/Ingress\_akslab-14b0\_1663697478177": no matches for kind "Ingress" in version "extensions/v1beta1"*

*ensure CRDs are installed first”*

1. Let’s take one step back and check the files that are added to the repository. Click the “Code” link.
2. There are two new folders:
   1. .github/workflows: Here there is the deployment file “deployToAksCluster.yml”
   2. Manifests: Here are the AKS deployment helper files. deployment.yml, ingress.yml and service.yml

A screenshot of a computer screen

Description automatically generated

1. Feel free to open the files and examine their contents.
2. Now, let’s go back to the error message. The message points to the ingress.yml file.
3. If you open the “ingress.yml” file, you will see that the API version is beta1. This is obsolete.It should be changed.
4. Another problem is with the syntax of the rules part.
5. Basically, the file should be changed to look like the following:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: "akslab-14b0"

labels:

app: "akslab-14b0"

annotations:

kubernetes.io/ingress.class: addon-http-application-routing

spec:

rules:

- host: webapp-akslab-14b0.494a36c84da14a328317.eastus.aksapp.io

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: "akslab-14b0"

port:

name: http

1. Commit the changes and switch to the “Actions” page and follow the execution of the run.
2. This time the workflow should be completed with success.
3. To test you can use the host that is in the ingress.yml file. Simply put the value in the host key to the address of a browser. If everything is ok, then you should see the working page.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

Text

Description automatically generated

## Exercise 5: Clean up

Estimated Time to complete this exercise: 10 min

**Task Description**

You will use Azure Cloud Shell and Github to remove the created resources.

1. Delete the Web App. Open the Cloud Shell and execute the following command

**az group delete --name akslabwebapprg**

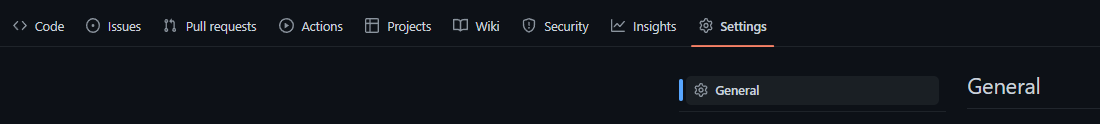
1. Delete the Azure SQL DB

**az group delete --name akslabsqldbrg**

1. Delete the AKS and ACR

**az group delete --name akslabrg**

1. Remove the github repository for WebApi. Click on the “**WebApi**” repository. Then, click on the “**Settings**” button.

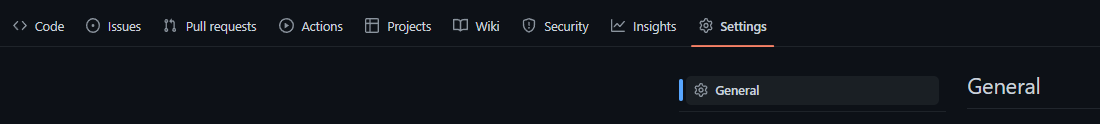


Scroll down until you arrive at the “**Danger Zone**”. There, click on the “**Delete this repository**” button.

A screenshot of a computer

Description automatically generated with medium confidence

1. Remove the github repository for WebApp. Click on the “**WebApi**” repository. Then, click on the “**Settings**” button.



Scroll down until you arrive at the “**Danger Zone**”. There, click on the “**Delete this repository**” button.

A screenshot of a computer

Description automatically generated with medium confidence

1. You can revoke permissions of any Authorized Applications on Github anytime. Click on your profile and select the “**Settings**” item.

Graphical user interface, application

Description automatically generated with medium confidence

On the “Settings” page’s left pane click on the “Applications” item. Then on the right-hand side click on the “Authorized OAuth Apps”. You can then revoke permission from any application you want.

A screenshot of a computer

Description automatically generated with medium confidence