Modules

Module 2 - Getting Started with Windows Containers

> Duration: 60 minutes

Module 2: Table of Contents

Prerequisites

The Windows LOD VM is packaged with all of the required software to perform this lab.

The lab files are required to complete the hands-on exercises and have been pre-downloaded on the Virtual Machines. Additionally, they are also accessible for download on GitHub. For the Windows labs, download the repository Microsoft/WorkshopPLUS-Modernizing-Applications-with-Containers-and-Orchestrators and extract the content of the **windows-labs** folder on your Windows machine.

Exercise 1: Working with Nano Server & Windows Server Core containers

In this exercise, you will learn about the Windows Nano Server and Server Core images. Please read below for an overview of each image. Then you will complete the steps to build and run these containers.

Return to list of exercises - Return to list of modules

Windows Server Core Overview

Microsoft starting with Windows Server 2016 has an option of Server Core installation. The Server Core option reduces the amount of space required on disk, the potential attack surface, and especially the servicing requirements. It is recommended that you choose the Server Core installation unless you have a need for the additional user interface elements and graphical management tools that are included in the Server with Desktop Experience option. For an even more lightweight option, see the next section on Nano Server. Server Core allows you to install various Server roles that may not be available in Nano Server including those listed below:

- Active Directory Certificate Services
- Active Directory Domain Services
- DHCP Server
- DNS Server
- File Services (including File Server Resource Manager)
- Active Directory Lightweight Directory Services (AD LDS)
- Hyper-V
- Print and Document Services
- Streaming Media Services
- Web Server (including a subset of ASP.NET)
- Windows Server Update Server
- Active Directory Rights Management Server
- Routing and Remote Access Server and the following sub-roles:
- Remote Desktop Services Connection Broker
- Licensing
- Virtualization
- Volume Activation Services

For a comprehensive list of features available in Server Core, visit https://technet.microsoft.com/en-us/windows-server-docs/get-started/getting-started-with-server-core.

Windows Nano Server Overview

Nano Server is optimized as a lightweight operating system for running "cloud-native" applications based on containers and micro-services. There are important differences in Nano Server versus Server Core. As of Windows Server 2016, version 1803, Nano Server is available only as a container base OS image. You must run it as a container in a container host, such as a Server Core installation of Windows Server. Running a container based on Nano Server in this release differs from releases prior to 1803 in the ways listed below:

- Nano Server has been optimized for .NET Core applications
- Nano Server size has been optimized in Windows Server 2019 version
- PowerShell Core, .NET Core, and WMI are no longer included by default as they were in Windows Server 2016, but you can include PowerShell Core and .NET Core container packages when building your container
- There is no longer a servicing stack included in Nano Server. Microsoft publishes an updated Nano image to Docker Hub that you redeploy
- You troubleshoot the new Nano Container by using Docker
- You can now run Nano containers on IoT Core

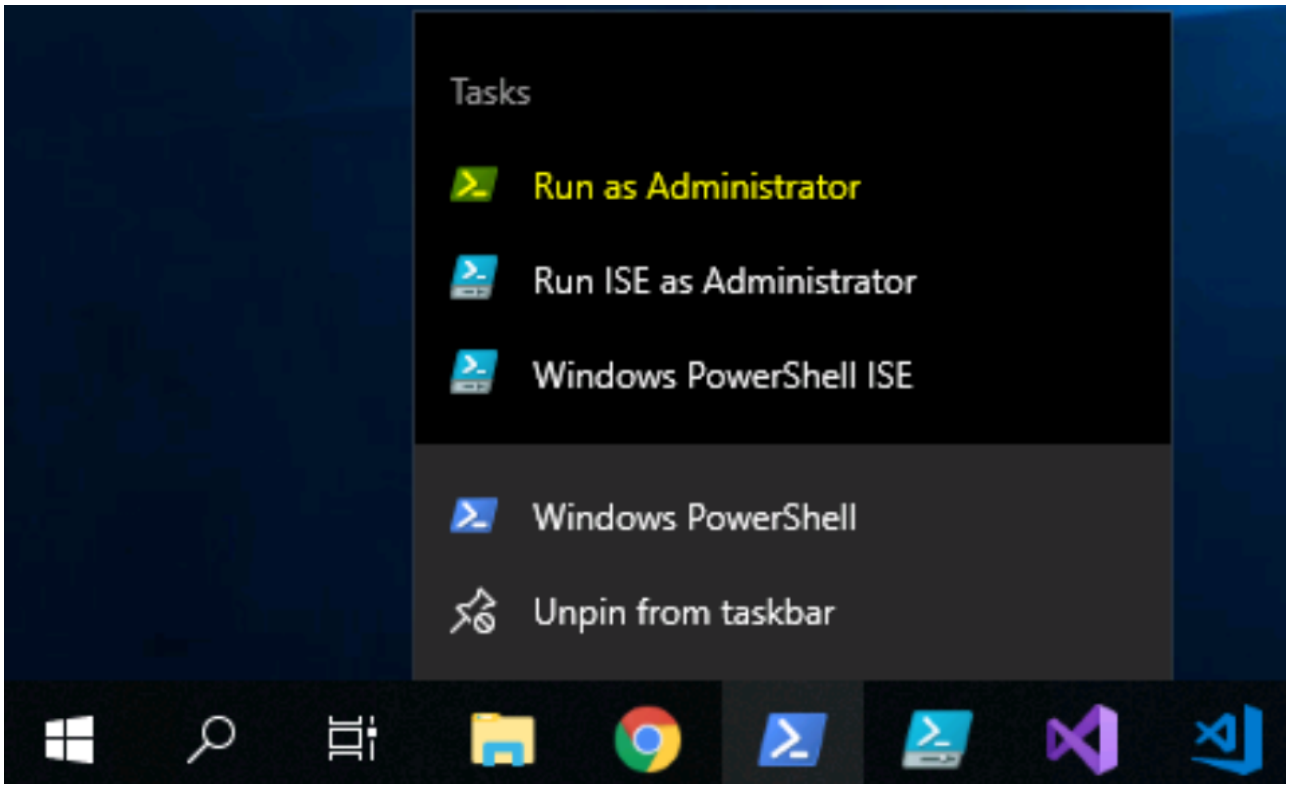
For a comprehensive list of capability differences between Nano Server and Server Core, visit https://docs.microsoft.com/en-us/windows-server/get-started/getting-started-with-nano-server)

Since Windows Server 2016, Microsoft offers both Nano Server and Server Core in the form of Docker images through Docker Hub. With the GA of Windows Server 2019, Microsoft also announced that its base images will now be hosted in the Microsoft Container Registery or MCR. For information on this change, visit https://azure.microsoft.com/en-us/blog/microsoft-syndicates-container-catalog/. These images will still be discoverable via Docker Hub. The nature of application you are building typically dictates your selection of base image. For example, the SQL Server 2016 Express image will need Server Core as its base image, but a simple windows service may able to run just fine on Nano server. In the following exercise, you will run a basic "hello world" container leveraging both the Nano and Server Core images.

Run a container based on the Nano Server base image

1. Ensure you are in the Lab on Demand (LOD) Windows VM.

2. You will need to run the commands in this section using the PowerShell console as an administrator. Right click the **PowerShell** icon on the taskbar and select "**Run as Administrator**".



3. The PowerShell console is now available to you. Make sure you are inside of the **module2** labs directory. You can do that by running the command `cd C:\labs\module2\` This will put you inside the **module2** lab folder where all the necessary files are located. Notice the file **Lab 2 Commands Sheet.txt** that contains the commands you will execute as part of this lab.

```
PS C:\Users\Administrator> cd C:\labs\module2\
PS C:\labs\module2> ls


    Directory: C:\labs\module2


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         5/10/2019   7:53 AM                aspnet4.7
d-----         5/10/2019   7:53 AM                aspnetcore
d-----         5/10/2019   7:53 AM                iis
-a----         5/10/2019   7:53 AM            196 commands.ps1
-a----         5/10/2019   7:53 AM            894 Lab 2 Commands Sheet.txt
```

4. First, let's get the list of all the container images available on this Docker host by running the command `docker images`. Notice that you already have **windows/servercore** and **windows/nanoserver** images available to you representing **Server Core** and **Nano Server** images.

> Knowledge: It's important to understand that you can always download specific version of **windows/servercore** and **windows/nanoserver** images by using an appropriate tag. For example, `docker pull mcr.microsoft.com/windows/servercore:10.0.17763.437` will pull the server core image that has a version number 10.0.17763.437. Notice the mcr.microsoft.com registry that is the container registry hosted on Microsoft servers, even though the images are discoverable on Docker Hub. All the concepts you learned about docker (Linux) containers and images generally apply as-is to windows containers too. The main deference is the fact that windows containers require the windows operating system as a host, while the Linux containers require Linux operating system.

```
PS C:\labs\module2> docker images
REPOSITORY                                  TAG                                 IMAGE ID            CREATED             SIZE
mcr.microsoft.com/windows/servercore/iis    windowsservercore-ltsc2019          47f926963b92        5 weeks ago         5.01GB
mcr.microsoft.com/dotnet/framework/aspnet   4.7.2-windowsservercore-ltsc2019    ad29882fe4a4        5 weeks ago         5.41GB
mcr.microsoft.com/dotnet/core/aspnet        3.1-nanoserver-1809                 0aa879aecd86        5 weeks ago         341MB
mcr.microsoft.com/dotnet/core/sdk           3.1-nanoserver-1809                 fb60591689e2        5 weeks ago         731MB
mcr.microsoft.com/windows/servercore        1809                                f8d2f4d3f2eb        6 weeks ago         4.82GB
mcr.microsoft.com/windows/nanoserver        1809                                c897e7c62e1e        6 weeks ago         251MB
mcr.microsoft.com/dotnet/framework/sdk      4.7.2-windowsservercore-ltsc2019    1afb4e8593d8        3 months ago        7.3GB
```

5. You will now run a container based on **Server Core** image (**mcr.microsoft.com/windows/servercore**). Before you do that, run the command `hostname`. This will reveal the hostname of your virtual machine.

> Note:Please note that your host machine name may be different.

```
PS C:\labs\module2> hostname
WIN-EJI3ACSSVHL
```

6. Run the command `docker run -it mcr.microsoft.com/windows/servercore:1809 powershell` . Please be patient as it will take a minute or so for this command to work. The **-it** switch provides you with an interactive session. The **powershell** is a parameter passed as an argument which basically gives you access to Powershell (command line) running inside the container. Technically, the **-it** switch puts you inside a running container.

> Note:Please be patient as it will take a minute or so for this command to work. The "**it**" switch provides you with an interactive session. The '**CMD**' is a parameter passed as an argument which

> basically gives you access to the CMD (command line) running inside the container. Technically, the "**it**" switch puts you inside a running container.

```
PS C:\labs\module2> docker run -it mcr.microsoft.com/windows/servercore:1809 powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
```

7. Run the command `hostname`. This time you are running it inside the running container. Notice that the host name is different from the host name you get in step 5. The host name you see inside the container is the host name of the container itself. It is based on the container ID. You may want to run other commands as you wish or checkout the filesystem that is independent from the host's filesystem.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\> hostname
4bdbbde8a36c
```

8. Finally, exit the interactive session by typing `exit` and pressing **Enter**. This will take you back to the PowerShell console on the host.

```
PS C:\> exit
PS C:\labs\module2>
```

9. Now let's run another container based on **Nano Server** image (**mcr.microsoft.com/windows/nanoserver**). To do that run the command `docker run -it mcr.microsoft.com/windows/nanoserver:1809 CMD`

> Note:It might take a few seconds to load. This time we are starting a Windows Command prompt instead of Powershell inside of the container)

```
PS C:\labs\module2> docker run -it mcr.microsoft.com/windows/nanoserver:1809 CMD
```

10. Run the command `hostname`. Notice that the host name is different from host name you get in the previous steps. Again, the host name you see inside the container is the host name of the container itself, which is based on the container id. You can run other commands as you wish.

```
Microsoft Windows [Version 10.0.17763.437]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\>hostname
d8e8e689d89f
```

11. Finally, exit the interactive session by typing `exit` and pressing **Enter**. This will take you back to the PowerShell console on the host.

Congratulations!

In this exercise, you have created and run containers based on the Windows Server Core & Nano Server container images that Microsoft provides and maintains. You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 2: Building and Running an IIS Server Windows Container Image

In the exercise you will learn how to install IIS Web Server (Web Server Role) on a Windows Server Core base core image. IIS Server is a popular Web Server released by Microsoft. Considering the strong footprint of IIS within enterprises, Microsoft supports IIS on Windows Server Core.

Return to list of exercises - Return to list of modules

Build and run an IIS Server Image

1. Make sure you have a PowerShell Console open as an administrator (if you have followed previous task you should already be running a Console). Also, change the current directory to "**iis**" by running the command `cd c:\labs\module2\iis\`

```
PS C:\labs\module2> cd iis
PS C:\labs\module2\iis> ls


    Directory: C:\labs\module2\iis


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        2/12/2018  12:02 PM            222 Dockerfile
-a----        2/12/2018  12:02 PM         135864 ServiceMonitor.exe
```

2. The iis folder contains the Dockerfile with instructions to install IIS Server (Web Server Role) on the Windows Server Core base image. Display the Dockerfile by running the command `cat .\Dockerfile`

```
PS C:\labs\module2\iis> cat Dockerfile
FROM mcr.microsoft.com/windows/servercore:1809

LABEL owner="cloudarchitectureteam@contoso.com"

RUN powershell -Command Add-WindowsFeature Web-Server

ADD ServiceMonitor.exe /ServiceMonitor.exe

EXPOSE 80

ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
```

- The **FROM** instruction points to the **mcr.microsoft.com/windows/servercore** to be used as a base image for the new container image
- The **RUN** instruction executes PowerShell to install Windows Feature "Web Server" (IIS Server)
- The next command is the **ADD** instruction which copies the **ServiceMonitor.exe** utility to the container image. The **ServiceMonitor.exe** is a utility that monitors **w3svc** service inside the container, if the service fails, the exe fails, so Docker knows the container is unhealthy. The **ServiceMonitor.exe** is developed and released by Microsoft (https://github.com/microsoft/iis-docker/tree/master/windowsservercore-ltsc2019)
- The **EXPOSE** instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published.
- The **ENTRYPOINT** instruction makes sure that monitoring of **w3svc** begins immediately as soon as container starts running. This is what will keep the container in running state.

3. To build the new image with IIS installed on it, run the command `docker build -t myiis:v1 .`. This command builds a new container image with name **myiis** and tag **v1**. The tag conveniently tells everyone information pertaining to the version of the image.

> Note: **STEP 3/6** of the build process performs the installation of the Web-Server (IIS Server) and may take few minutes. Eventually you should see the results as follow.

```
PS C:\labs\module2\iis> docker build -t myiis:v1 .
Sending build context to Docker daemon  138.8kB
Step 1/6 : FROM mcr.microsoft.com/windows/servercore:1809
 ---> 29a2c2cb7e4d
Step 2/6 : LABEL owner="cloudarchitectureteam@contoso.com"
 ---> Running in d61cf37b0259
Removing intermediate container d61cf37b0259
 ---> 43639b144ca5
Step 3/6 : RUN powershell -Command Add-WindowsFeature Web-Server
 ---> Running in 6a9a67b56d41

Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {Common HTTP Features, Default Documen...


Removing intermediate container 6a9a67b56d41
 ---> 135a6bdc4c43
Step 4/6 : ADD ServiceMonitor.exe /ServiceMonitor.exe
 ---> 52269fd95364
Step 5/6 : EXPOSE 80
 ---> Running in 1ee5693b6f25
Removing intermediate container 1ee5693b6f25
 ---> db56ed82165f
Step 6/6 : ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
 ---> Running in b79521456a76
Removing intermediate container b79521456a76
 ---> 348962ba6bba
Successfully built 348962ba6bba
Successfully tagged myiis:v1
```

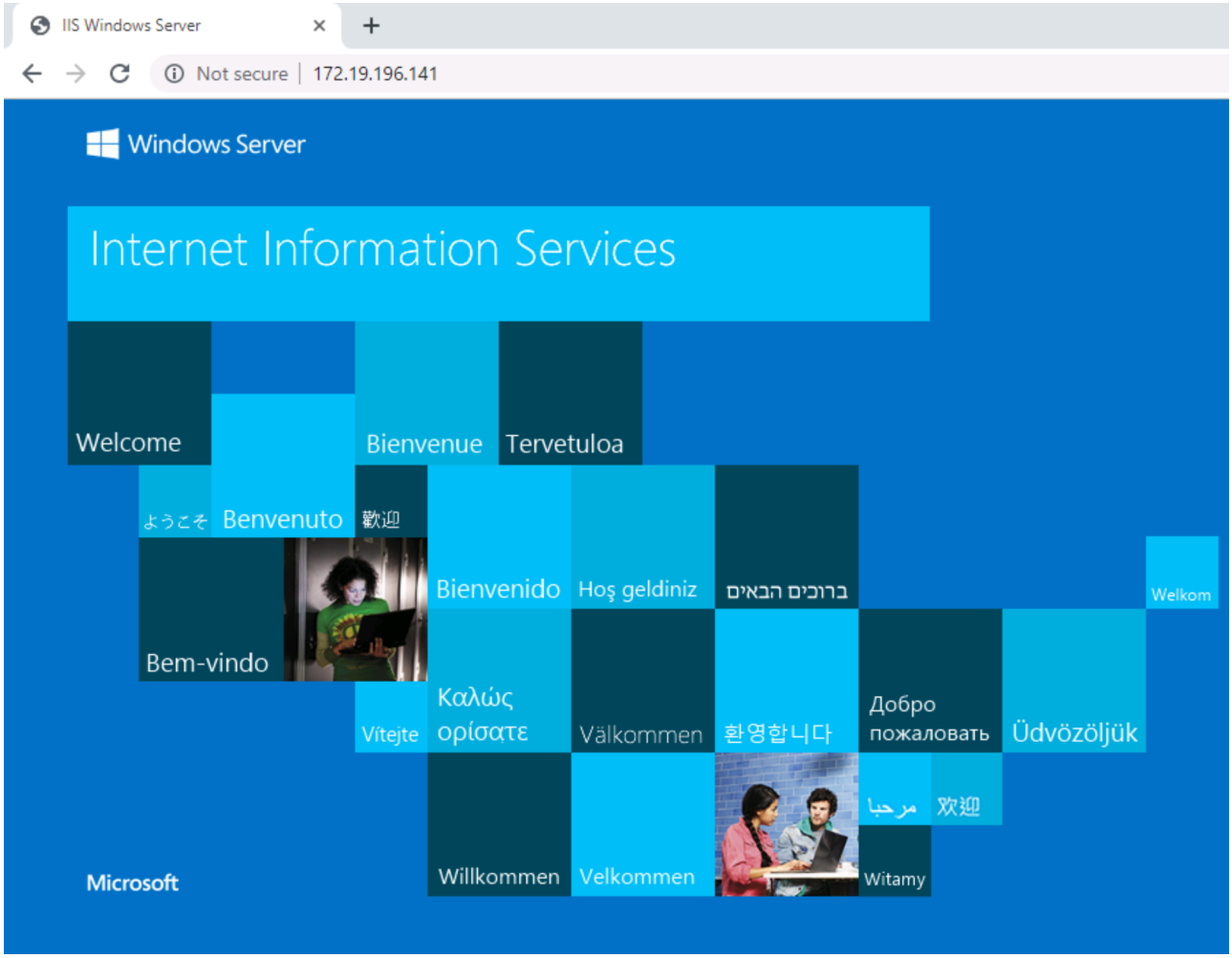
4. Run a new container based on **myiis:v1** image by using the command: `docker run -d -p 8099:80 myiis:v1`

```
PS C:\labs\module2\iis> docker run -d -p 80:80 myiis:v1
d83e6af280f7ca66188e49facd9ec267c8b8e2a1e693768aa7f205dd7069490f
```

5. The full container ID is shown after the run command (**d83** in the above screenshot), or can be obtained by using `docker ps`

6. To get the IP address of the container, run the following command:
   `start http://localhost:8099`

7. Open any web browser of your choice and browse to the IP address from the previous step.



> Note: You can get the container's IP address of the container with **docker inspect** command as follow: **docker inspect --format '{{ .NetworkSettings.Networks.nat.IPAddress}} ' containerid **. When accessing the container using it's IP Address you would use the port the container is listening on (port 80 in this case)

Congratulations!

This concludes the exercise on creating a new image with IIS server. If you are looking to leverage IIS server beyond this lab, then you may want to use Microsoft official IIS server image (**mcr.microsoft.com/windows/servercore/iis**) which is available at (https://hub.docker.com/r/microsoft/iis/). The underlying process is pretty much same but the main benefit of using the official IIS image is that Microsoft releases updated images on a regular basis including patches and fixes.

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 3: Building and running an ASP.NET 4.7 application in a container

In this task, you will learn how to package an existing ASP.NET 4.7 web application into a container. It's important to understand that Microsoft supports both the latest .NET frameworks like .NET Core, ASP.NET Core etc. as well as more legacy .NET Frameworks like .NET 3.5, .NET 4.5 and ASP.NET 4.5 on Windows Containers. Most customers today have critical workloads that depend on some legacy Microsoft technologies, therefore Microsoft provides a path for application containerization for legacy applications in addition to more modern apps.

> Knowledge: You can find more comprehensive list of application frameworks supported by Microsoft on Windows Containers at: https://docs.microsoft.com/en-us/virtualization/windowscontainers/samples#Application-Frameworks

Return to list of exercises - Return to list of modules

Build and run an ASP.NET 4.7 MVC application

1. Make sure you have a PowerShell console open as an administrator (if you have followed previous task you should already be running a console). Also, change the current directory to **aspnet 4.7** by running the command `cd C:\labs\module2\aspnet4.7\`

```
PS C:\labs\module2\aspnet4.7> ls


    Directory: C:\labs\module2\aspnet4.7


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         5/6/2019   3:15 PM                WebAppLegacy
-a----         5/6/2019   3:14 PM            434 Dockerfile
```

2. Before proceeding further, let's stop and remove all the running containers from previous task. Run the command `docker rm -f (docker ps -aq)`

```
PS C:\labs\module2\aspnet4.7> docker rm -f (docker ps -aq)
d83e6af280f7
eb618b5b542a
ae998ed4771b
d8e8e689d89f
4bdbbde8a36c
```

3. Let's examine the Dockerfile. Display its content by running the command `cat .\Dockerfile`

```
PS C:\labs\module2\aspnet4.7> cat Dockerfile
FROM mcr.microsoft.com/dotnet/framework/sdk:4.7.2-windowsservercore-ltsc2019 AS build
WORKDIR /app

# copy csproj and restore as distinct layers
COPY *.sln .
COPY WebAppLegacy/*.csproj ./WebAppLegacy/
COPY WebAppLegacy/*.config ./WebAppLegacy/
RUN nuget restore

# copy everything else and build app
COPY WebAppLegacy/. ./WebAppLegacy/
WORKDIR /app/WebAppLegacy
RUN msbuild /p:Configuration=Release


FROM mcr.microsoft.com/dotnet/framework/aspnet:4.7.2-windowsservercore-ltsc2019 AS runtime
WORKDIR /inetpub/wwwroot
COPY --from=build /app/WebAppLegacy/. ./
```

- The first noticeable statements are the two **FROM** statements which are used in what is called, a **multi-staged build** process. It allows us to create two Docker images with a single **docker build** command
- The first image is built on top of the .NET 4.7 SDK base image containing all utilities necessary to build your application: **mcr.microsoft.com/dotnet/framework/sdk:4.7.2-windowsservercore-ltsc2019**. This resulting image will be much bigger than what is required to simply run your application. This build stage will produce all application artifacts that will be picked up when building the second image. They will be copied in the folder **/app/WebAppLegacy** of the first image. Also note that this first image is identified as **build**
- The second image is built on top the .NET 4.7 runtime base image and only contains what is needed to run the application: **mcr.microsoft.com/dotnet/framework/aspnet:4.7.2-windowsservercore-ltsc2019**. It uses the output from the first image to build its own runtime image **COPY --from=build /app/WebAppLegacy/. ./** . Keeping an image as small as possible is beneficial to reduce the attack surface (less tools equals less opportunities to exploit in an attack) and they will be much faster to download at deployment time.

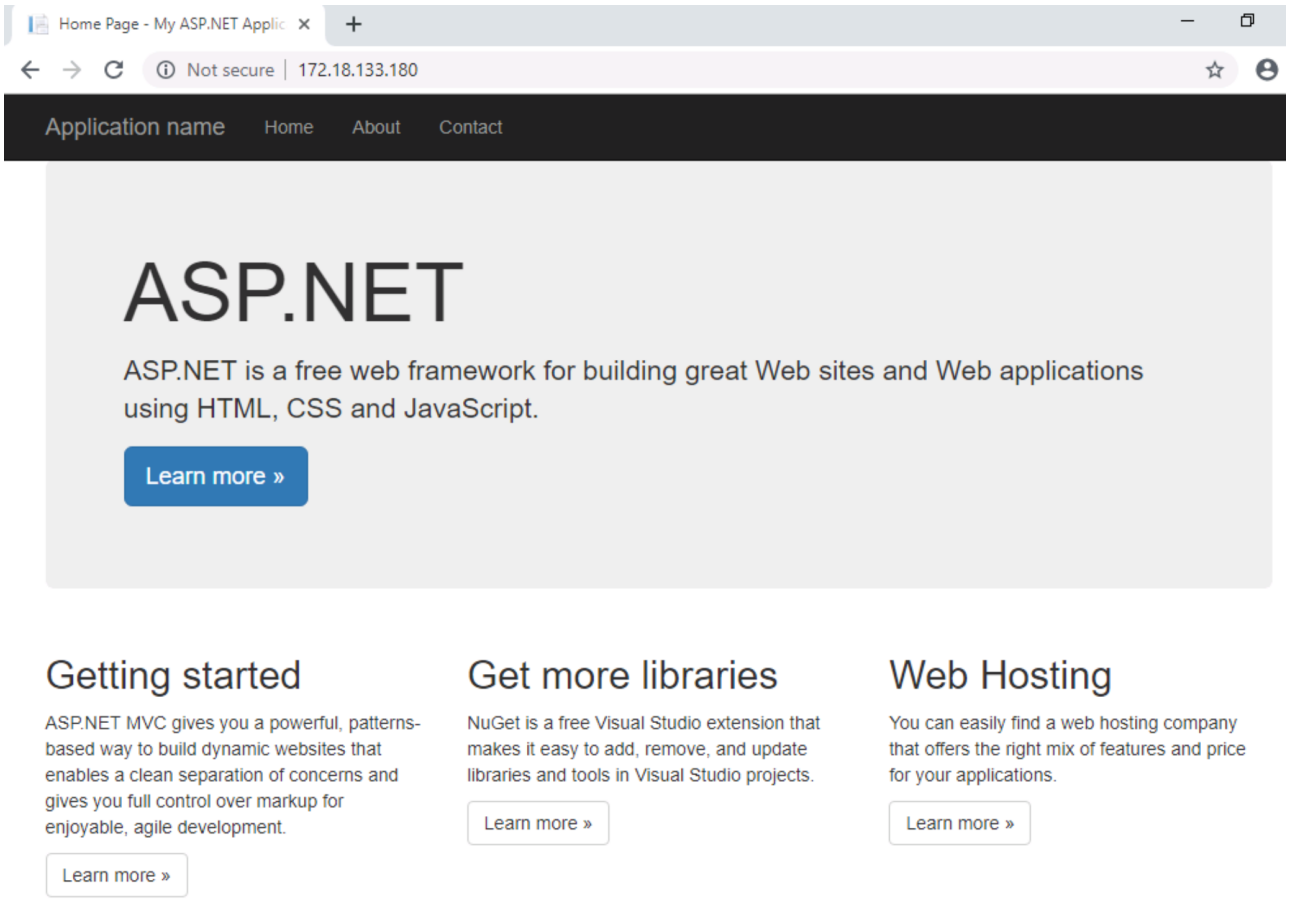
4. Build a new image with web application packaged inside it by running the command `docker build -t aspnetapp:v4.7 .`. Notice the tag **v4.7** that indicates the version of ASP.NET framework. The use of this tag is optional but recommended.

> Note: At the end of the build process, feel free to look at the produced images with **docker images**. You will see the two images that have been built, the SDK image (that is not named) and the image that is actually going to be hosting our application: **aspnetapp:v4.7**. We can automatically remove the SDK image once the runtime image is built. For that, use the **--rm** parameter in the **docker build** command

5. To run a container with the ASP.NET 4.7 web application based on the container image we just built, run the command: `docker run -d -p 8088:80 aspnetapp:v4.7`

```
PS C:\labs\module2\aspnet4.7> docker run -d -p 80:80 aspnetapp:v4.7
5ddb0f0f9e9660604bce01cfa0a003b1ff2587417ac16f1ec8a67abedc5b37d4
```

6. Start your default browser and connect to the web site running in the container using the port mapped from the host (8088) : `start http://localhost:8088`



Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 4: Building an ASP.NET Core application

In the previous task, you built container images using some of the more mature technologies and products released by Microsoft. In this task, you will build container that will run ASP.NET Core Web Application. If you completed the Module 1 lab, this will be very similar. However, we will now build the Core application on Windows instead of Linux. Furthermore, we will use the multi-stage build process rather than building the application manually with **dotnet** CLI.

ASP.NET Core is a significant step forward for Microsoft to allow ASP.NET to run across platforms including MacOS, Linux and Windows. ASP.NET sits on top of .NET Core, so it also offers cross-platform support.

> Note: To understand when to use .NET Core and when to use .NET Framework please read article: https://docs.microsoft.com/en-us/dotnet/articles/standard/choosing-core-framework-server

In this exercise, you will package a simple ASP.NET Core MVC application into a container image using a Dockerfile. Finally, you will run container hosting the ASP.NET Core application using the **docker run** command.

Return to list of exercises - Return to list of modules

Building and Running ASP.NET Core 3.x Application Inside Container

1. Change to the relevant directory using the following command: `cd C:\labs\module2\aspnetcore`

```
PS C:\labs\module2\aspnet4.7> cd C:\labs\module2\aspnetcore\
PS C:\labs\module2\aspnetcore>
```

2. You are provided with a Dockerfile. View the content of the Dockerfile by running the command `cat .\Dockerfile`. The Dockerfile should look like the one below (note this is a multi-stage Dockerfile just like the .NET 4.7 example).

```
PS C:\labs\module2\aspnetcore> cat .\Dockerfile
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-nanoserver-1809 AS build-env
WORKDIR /app

# Copy csproj and restore as distinct layers
COPY *.csproj ./
RUN dotnet restore

# Copy everything else and build
COPY . ./
RUN dotnet publish -c Release -o out

# Build runtime image
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-nanoserver-1809
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT ["dotnet", "mywebapp.dll"]
```

3. To create the container image run the command `docker build -t aspnetcoreapp:3.1 .`

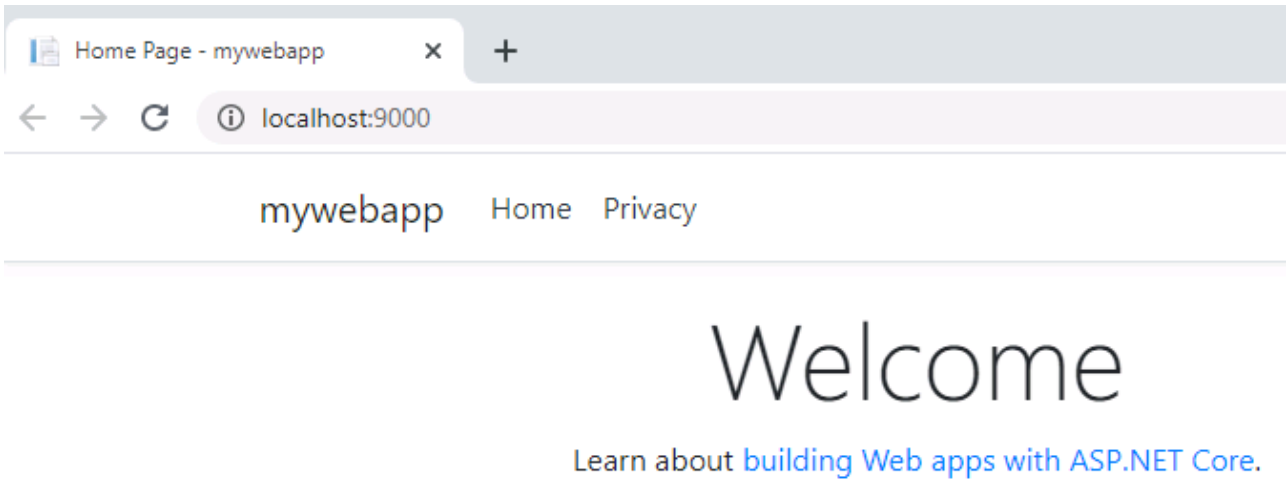
> Note: Notice the use of tag **3.1** that signifies the dotnet core 3.1 framework version

4. Launch the container running the app inside it by running the command `docker run -d -p 9000:80 aspnetcoreapp:3.1`

```
PS C:\labs\module2\aspnetcore> docker run -d -p 8080:80 aspnetcoreapp:3.1
945f238d319c05a243bfced850b5b0d88395367a25f09c58f24f40e71771128f
```

5. You are now running ASP.NET Core application inside the container listening on the port 80 which is mapped to port 9000 on the host.

6. To see the ASP.NET Core web application in action open the web browser and navigate to **localhost** port **9000** `start http://localhost:9000` . This will take you to the Home page of the Web Application.



7. Run the following command to stop and remove all containers: `docker stop (docker ps -aq) ; docker rm (docker ps -aq)`.

Congratulations!

You have successfully completed this lab. Click **Next** to advance to the next lab.

Module 3 - Docker Advanced Concepts

> Duration: 75 minutes

Module 3: Table of Contents

Exercise 1: Working with Data Volumes

In this exercise, you will learn how to mount a host directory as a data volume. The host directory will be available inside the container along with all the files (and sub directories). Later you will update a file on the host shared through a data volume from within the container. Remember that by default, data volumes at the time of mounting are read/write (unless you choose to only enable them for read only access).

Docker containers, by design, are volatile when it comes to data persistence. This means that if you remove a container, for example, using docker "rm command, all the data that was in the container (running or stopped) with be lost. This certainly causes a challenge for applications that are running in the container and

need to manage state. A good example here would be a SQL Server Database file from a previous lab that is required to be persisted beyond the life of the container running the SQL engine. The solution to this problem is to use data volumes. Data volumes are designed to persist data, independent of the container's lifecycle.

Volumes are initialized when a container is created. Some of the key characteristics of volumes are listed below:

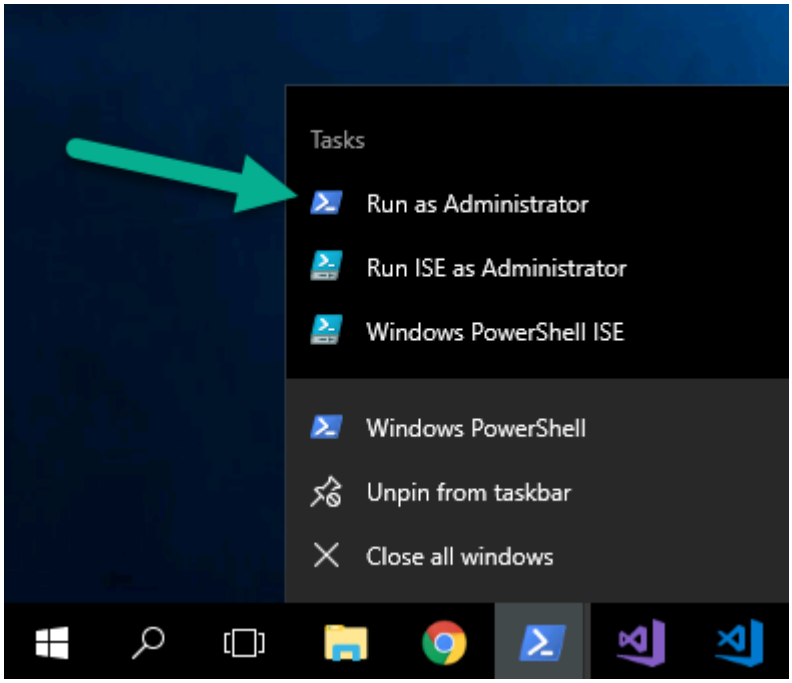
- Data volumes can be shared and reused among containers.
- Changes to a data volume are made directly by the container or the host.
- Data volumes persist even if the container itself is deleted.

> Knowledge:Docker never automatically deletes volumes when you remove a container nor will it "garbage collect" volumes that are no longer referenced by a container. This means you are responsible for cleaning up volumes yourself.

Return to list of exercises - Return to list of modules

Mount a host directory as a data volume

1. You will need to run the commands in this section using the PowerShell console as an administrator. Right click the **PowerShell** icon on the taskbar and select **Run as Administrator**.

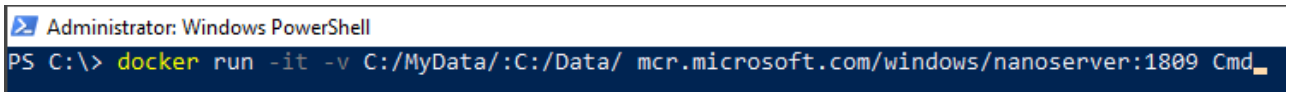


2. Navigate to your C:\ drive: `cd C:\\`
3. Create a directory on the host operating system and then add a plain text file to it. Create a new directory on the C drive by running the command `mkdir MyData`

```
PS C:\> mkdir MyData


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         5/7/2018   3:17 PM                MyData
```

4. Display the contents of the folder you just created in previous step by running the command `ls MyData`. It is currently empty.

5. You are now ready to run a container in the interactive mode and mount the host directory as a data volume. Run the command `docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 CMD`



> Knowledge: Notice the **-v** switch that is required to mount the host directory **C:\MyData** inside the container as **C:\Data**. This will result in container access to contents of **C:\MyData** on the host inside the container as **C:\Data**. You can choose same name for the directory inside the container and host but it's not mandatory as you see in the above command (**C:\MyData** on the host and **C:\Data** inside the container)

6. On the container PowerShell Console first check the hostname by running the command `hostname`.

> Note:The actual hostname for your container may be different than pictured below. Most importantly though, the container hostname will be different from you VM hostname.

```
PS C:\> hostname
3a45f5770238
```

7. List the directories by running the command `dir`.

> Note:Notice the **data** directory as part of the listing.

```
PS C:\> dir


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----l         10/2/2018  11:28 AM                data
d-----         10/2/2018  11:29 AM                Program Files
d-----         7/16/2016   5:09 AM                Program Files (x86)
d-r---         10/2/2018  11:29 AM                Users
d-----         10/2/2018  11:29 AM                Windows
-a----        11/20/2016   3:32 AM           1894 License.txt
```

8. Create a file in the folder and add more text to it. Run the command: `echo File is updated by container: %COMPUTERNAME% >> c:\data\file.txt`

```
C:\>echo File is updated by container: %COMPUTERNAME% >> C:\data\file.txt
```

> Note: **%COMPUTERNAME%** is equivalent to **hostname**

9. You can access and update the content of the data directory. First, run the command `dir c:\data`

> Knowledge: This will list the content structure residing inside the data directory.

> Note:Notice **file.txt** is present inside the data directory. This is the same file you created earlier on the host.

```
PS C:\> dir c:\data


    Directory: C:\data


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        10/2/2018  11:27 AM             88 file.txt
```

10. Look at content inside the **file.txt** by running the command `more c:\data\file.txt`

```
C:\>more c:\data\file.txt
File is updated by container: 225D265D584B
```

11. You can now exit the container and return to host by running the command `exit`

```
PS C:\> exit
```

12. On the host PowerShell Console run the command `gc C:\MyData\file.txt`. gc stands for **Get-Content**.

> Note:Notice that changes made from the container persist on the host by the **file.txt**.

```
PS C:\> gc C:\MyData\file.txt
File created on the host: WIN-EJI3ACSSVHL
File is updated by container: 3a45f5770238
```

13. Run `docker ps -a` to get the ID of stopped containers, and then record the container ID in the following text box:

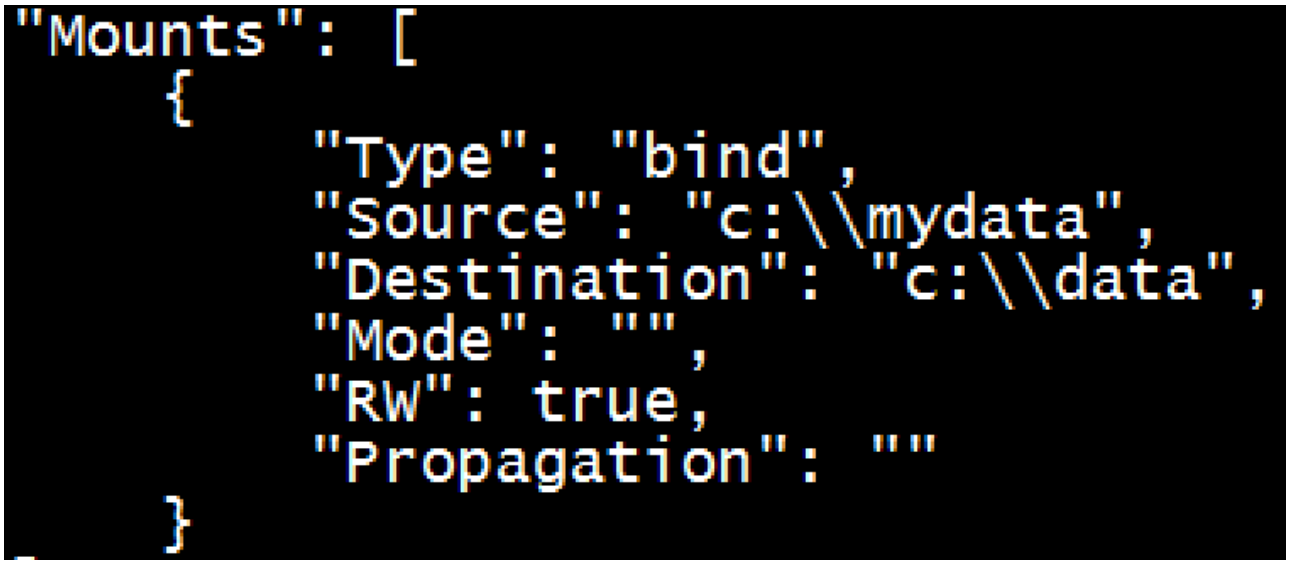
Container ID
@lab.TextBox(ContainerID2)

> Note:To gather more information about container and volumes that has been mounted you can run the command `docker inspect <ContainerID2>`.

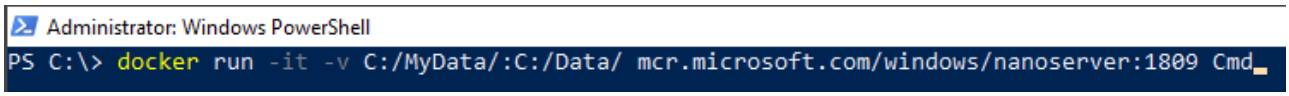
```
PS C:\> docker inspect 3a45f5770238
```

14. The docker inspect command outputs a rather large JSON file on the display. You may need to scroll down to find the section labeled "Mounts".

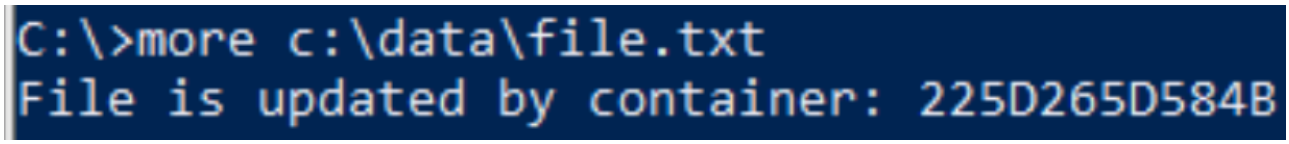
    > Note:Notice that **c:\mydata** is the source and **c:\data** is the destination. Also, RW refers to Read/Write.



15. Let's run another container in interactive mode and mount the host directory as a data volume. Run the command `docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 Cmd`

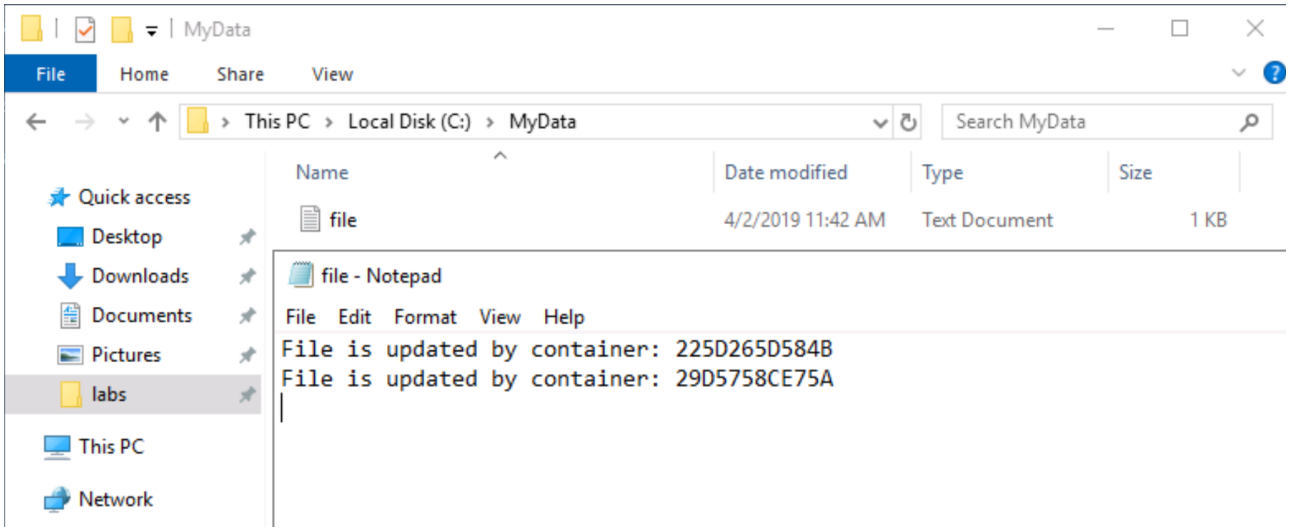


16. Look at content inside the **file.txt** by running the command `more c:\data\file.txt`

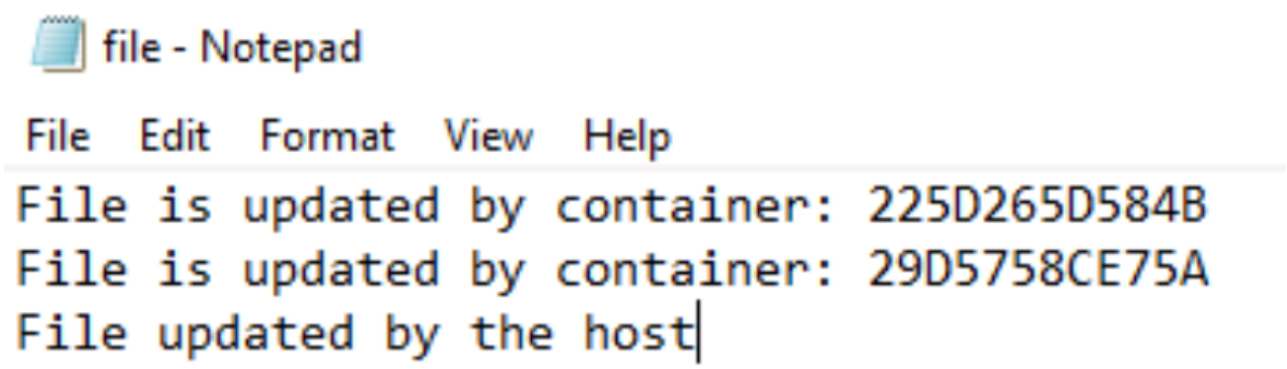


17. Add more text to it. Run the command: `echo File is updated by container: %COMPUTERNAME% >> c:\data\file.txt`

18. On the host machine, go to **C:\MyData** from the file explorer and open **file.txt**

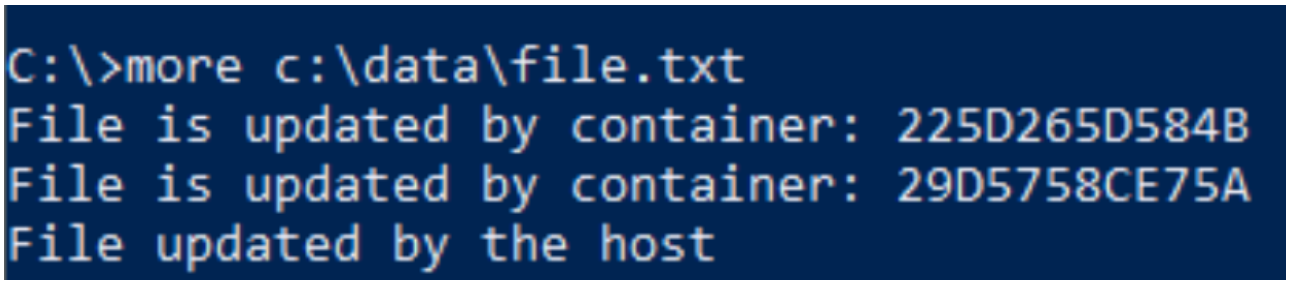


19. Update the content of the file with notepad and save it.



> Note: The two different **hostnames** correspond to the two Ids of the containers that wrote in the file.

20. Go back to the Powershell windows and check that the container can see the host changes with the command `more c:\data\file.txt`



> Note: Because of concurrency challenges, you would probably not have multiple containers and hosts writing in the same file. The purpose of this exercise was only to show how we can persistent data across containers beyond their short lifecycle.

21. Finally, you can run `exit` to stop the running containers

Mount a shared-storage volume as a data volume

In the previous section you learn how to mount a directory on the host as a data volume. That's a very handy way to share the content from host to container but it's not ideal in terms of portability. Basically, if you later run the container on a different host there is no guarantee that host will have the same directory. This would cause the app inside the container to break as it depends on a share that is not implemented by the host. In cases like these when a higher level of portability is desired, you can mount a *shared storage volume*. Docker has some volume plugins that allow you to provision and mount shared storage, such as iSCSI, NFS, or FC. A benefit of using shared volumes is that they are host-independent. This means that a volume can be made available on any host on which a container is started as long as the container has access to the shared storage backend, and has the plugin installed.

In this exercise, you will learn how to create and use a shared-storage volume. To keep the lab accessible and easy to follow, you will use the *local* driver which uses local host for the storage. However, the exact same concepts will work against production ready storage drivers like Convoy and others. For more information on the Convoy volume plugin, please visit: https://github.com/rancher/convoy

Return to list of exercises - Return to list of modules

1. First, let's create a volume by running the command following command from a Powershell window `docker volume create -d local myvolume`

```
docker volume create -d local myvolume
```

2. You can list all the volumes by running the command `docker volume ls`. Notice that **myvolume** is available as a local driver.

```
PS C:\> docker volume ls
DRIVER              VOLUME NAME
local               myvolume
```

3. You can use **docker inspect** command with the volumes too. Run the command `docker inspect myvolume`

```
PS C:\> docker inspect myvolume
[
    {
        "CreatedAt": "2019-04-02T12:18:08-07:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "C:\\ProgramData\\docker\\volumes\\myvolume\\_data",
        "Name": "myvolume",
        "Options": {},
        "Scope": "local"
    }
]
```

> Note: **Mountpoint** is set at a location on the **C** drive under the **ProgramData\docker** folder. This is the default location for local storage drivers. If you use another commercial storage driver, the location may be different.

4. To launch a container and make that storage volume available inside the container run the command `docker run -it -v myvolume:C:/Data/ mcr.microsoft.com/windows/servercore:1809 powershell`. This command is like the command from last section where you shared the host directory,

except that within the **-v** switch you are using the name of the storage volume rather than the path to the host directory.

5. On the PowerShell command prompt inside the container, run the command `dir` to list the directories available in the container.

```
PS C:\> dir


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         4/2/2019  12:18 PM                data
d-r---         3/8/2019   7:11 PM                Program Files
d-----         3/8/2019   7:09 PM                Program Files (x86)
d-r---         3/8/2019   7:11 PM                Users
d-----         4/2/2019  12:34 PM                Windows
-a----        9/15/2018   2:42 AM           5510 License.txt
```

6. Notice the data directory. You can now add/remove files to it. Let's create a new text file and add text content to it. On the command prompt run the command `"File created on the host: $(hostname)" >> c:\data\sample.txt`.

```
PS C:\> "File created on the host: $(hostname)" >> C:\data\sample.txt
```

7. Confirm that file sample.txt has been created successfully by running the command `more c:\data\sample.txt`

```
PS C:\> more C:\data\sample.txt
File created on the host: bc4686e2a603
```

8. Now exit the container by running the command `exit`. This will take you back to PowerShell console on the host.

9. To check the content of sample.txt file from the host run the command `gc C:\\ProgramData\\docker\\volumes\\myvolume\\_data\sample.txt`.

```
PS C:\> gc C:\\ProgramData\\docker\\volumes\\myvolume\\_data\sample.txt
File created on the host b1e8ef12d185
```

Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 2: Working with Docker-Compose

##Challenges with Multi-Container Applications

When working with multi-containers applications, you will need to make sure that applications can discover each other in a seamless fashion. Consider a quite common scenario where a web application (that acts as a front-end) calls to a backend RESTful web API to fetch content. In this scenario, the web app would need to access the web API in a consistent fashion. In addition, due to the fact the web application has a dependency on the web API, that dependency must be expressed when launching the applications in containers. It is imperative that we are able to launch and test a multi-container application the same way across development, test and production environments.

Docker has provided a tool called **"docker-compose"** that enables you to describe your applications as services within a YAML file, docker-compose.yml. A service in this context really means, "a container in production". A service only runs one image, but it codifies the way that image runs - what ports it should use, how many replicas of the container image should run (so that the service has the capacity it needs), and so on. Scaling a service increases the number of container instances running the application image, assigning more computing resources to that service in the process.

##Working with Docker-Compose

In this exercise, you will work with a simple "Famous Quotes" application that is composed of a frontend web app that talks to a RESTful API to fetch quotes in JSON format. Both the web app and API are developed using ASP.NET Core and each will run in a separate container. As this is a multi-container scenario, you will use a docker-compose file to:

- Ensure the web API can be accessed by the web app without the need to hardcode its FDQN or IP Address. Instead of hardcoding the IP Address (or FDQN) you can use a docker-compose.yml file to make these services discoverable

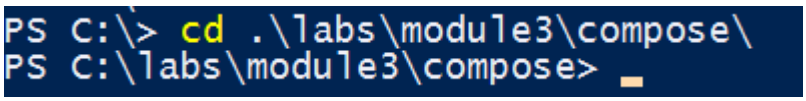
  > Note: Recall from the previous lab where a web application needed to access SQL Server running in a separate container. In that situation, we provided the web application the IP Address of the container running SQL Server in the web.config configuration file.

- Express specific dependencies, such as the web app container **depends on** the web API container
- Get both application components up and running in separate containers with a single command (i.e., without using individual docker-run commands for each container).

Return to list of exercises - Return to list of modules

Running Multi-Container applications using Docker Compose

1. Launch the **PowerShell Console** (if not already running) and change your current directory to "**compose**" folder by running the command `cd C:\labs\module3\compose`

2. Before proceeding further let's stop all the running containers from previous task. Run the command `docker stop (docker ps -aq)`

```
PS C:\labs\module3\compose> docker stop (docker ps -aq)
b1e8ef12d185
cd0c78cf4daa
11573e615450
aa6f82318f61
89f20498f798
d08522c6d142
```

3. First, look at directory structure by running the command `dir`.

```
PS C:\labs\module3\compose> dir


    Directory: C:\labs\module3\compose


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         5/1/2018   5:05 PM                mywebapi
d-----         5/1/2018   5:06 PM                mywebapp
-a----         5/1/2018   5:02 PM            253 docker-compose.yml
```

4. Notice that you have two folders "**mywebapi**" and "**mywebapp**" representing the web API and web application, respectively. First, you will inspect the piece of code that is making the RESTful call to mywebapi. To do that run the command: `gc .\mywebapp\Controllers\HomeController.cs`

```
gc .\mywebapp\Controllers\HomeController.cs
```

5. This displays the code within HomeController.cs file. You may need to scroll down to view the code that calls the mywebapi RESTful endpoint. The actual URI is http://demowebapi:9000/api/quotes.

> Note:Notice the use of **demowebapi** which is not a FQDN nor IP Address, but rather a service that is defined within the **docker-compose.yml** file (which we will review next). By using the service name, the web application can simply refer to the Web API app (using that same name) across all environments, including development, test and production etc.

```
await client.GetStringAsync("http://demowebapi:9000/api/quotes");
```

6. Let's inspect the **docker-compose.yml** file. Run the command `gc .\docker-compose.yml`

```
gc .\docker-compose.yml
```

```
version : '3'

services:
  demowebapp:
    build: ./mywebapp
    ports:
      - 80:80
    depends_on:
      - demowebapi
  demowebapi:
    build: ./mywebapi
    ports:
      - 9000:9000
networks:
  default:
    external:
      name: nat
```

> Knowledge: First, notice the structure of the file. All .YML files follow the YAML structure (more information about the extension can be found at https://www.reviversoft.com/file-extensions/yml. For docker compose usage you first define the version number and then specify the structure of your services. In this case, we have two services, namely "*demowebapp*" and "*demowebapi*". The demowebapp service declaration starts with the build instruction and points to folder "*mywebapp*" that contains the ASP.NET core application and relevant Dockerfile (recall the file entitled, DockerFile, that resides in the root of the application). Note how the compose file contains sections, or "instructions": Services, networks, etc. The build instruction is equal to the *docker build* command. Then ports are mapped from the host's port 80 to the container's port 80.The *depends_on* directs the docker-compose to launch the *demowebapi* container first since *demowebapp* depends on it. Also, the discoverability is done by using the service names (as mentioned in the paragraph above whereas, *demowebapp* can access *demowebapi* by its service name, rather than FDQN or IP Address).
>
> Next is the *demowebapi* service declaration. It also starts with the build command pointing to the "mywebapi" folder that contains the Dockerfile and relevant ASP.NET Core files. Ports are mapped from host port 9000 to container port 9000.
>
> Finally, the networks section keeps the default settings to nat networking. This network declaration is needed for windows containers now. Basically, it tells docker compose to use default nat networking.

Docker Compose Up

1. We have pre-downloaded the docker-compose.exe file for you onto the VM, if you would like to see it, you can see the URL here: **https://github.com/docker/compose/releases/download/1.12.0/docker-compose-Windows-x86_64.exe**

2. At this point, you are all set to run the multi-container application with a single command `docker-compose.exe up -d`

```
PS C:\labs\module3\compose> docker-compose.exe up -d
```

> Knowledge: The docker-compose.exe tries to make it simple to start and stop the services (running containers) with commands like up and down. The **-d** switch works the same as when used with the docker build command, which instructs docker to run the container in the background rather than interactively. If you don't provide any switch parameter, the default is set to interactive.

> Note:As the command executes, you will notice that the "mywebapi" container is built first. This is because we mention in the yml file that "mywebapp" depends on it, so it will build first. Also, if the image for "mywebapi" already exists, then it won't be built again.

```
Building demowebapi
Step 1/12 : FROM mcr.microsoft.com/dotnet/core/sdk:3.1-nanoserver-1809 AS build-env
 ---> fb60591689e2
Step 2/12 : WORKDIR /app
 ---> Using cache
 ---> 5f4d8aba7c76
Step 3/12 : COPY *.csproj ./
 ---> ff9b8f134183
Step 4/12 : RUN dotnet restore
 ---> Running in ccb6d75f7e4a
  Restore completed in 317.34 ms for C:\app\mywebapi.csproj.
```

> Note:Next, Docker will build the container image for "mywebapp."

```
Building demowebapp
Step 1/10 : FROM mcr.microsoft.com/dotnet/core/sdk:3.1-nanoserver-1809 AS build-env
 ---> fb60591689e2
Step 2/10 : WORKDIR /app
 ---> Using cache
 ---> 5f4d8aba7c76
Step 3/10 : COPY *.csproj ./
 ---> aafa8fe68b08
Step 4/10 : RUN dotnet restore
 ---> Running in e28e75cac2b6
  Restore completed in 328.01 ms for C:\app\mywebapp.csproj.
Removing intermediate container e28e75cac2b6
```

> Note:You can safely ignore any warnings.

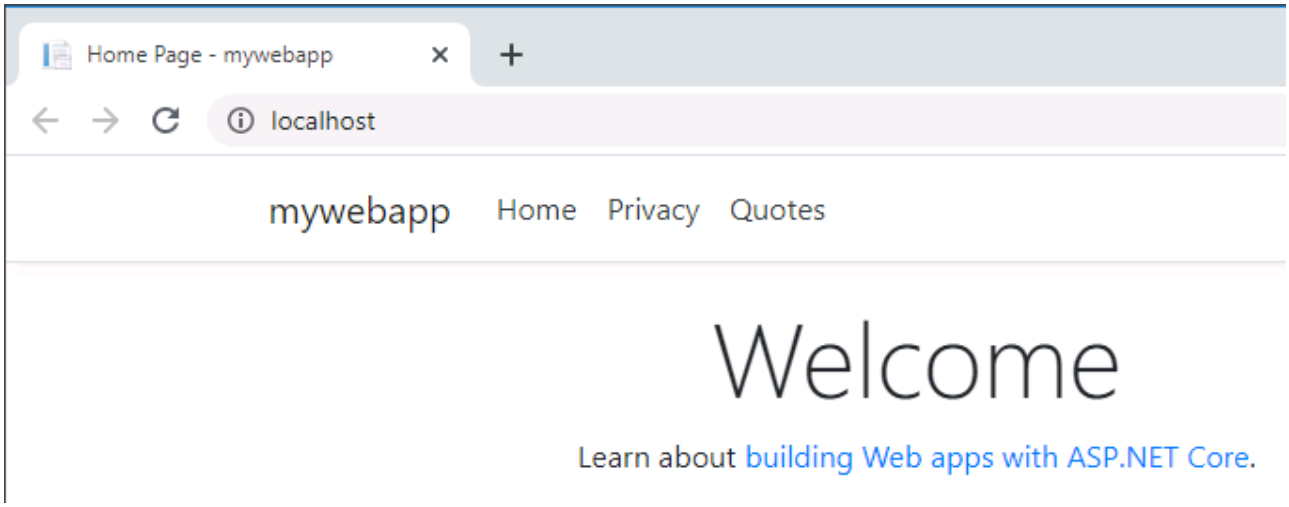
3. Finally, docker-compose will run both containers using the instructions from the docker-compose.yml file.

```
Creating compose_demowebapi_1
Creating compose_demowebapp_1
```

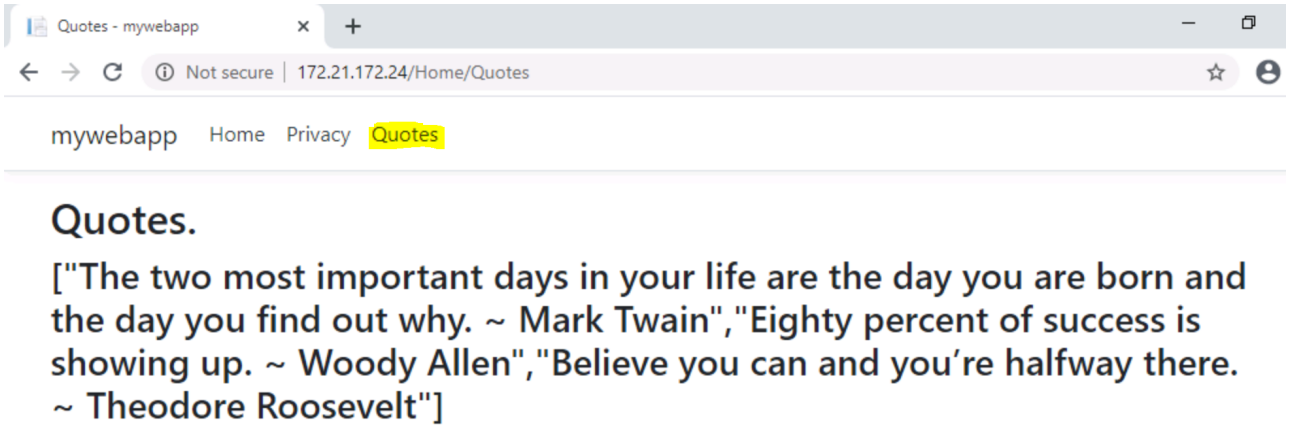
4. You can check details about running docker cmpose services by executing the command `docker-compose ps`

```
PS C:\labs\module3\compose> docker-compose ps
        Name                      Command         State           Ports
-------------------------------------------------------------------------------
compose_demowebapi_1   dotnet mywebapi.dll   Up      0.0.0.0:9000->9000/tcp
compose_demowebapp_1   dotnet mywebapp.dll   Up      0.0.0.0:80->80/tcp
```

5. Open web browser of your choice and browse to localhost `start http://localhost`. You should land on the home page of web application as shown below.



> Note:To test the Web API you will can select the **Quotes** option from the top menu bar. This will result in a call to web API and results being displayed on the web application.



Docker Compose Down

When you wish to stop and remove the multi-container application that was launched by docker compose, you will use docker-compose down command. The down command safely stops and removes all the containers that were launched by the up command earlier using the docker-compose.yml file.

> Knowledge: If you only wish to stop the multi-container applications and associated running containers use "**docker-compose stop**" command instead. This command will stop the containers, but won't remove them.

Return to list of exercises - Return to list of modules

1. On the PowerShell console run the command `docker-compose down`

   > Note: First the containers are stopped and then they are removed.

Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 3: Docker Networking

In this exercise, you will work with various PowerShell and Docker CLI commands to view Docker default networks and create a custom nat network. Finally, you will remove the custom nat network and observe how Docker responds by creating a new default nat network automatically.

Return to list of exercises - Return to list of modules

Display all Docker networks

You can retrieve container networks using the Docker CLI.

1. Docker provides native docker command that provides list of networks available to docker. To view the list of networks available to docker run the command `docker network ls`.

```
PS C:\labs\module3> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
9699e7c25af5        nat                 nat                 local
39eccb4ad307        none                null                local
```

   > Knowledge: The 'nat' network is the default network for containers running on Windows. Any containers that are run on Windows without any flags or arguments to implement specific network configurations will be attached to the default 'nat' network, and automatically assigned an IP address from the 'nat' network's internal prefix IP range. The default nat network also supports port forwarding from container host to internal containers. For example, you can simply run SQL Server Express in a container by providing the "p" flag so that specified port numbers will be mapped from host to container.

2. To view detail information about the Docker default nat network, run the command `docker inspect nat`

   > Note:Notice the output is in JSON format. The "Containers" key (which is empty in this case) refers to all containers that are using the specified network. The containers key is empty in this case because there are no containers currently running.

```
PS C:\labs\module3> docker inspect nat
[
    {
        "Name": "nat",
        "Id": "9699e7c25af5aa747db8e9ad244f375eb086030f8b327383da8247e906af5911",
        "Created": "2019-04-01T14:17:49.5199566-07:00",
        "Scope": "local",
        "Driver": "nat",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "windows",
            "Options": null,
            "Config": [
                {
                    "Subnet": "0.0.0.0/0"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "4c23a79396d5f07ca5c3b0f2f969d383952a081f3ea2d6afe6a62cc1dafd0c46": {
                "Name": "thirsty_ritchie",
                "EndpointID": "2a02165f83a7f6742c8c2b43f5f473f06a69e3049367a20a45dc38a170ed8712",
                "MacAddress": "00:15:5d:46:57:ea",
                "IPv4Address": "172.30.189.187/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.windowsshim.hnsid": "9713D340-4450-4991-99F6-729243AA4B06",
            "com.docker.network.windowsshim.networkname": "nat"
        },
```

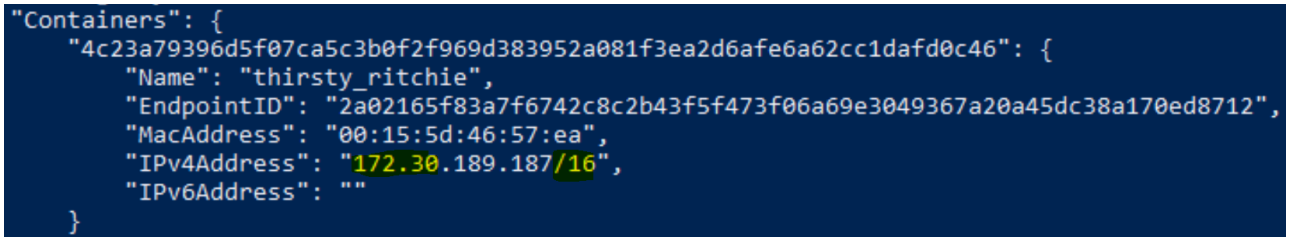
3. Run `ipconfig` to see the two networks: the physical network, localdomain, and the local container network, nat.

```
PS C:\labs\module3> ipconfig

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::4cfc:26d9:421f:70c0%12
   IPv4 Address. . . . . . . . . . . : 192.168.1.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1

Ethernet adapter vEthernet (nat):

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::81b:4dcd:9020:5b2d%18
   IPv4 Address. . . . . . . . . . . : 172.30.176.1
   Subnet Mask . . . . . . . . . . . : 255.255.240.0
   Default Gateway . . . . . . . . . :
```

4. Launch a new container by running a command `docker run -d mcr.microsoft.com/windows/nanoserver:1809 ping -t localhost**+++. Once the`

container is running execute the command +++**docker inspect nat

> Note:Notice that this time the "Containers" section now includes information pertaining to the container that is using the nat network including its ID and IPv4 address.



Create a custom Docker nat network

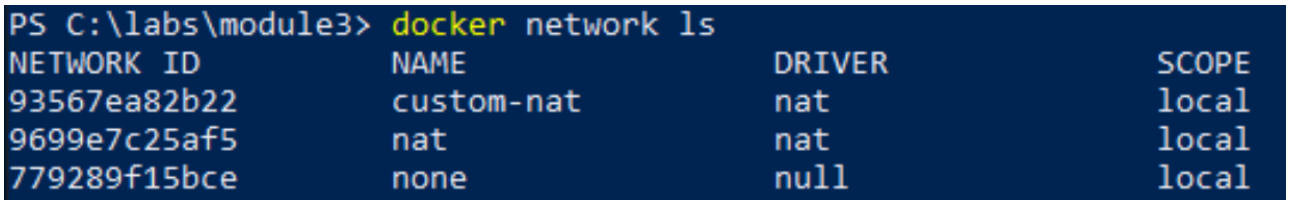
Docker allows you to create custom nat networks. In this task, you will create and configure a custom nat network replacing the default nat network.

Return to list of exercises - Return to list of modules

1. Create a new docker network by running the command `docker network create -d nat --subnet=192.168.15.0/24 --gateway=192.168.15.1 custom-nat`

   > Note:The **"d"** flag stands for network driver and specifies the network type you want to create which in this case is "nat". You are also providing the IP prefix and gateway address using the -subnet and -gateway flags.

2. Use the **+++docker network ls**+++ command and notice that "**custom-nat**" network is available.



3. To use the new custom nat network for containers launch a new container by using the command `docker run -d --network=custom-nat mcr.microsoft.com/windows/nanoserver:1809 ping -t localhost`

   > Note:Notice the use of --network switch which allows you to force docker to use specific network for the container.

4. Now, use the **+++docker network inspect custom-nat**+++ command to get the detailed information about custom-nat network and container(s) that is using it.

   > Note:Notice the subnet and gateway values reflect the values you used earlier during the creation of the network. Also note that the container's IPv4 Address, 192.168.15.224 (may be different in your case), is in the custom-nat network.

```
docker inspect custom-nat

 {
     "Name": "custom-nat",
     "Id": "93567ea82b2226008ddbf72b6188ecffb9ab03ab0ee1b422cce2c9cc9fd347b4",
     "Created": "2019-04-02T14:58:15.4008894-07:00",
     "Scope": "local",
     "Driver": "nat",
     "EnableIPv6": false,
     "IPAM": {
         "Driver": "windows",
         "Options": {},
         "Config": [
             {
                 "Subnet": "192.168.15.0/24",
                 "Gateway": "192.168.15.1"
             }
         ]
     },
     "Internal": false,
     "Attachable": false,
     "Ingress": false,
     "ConfigFrom": {
         "Network": ""
     },
     "ConfigOnly": false,
     "Containers": {
         "16a51a109c0c67c813d28bd3109adc8c4bf96b348340a7d45691dbfbd3d41e81": {
             "Name": "agitated_pascal",
             "EndpointID": "a8f4e4007577ee6ddf5856c67240057362bc3e2ba311fac35d6cae094f0679ae",
             "MacAddress": "00:15:5d:4d:0f:f1",
             "IPv4Address": "192.168.15.108/24",
             "IPv6Address": ""
         }
     },
```

5. To confirm that the container host and access container run the command `ping <Container - IPv4 Address>`

> Note: You can look for container's IP Address in the output from previous command. Notice that the host can successfully access the container using its IP.

```
PS C:\labs\module3> ping 192.168.15.108

Pinging 192.168.15.108 with 32 bytes of data:
Reply from 192.168.15.108: bytes=32 time<1ms TTL=128
Reply from 192.168.15.108: bytes=32 time<1ms TTL=128
Reply from 192.168.15.108: bytes=32 time<1ms TTL=128
```

6. Now let's start a new container on the nat network and open a command prompt `docker run -it --network=nat mcr.microsoft.com/windows/nanoserver:1809 cmd`

7. We can try to ping the previous container with `ping <Container - IPv4 Address>` where the IP Address is the same one we just pinged from the host. Hit **Ctrl-C** to stop the ping operation.

```
C:\>ping 192.168.15.108

Pinging 192.168.15.108 with 32 bytes of data:
Request timed out.

Ping statistics for 192.168.15.108:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
Control-C
```

> Note: Because the two containers are on separated networks, they cannot ping each other using their IP Address.

8. Run `ipconfig**+++ from the container to check the that IP Address belongs to the nat network. Then run +++**exit` to go back to the host.

```
C:\>ipconfig

Windows IP Configuration


Ethernet adapter vEthernet (Ethernet) 2:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::9577:f0c0:2703:2746%41
   IPv4 Address. . . . . . . . . . . : 172.30.180.9
   Subnet Mask . . . . . . . . . . . : 255.255.240.0
   Default Gateway . . . . . . . . . : 172.30.176.1
```

9. Remove all containers so that you can then remove the custom network you have created (if containers are still attached to the network, the network deletion will fail). `docker rm (docker ps -aq) -f`

10. You may now remove the **custom-nat** network `docker network rm custom-nat`

```
PS C:\labs\module3> docker network rm custom-nat
custom-nat
```

11. Check that only nat network remains `docker network ls`

```
PS C:\labs\module3> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
9699e7c25af5        nat                 nat                 local
779289f15bce        none                null                local
```

Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 4: Running containers with memory and CPU constraints

By default, when a container is run, it has no resource constraints. Without constraints, a container can use as much of a given resource as the host's kernel scheduler will allow. Docker enables you to control how much memory, CPU, and/or block IO a container can consume by setting runtime configuration flags. In this exercise, you will run a container with resource constraints. To follow best practices, you should always establish resource constraints on your containers.

Return to list of exercises - Return to list of modules

Run a container with memory constraints

In this task, you will launch a container with a pre-defined memory limit, to ensure the container does not consume the host's memory beyond the memory limit. Later, you will test the container memory limit by simulating higher memory consumption inside the container.

Return to list of exercises - Return to list of modules

1. Before running the container open two new **PowerShell** Consoles

   > Note:You will use one of these consoles to run the docker container and interact with it. The other console will be leveraged to monitor the memory usage of the container.

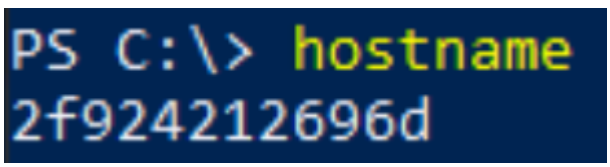
2. Use one of the open **PowerShell** consoles to launch a new container with a memory limit of 500 megabytes (MB) by running the command `docker run -it -m 500M --mount 'type=bind,source=C:/labs/module3/tools/,target=C:/tools/' mcr.microsoft.com/windows/servercore:1809 powershell`

   > Note:Notice the use of -m (or --memory) switch within the run command. The switch specifies the maximum amount of memory the container can use. In this case, you are setting it to 500 M (where M = Megabytes). Other valid options are B = Bytes, K= Kilobytes and G = Gigabytes. These are also not case sensitive).

   > Knowledge:The use of -m switch is not related to memory but rather to bind mounting the tools folder on from the host into the container. This folder containes a Sysinternals tool "TestLimit64.exe" which you will be using next to test the container memory limit.

3. You should now have access to **PowerShell** console that is running inside the container. Run the `hostname` command and note the name of the container. You will need it later in this task.

   Container ID
   @lab.TextBox(ContainerID3)

4. To test the memory limit of container you will use the **testlimit** tool. Run the following command `C:\tools\testlimit64.exe -d -c 1024`

```
PS C:\> C:\tools\testlimit64.exe -d -c 1024

Testlimit v5.04 - test Windows limits
By Mark Russinovich - www.sysinternals.com

Leaking private bytes with touch (MB)...
Leaked 379 MB of private memory (379 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 0 MB of private memory (379 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 0 MB of private memory (379 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Sleeping for 5 seconds to allow for paging file expansion...
```

> Note:The tool will attempt to push the memory consumption of the container to 1024 MB (1 GB). However,because the container can't go beyond 500 MB, the value of memory consumed will always be under 500 MB (the exact value of how much memory is used will vary but won't go beyond the maximum available memory on container which is 500 MB)

5. Go back to the **PowerShell** console on the host (this is the second console that you opened earlier). Run the docker stats command `docker stats <ContainerID3>`

> Note: This command gives you a live stream of various vital stats including memory, CPU, etc. directly from the container.

6. Notice the value under the column "**PRIV WORKING SET**". This represents the memory usage by the container; this is the value that docker has constrained to 500 MB.

```
CONTAINER ID        NAME                CPU %               PRIV WORKING SET
2f924212696d        eager_lalande       0.00%               454.1MiB
```

7. Now, you will reclaim the memory occupied by the running tool. Go back to the **PowerShell** console that was used to run the container and press the key combination "**Ctrl+C**".

> Note:This will stop the tool and free the memory on the container used by this tool.

8. Go back on the **PowerShell** console on the host that is displaying the vital stats for the container.

> Note:Notice that memory usage has dropped significantly.

```
CONTAINER ID        NAME                CPU %               PRIV WORKING SET
2f924212696d        eager_lalande       0.00%               72.53MiB
```

9. Hit "**Ctrl+C**" to stop the docker stats command

10. In the other **Powershell** window, type `exit` to exit the running container

Run a container with a CPU usage limit

In addition to setting a memory constaint, you can also constrain the CPU usage by the container. By default,Docker does not apply any constraint on container CPU usage, which essentially means the container is free to utilize host CPU up to 100%. In this task, you will put a limit on CPU utilization by the container.

Return to list of exercises - Return to list of modules

1. Since modern machines have CPUs with multiple cores you will first determine the number of cores available to the host virtual machine by running the command `Get-WmiObject –class Win32_processor | Select -ExpandProperty NumberOfCores`

   > Note:Take note of the number of cores available. In this case there are 2 cores, but the value you see may differ.

```
PS C:\> Get-WmiObject -class Win32_processor | Select -ExpandProperty NumberOfCores
4
```

2. You will now launch a new container and limit its host CPU utilization to ~25%. Make sure that you set half of the availble CPU here. If your machine just has 4 cores, set the value to 1.

   ```
   docker run -it --cpus 1.0 --mount
   'type=bind,source=C:/labs/module3/tools/,target=C:/tools/'
   mcr.microsoft.com/windows/servercore:1809 powershell
   ```

   > Note:Notice the use of --cpus switch which will specify how much of the available CPU resources the container can use. For example, the host machine has two CPUs and if you set --cpus to 1.0, the container will be able to access, **at most**, one of the CPUs on the host.

3. You should now have access to PowerShell console that is running inside of the container. Run the `hostname` command and take note of the container name. You will need it later in this task.

   Container hostname @lab.TextBox(ContainerID4)

4. Go back to the **PowerShell** console on the host (this is the second console that you opened earlier). Run the docker stats command `docker stats <ContainerID4>`

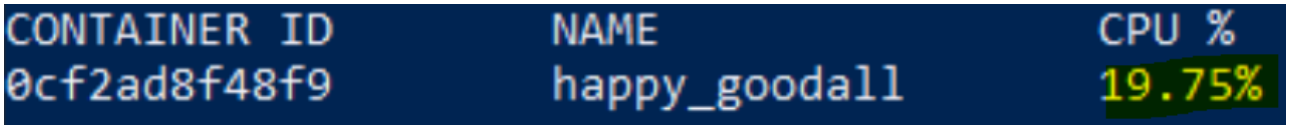
```
CONTAINER ID        NAME                CPU %               PRIV WORKING SET
0cf2ad8f48f9        happy_goodall       0.00%               83.31MiB
```

5. Go back to the **PowerShell** console in the container and make sure that you are authorized to run PS1 scripts by running the following command `Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass`

6. You will now test the CPU constraint of ~25% by stress testing the CPU utilization on the container. Switch back to the **PowerShell** console that you used earlier to launch the container with a CPU usage limit. Execute the command `C:\tools\cpu-stress.ps1`

```
PS C:\> Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
PS C:\> C:\tools\cpu-stress.ps1
Number of cores to target:  4

Id     Name            PSJobTypeName   State         HasMoreData
--     ----            -------------   -----         -----------
1      Job1            BackgroundJob   Running       True
3      Job3            BackgroundJob   Running       True
5      Job5            BackgroundJob   Running       True
7      Job7            BackgroundJob   Running       True
```

> Note:This command executes the script on the container which stress tests CPU, targeting all cores available to the container. However, because you set a constraint on CPU utilization, the container will never able to consume more than 1 CPU.To validate the CPU usage, go back to PowerShell console displaying **docker stats**. Notice the container CPU utilization is ~25% and not 100%. The CPU utilization may be slightly lower or higher than 25%, so it may not be exact.



Congratulations!

You have successfully completed this lab. Click **Next** to advance to the next lab.

Module 5 - Container Orchestrators in Azure

Duration

> 90 minutes

Module 5: Table of Contents

Exercise 1: Create and Scale Azure Kubernetes Service (AKS) Cluster

In this exercise, you will create a Kubernetes cluster using Azure Kubernetes Service and deploy a NGINX container image to that cluster. The container image will be pulled from the Docker Hub public registry. You will also learn how the application can be scaled via the command line or kubernetes dashboard.

Return to list of exercises - Return to list of modules

1. Navigate to +++@lab.CloudPortal.Url+++ and sign in with your **Azure** account credentials found on the Resource Tab.

   +++@lab.CloudPortalCredential(User1).Username+++

   +++@lab.CloudPortalCredential(User1).Password+++

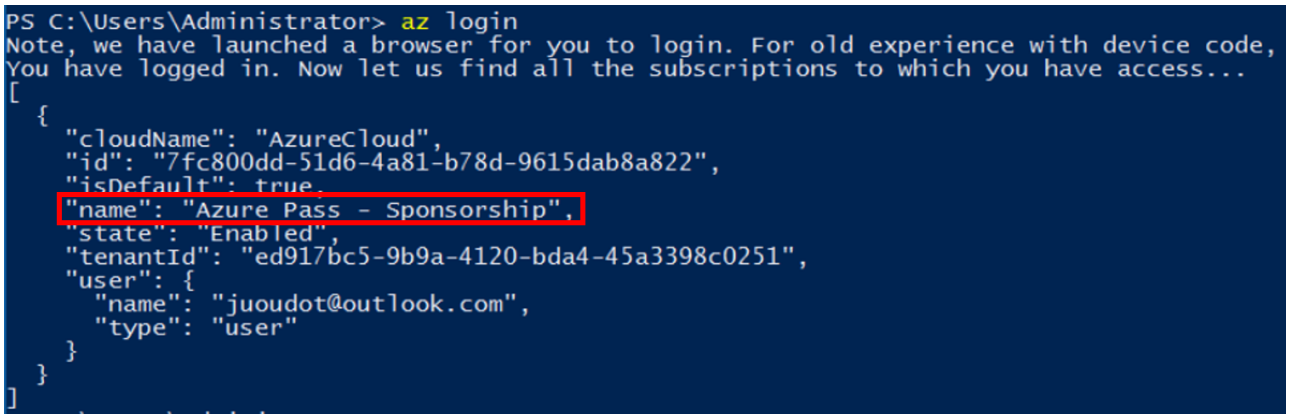
2. On your Windows server VM, open **PowerShell**.

   > Note: Do not use Powershell ISE, as the IDE won't work as expected with the **az** CLI

3. Login to **Azure** using `az login`

> Note:The command line will automatically open your browser to login. Once you are logged in, you can close the browser and return to the command line.

4. If you have multiple subscriptions attached to your account, you will see a list of them displayed post-login. To select the correct subscription, put the name of the subscription in quotes. You can find the **name** field in the output of the **az login** command executed above



Record the subscription name in the following text box:

Subscription Name
@lab.TextBox(SubscriptionName)

Now run the following command:

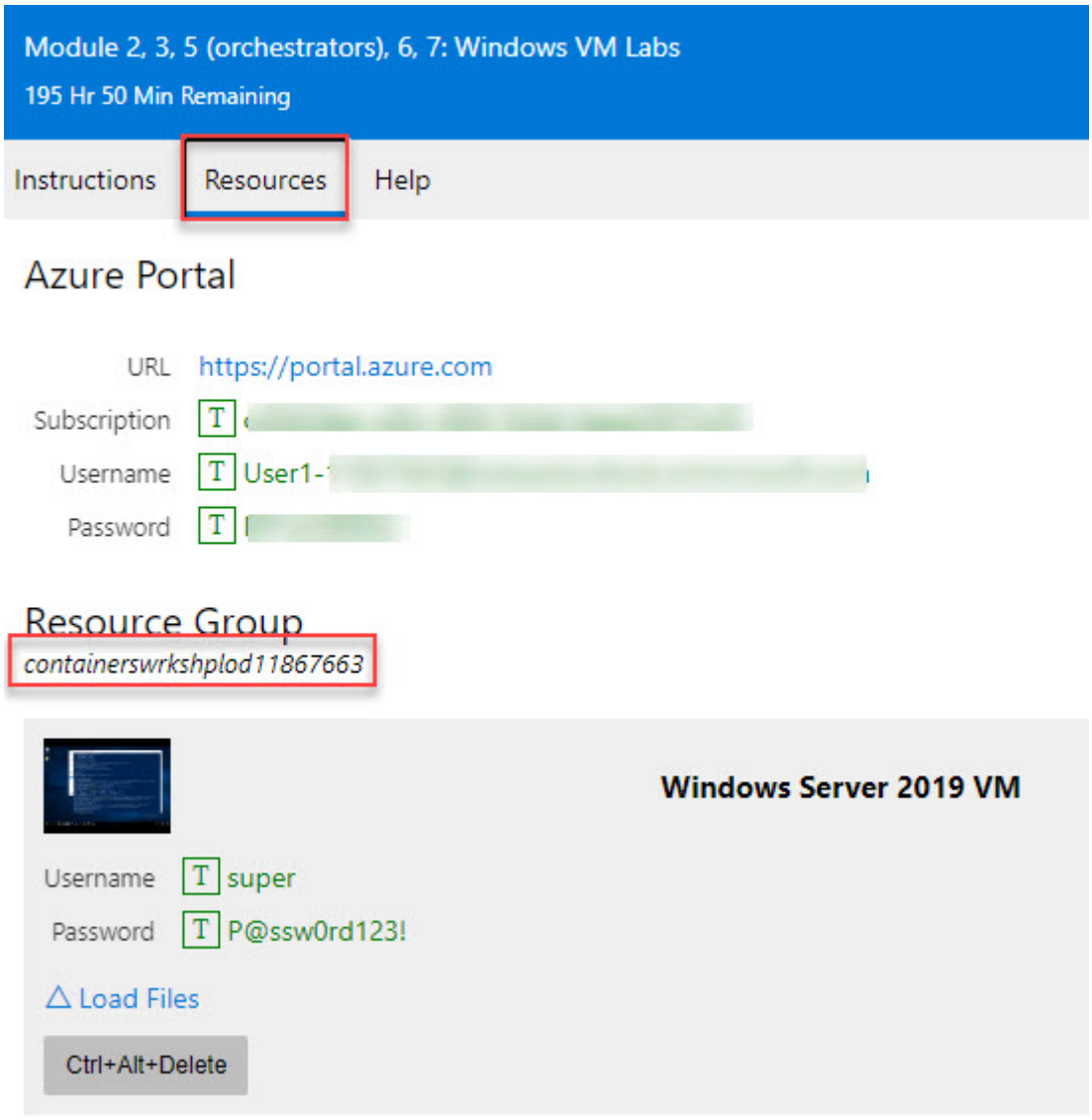
```
az account set --subscription "<SubscriptionName>"
```

5. Get the Resource Group Name for your Labs

Resource Group Name ++@lab.CloudResourceGroup(containerswrkshp).Name++

@lab.TextBox(resource_group_name)

> [!hint] The resource group name can be found on the **Resources** tab



> [!help] Expand the next section **IF** you are using your own Azure Subscription and NOT the one provided in the Lab

▶ Click to expand **if** you are using you own Azure subscription

Run the following command to create a resource group:

```
az group create --name=<resource_group_name> --location=eastus
```

6. Once the resource group is created, deploy a **Kubernetes cluster** using AKS!

```
az aks create --resource-group <resource_group_name> --name aks-k8s-cluster --
disable-rbac --node-count 2 --generate-ssh-keys --network-plugin azure
```

> Note:Note that this command will generate public/private ssh keys in **c:\users\Administrator\.ssh** folder.

> Alert:It can take up to 10 minutes for the cluster to provision successfully in the US. If you are located in Europe, the provisioning can take up to 30 minutes.

7. After the cluster is provisioned successfully you will be shown **JSON** output describing the **AKS cluster**.

8. Locate and copy the value of "**fdqn**" attribute from the **JSON** output.

> Note:Note that your fdqn value may differ from the output shown below.

```
"dnsPrefix": "aks-k8s-cl-k8s-aks-cluster--9bb642",
"fqdn": "aks-k8s-cl-k8s-aks-cluster--9bb642-bd2d483c.hcp.eastus.azmk8s.io",
"kubernetesVersion": "1.7.7",
"linuxProfile": {
  "adminUsername": "azureuser",
  "ssh": {
    "publicKeys": [
      {
```

9. Run the following command to download the Kubernetes cluster configuration to the local config file **C:\users\Administrator\.kube\config**.

   ```
   az aks get-credentials --resource-group <resource_group_name> --name aks-k8s-cluster
   ```

   > [!help]Be aware that you will need the contents of the **.kube/config** file when you create a Kubernetes service endpoint on Azure DevOps in the DevOps module.

10. Run following command to ensure context is set to the correct cluster:

    ```
    kubectl config set-context aks-k8s-cluster
    ```

11. You will now test the cluster by running a nginx container. First create a new deployment using the following command:

    ```
    kubectl create deployment nginx --image=nginx
    ```

12. Next, make sure that you can access the container from the external (public IP). To do that use the expose command to expose port 80 and enable the external IP (type=LoadBalancer). It is going to expose a Kubernetes service through the Azure Load Balancer deployed as part of the AKS cluster. The Kubernetes service will redirect traffic to the deployment we just created:

    ```
    kubectl expose deployment nginx --port=80 --type=LoadBalancer
    ```

13. The above command creates a service with the name nginx. You can view the service by running following command

    +++**kubectl get service nginx**+++

```
PS C:\Users\Administrator> az aks get-credentials --resource-group k8s-aks-cluster-rg-jo --name aks-k8s-cluster
Merged "aks-k8s-cluster" as current context in C:\Users\Administrator\.kube\config
PS C:\Users\Administrator> kubectl config set-context aks-k8s-cluster
Context "aks-k8s-cluster" modified.
PS C:\Users\Administrator> kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
PS C:\Users\Administrator> kubectl expose deployment nginx --port=80 --type=LoadBalancer
service/nginx exposed
PS C:\Users\Administrator> kubectl get services
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP      10.0.0.1       <none>        443/TCP        8m12s
nginx        LoadBalancer   10.0.241.197   <pending>     80:31562/TCP   11s
```

> Note:Please wait until **EXTERNAL-IP** for the **nginx** service change from **<pending>** to a valid IP address. It may take few minutes and you may have to run the command `kubectl get service nginx` a few times to probe the status of external IP assignment. Another useful parameter is **-w** that can be added to the command to watch as the service output changes. When it is done, it will look like below:

```
PS C:\Users\Administrator> kubectl get service nginx -w
NAME    TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
nginx   LoadBalancer   10.0.241.197   <pending>     80:31562/TCP   32s
nginx   LoadBalancer   10.0.241.197   137.135.109.187   80:31562/TCP   2m2s
PS C:\Users\Administrator>
```

14. You can now simply query the content hosted by nginx using the curl command: `start http://$(kubectl get service nginx -o=jsonpath='{.status.loadBalancer.ingress[*].ip}')`

15. Now, to access the Kubernetes dashboard run the following command. Please refresh the opened browser if needed after a couple of seconds.

    ```
    az aks browse --resource-group <resource_group_name> --name aks-k8s-cluster
    ```

16. Log into the dashboard using the credentials stored in the *C:\users\Administrator.kube* file downloaded previously in step 9

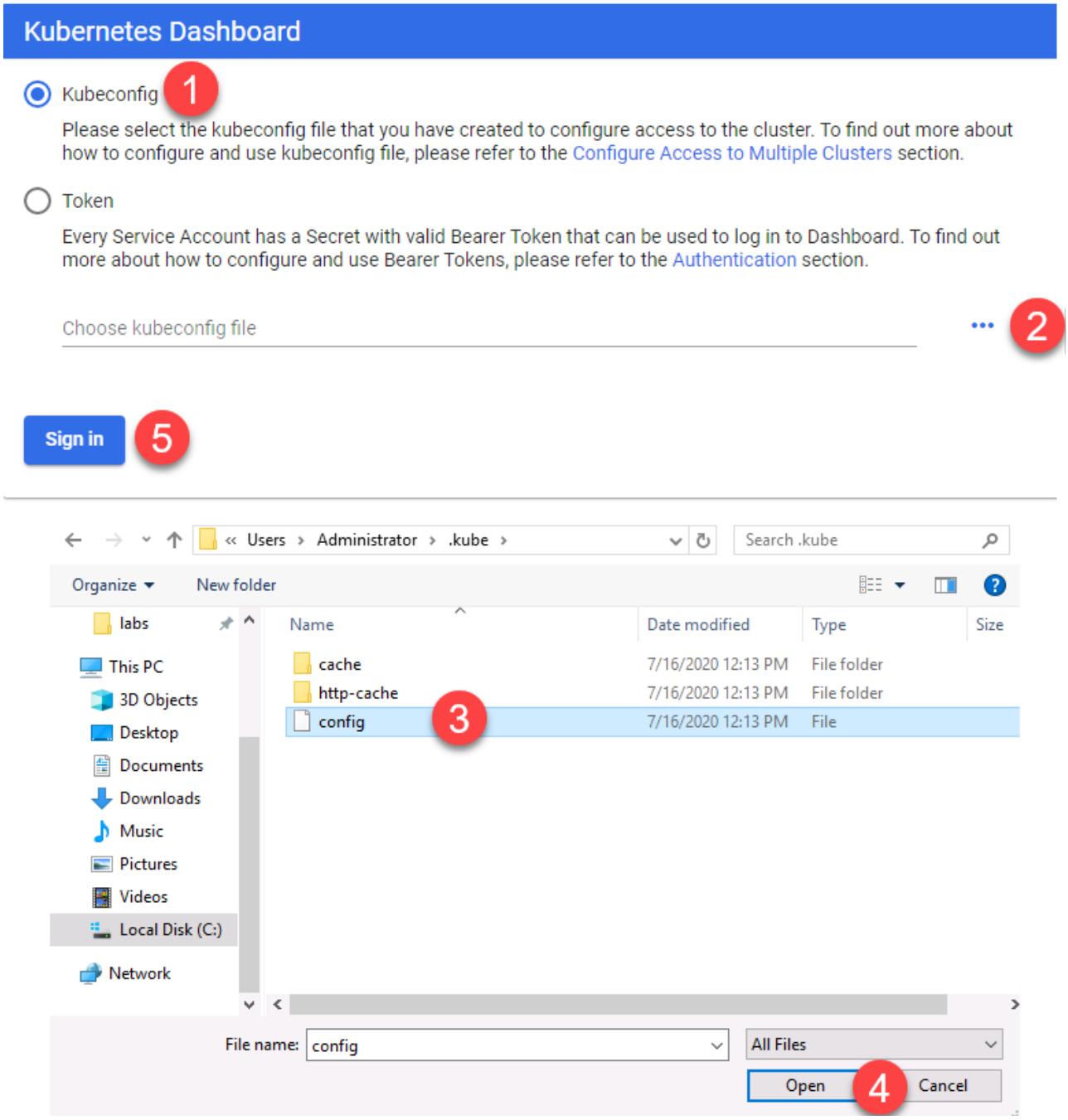
    [1] Select kubeconfig

    [2] locate the file by navigating to the **C:\users\Administrator.kube\config** directory

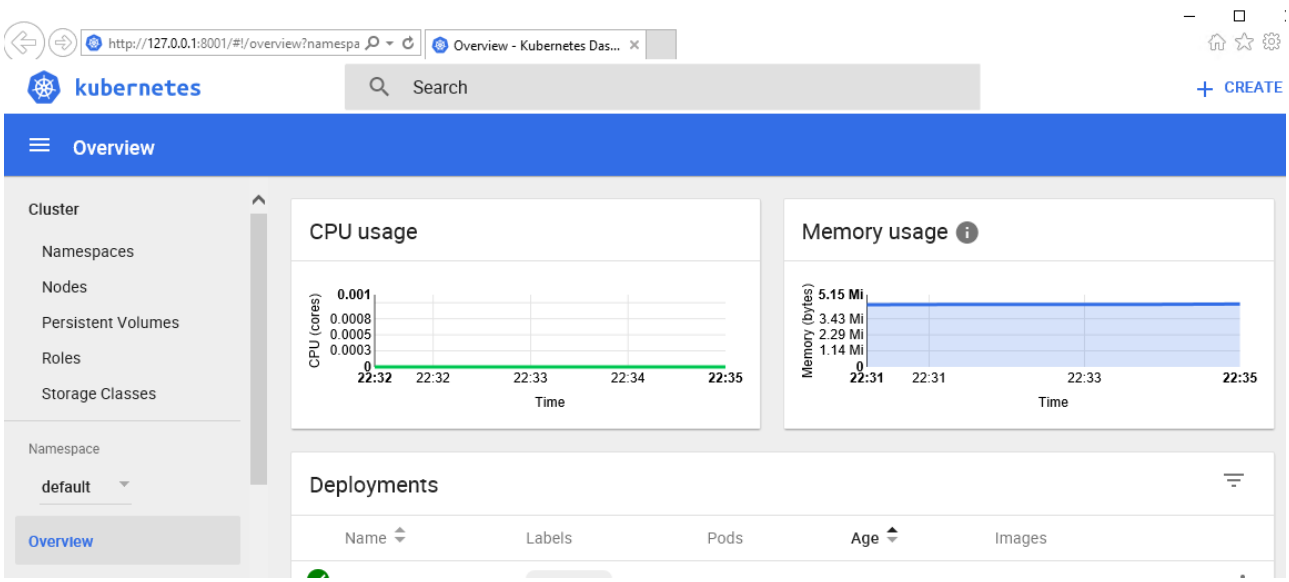
    [3] Choose the **config** file

    [4] Press Open

    [5] Click on **Sign In**


Kubernetes Dashboard
Kubeconfig
1
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.
Token
Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.
Choose kubeconfig file
2
Sign in
5
Users › Administrator › .kube ›
Search .kube
Organize
New folder
labs
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk (C:)
Network
Name
Date modified
Type
Size
cache 7/16/2020 12:13 PM File folder
http-cache 7/16/2020 12:13 PM File folder
config 7/16/2020 12:13 PM File
3
File name: config
All Files
Open
4
Cancel

Dashboard


http://127.0.0.1:8001/#!/overview?namespa
Overview - Kubernetes Das...
kubernetes
Search
+ CREATE
Overview
Cluster
Namespaces
Nodes
Persistent Volumes
Roles
Storage Classes
Namespace
default
Overview
CPU usage
CPU (cores)
0.001
0.0008
0.0005
0.0003
0
22:32
22:32
22:33
22:34
22:35
Time
Memory usage
Memory (bytes)
5.15 Mi
3.43 Mi
2.29 Mi
1.14 Mi
0
22:31
22:31
22:33
22:35
Time
Deployments
Name
Labels
Pods
Age
Images

> Note:You can always access the dashboard by browsing to the URL: **http://127.0.01:8001**. Although this command works with both Windows and Linux, you should only run it on Windows as you have access to a full browser on Windows running inside the lab virtual machine.

17. Stop running the dashboard by entering "**Ctrl + C**", followed by "**Y**" to terminate the batch job.

18. Let's check total number of pods running now. You should see one pod ready with a status of running.

> Note:When using **kubectl create**, deployments are created with a single pod by default.

```
kubectl get pods
```

19. You will now scale the number of pods using a replica set. You can get more details about the replica set using the following command:

```
kubectl get replicaset
```

20. To scale, run the command and pass the name of deployment that was created earlier

```
kubectl scale deployment nginx --replicas=3
```

21. Now, run the following command to view the number of running pods (Expected result is that the number has scaled from 2 to 3).

```
kubectl get pods -o wide
```

22. In a previous step, you successfully increased the number of pods by increasing the replica set. However, all the pods are running on two nodes. You can check that by running the following command

```
kubectl get nodes
```

In addition to the number of pods, we might want to adjust the compute capacity of the cluster itself. For instance, to remove one worker node from the cluster, we can run the following command:

```
az aks scale --node-count=1 --resource-group <resource_group_name> --name aks-k8s-cluster
```

> Note:Please wait while the worker node is successfully removed from the cluster. This may take few minutes to complete.

23. If you check the number of nodes again, you should see a single worker node running in the cluster. You can independently adjust the number of pods and number of working nodes in a cluster. You can also look at the maximum pod capacity of a node by running the command:

```
kubectl get node NODE-NAME -o=jsonpath='{.status.capacity.pods}'
```

24. To scale back down to a node count of 2, run the following command:

```
az aks scale --node-count=2 --resource-group <resource_group_name> --name aks-k8s-cluster
```

25. Finally, remove the deployment and service using the commands below:

```
kubectl delete deployment nginx; kubectl delete service nginx
```

26. Do not delete your AKS cluster in the Azure Portal, you will need the cluster for the CI/CD portion in the next module.

Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 2: Create a Windows Node Pool

In this exercise, you will add an additional node pool to your Azure Kubernetes Cluster (AKS) that will allow you to run Windows Server containers. The default node pool in AKS creates nodes running a Linux operating system and only allows linux containers. We will create an additional node pool using a Windows Server operating systems allowing you to run Windows containers. The container image will be fetched from the Docker Hub public registry.

Return to list of exercises - Return to list of modules

1. On your Windows server VM, open PowerShell and past the following command to create a new Windows Server Node Pool

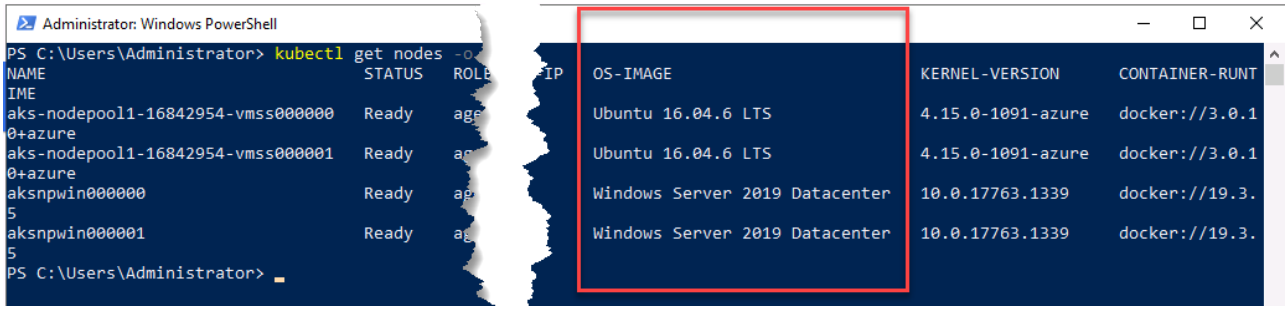
   > Alert: It may take approximately 10 minutes for the node pool to provision successfully in the US. If you are located in Europe, the provisioning may take longer.

```
az aks nodepool add --resource-group @lab.CloudResourceGroup(containerswrkshp).Name --cluster-name aks-k8s-cluster --os-type Windows --name npwin --node-count 2 --node-taints kubernetes.io/os=windows:NoSchedule
```

> Knowledge:
>
> What does **--node-taints kubernetes.io/os=windows:NoSchedule** do?
>
> When a taint is applied to a node it means only pods with a declared toleration that matches can be scheduled on it. Therefore, in order to deploy a pod to the Windows Nodes you first have to add a **tolerations** section and **nodeSelector** to your YAML manifest (see below). Manifests for Linux pods do not need to be modified and will, by default, only deploy to Linux nodes. Without these restictions, Kubernetes will simply choose an available node and if it chooses the wrong operating system the Pod will fail to create.

2. Verify that your Kubernetes cluster has 2 Linux nodes and 2 Windows nodes

```
kubectl get nodes -o wide
```



> Alert: At the time of this writing, AKS requires Windows containers to use images based on **Windows Server 2019 Build 1809** .

When deploying Windows containers using a Kubernetes manifest file a **toleration** and a **node selector** must be defined to tell your AKS Cluster to deploy your containers to Nodes running Windows Server only. The **toleration** overrides the **taint** setup when the Node Pool was created and the **node selector** is usually accomplished by using the built-in label **kubernetes.io/os** with a value of **windows** (can also be linux)

```

```

spec: tolerations: - key: kubernetes.io/os operator: Equal value: windows effect: NoSchedule nodeSelector: "beta.kubernetes.io/os": windows ```

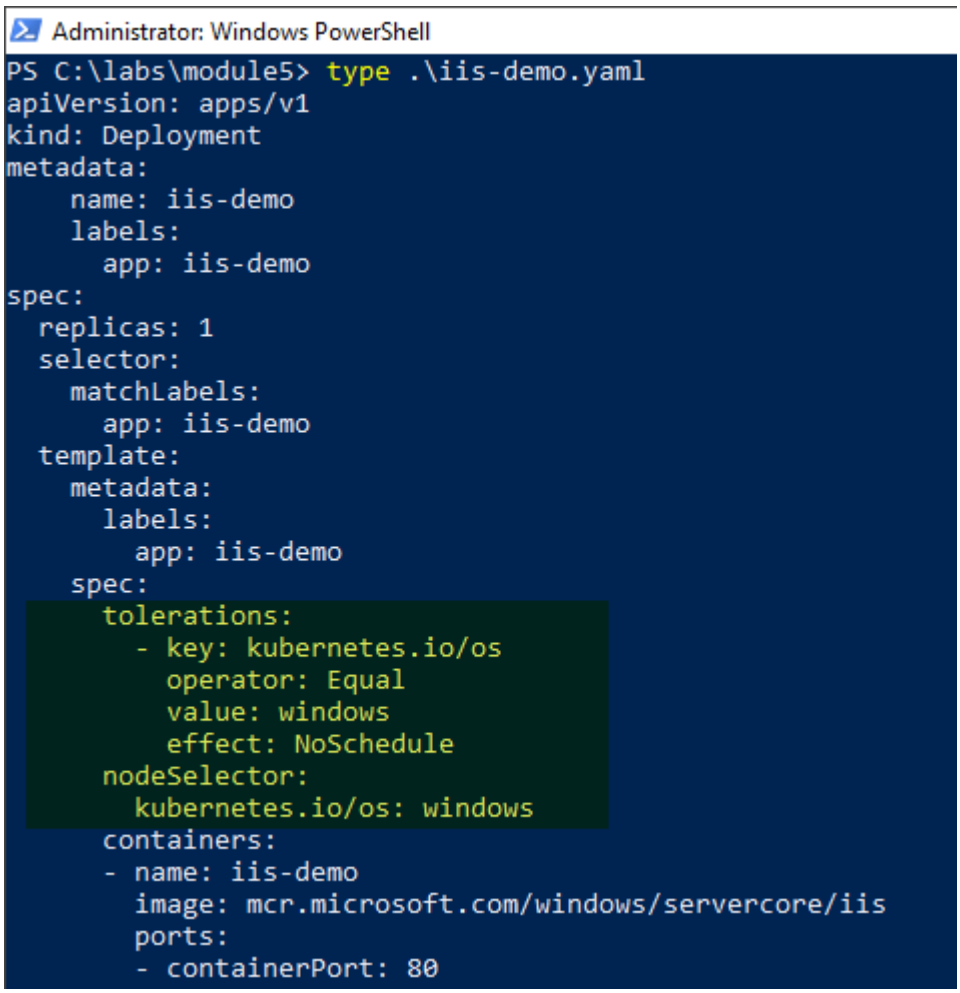
1. Change to the c:\labs\module5 directory

   +++cd c:\labs\module5+++

2. View the contents of the Kubernetes manifiest file **iis-demo.yaml**

   +++type .\iis-demo.yaml+++

   Notice the **tolerations** and **nodeSelector** sections needed to deploy to Windows Nodes.



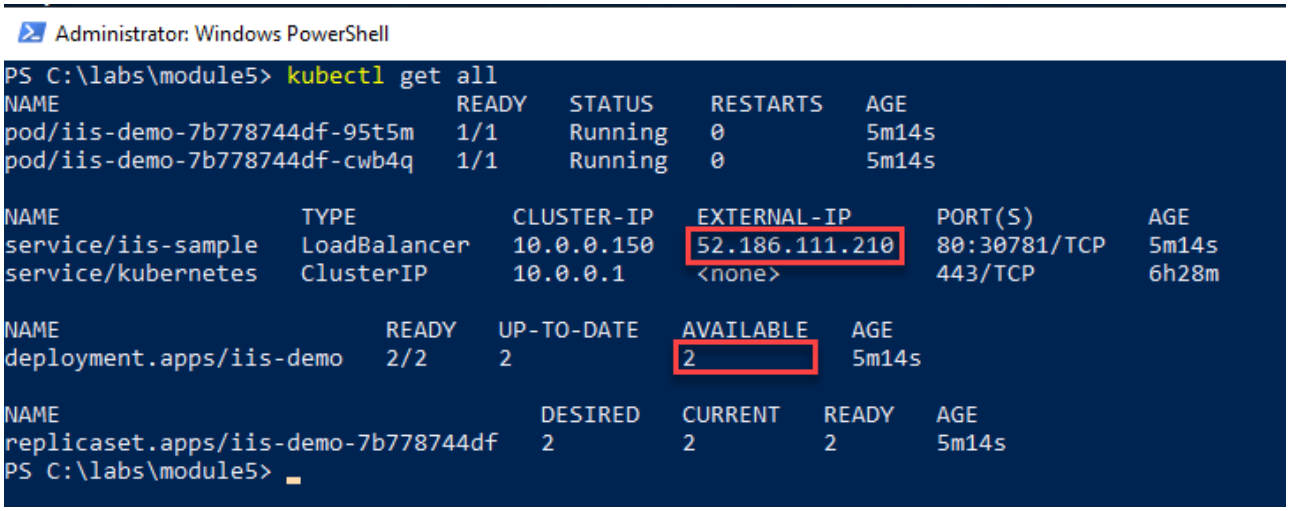
3. Deploy the Application and Load Balancer using the command

   +++kubectl apply -f .\iis-demo.yaml+++

> Note: It may take several minutes for the nodes to download the windows image before the pod status changes to **running**

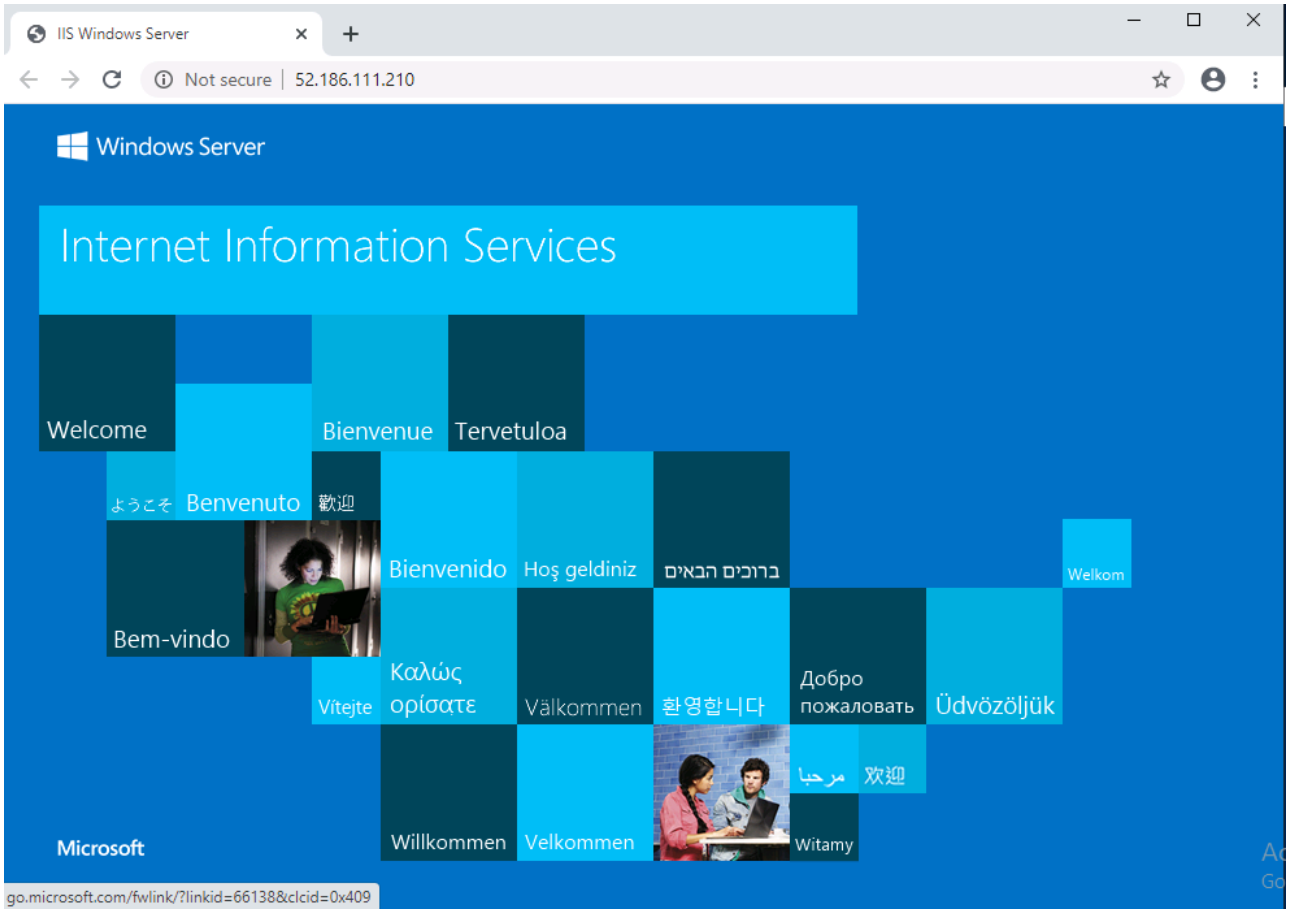
4. Wait for the application to deploy. Verify that EXTERNAL-IP is available for **service/iis-sample** and **deployment.apps/iis-demo** has **2** pods available

   +++kubectl get all+++



5. launch the browser and verify that the default IIS landing page is shown

   +++start http://$(kubectl get service iis-sample -o=jsonpath='{.status.loadBalancer.ingress[*].ip}')+++



6. Cleanup resources by deleting the IIS Pod and Service

   +++kubectl delete -f .\iis-demo.yaml+++

Congratulations!

You have successfully completed this lab. Click **Next** to advance to the next lab.

Module 6 - DevOps with Containers

Duration

> 120 minutes

Module 6: Table of Contents

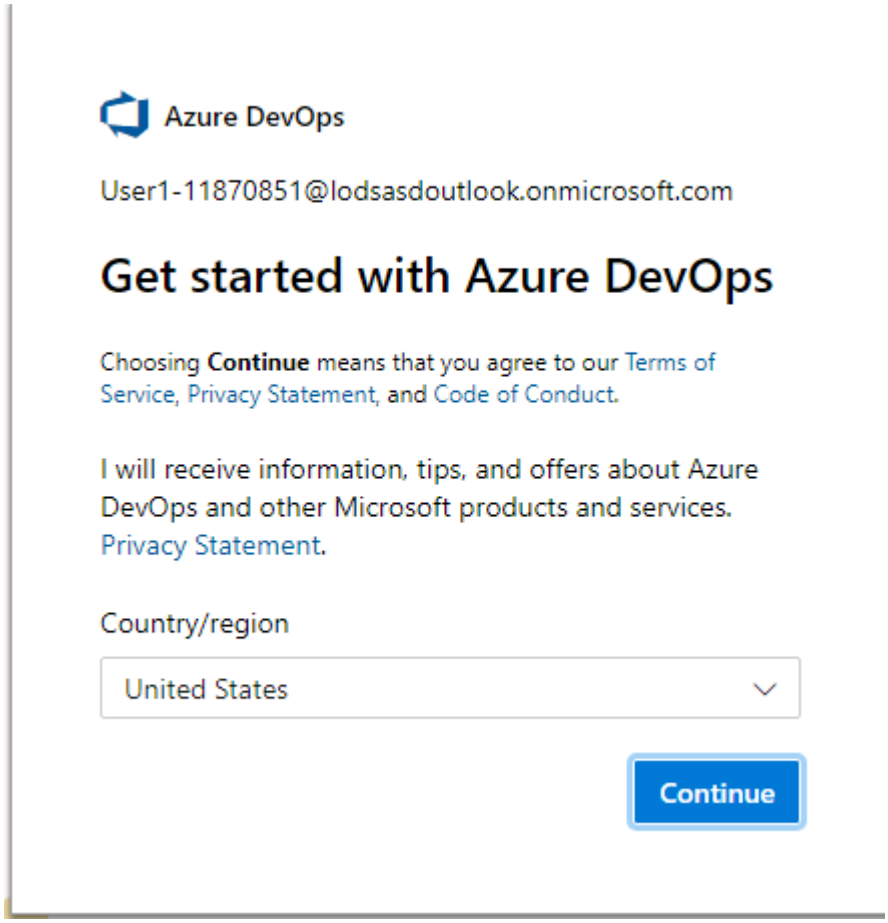
Exercise 1: Setup an Azure DevOps account

In this exercise, you are going to create an Azure DevOps account which will be leveraged to create a CI/CD pipeline for your application. If you already have an Azure DevOps account, you can skip this exercise.

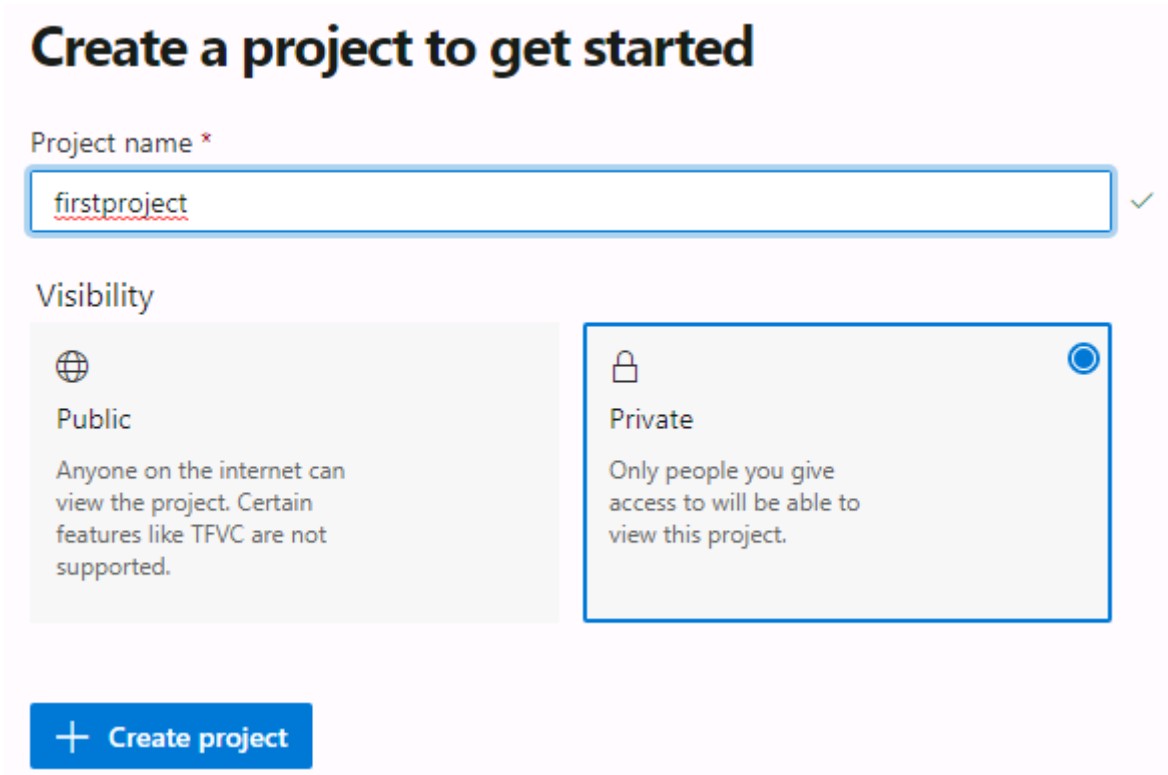
Return to list of exercises - Return to list of modules

1. Sign into your **Windows VM** LOD and open a Chrome window inside of the VM. Navigate to *dev.azure.com* +++start https://dev.azure.com+++ and sign in with your Azure Pass account. Click on **Start Free** to open Azure DevOps.

2. From the Dev Portal, click on the **Continue** button.



3. For the Project Name use `firstproject`. then click on **Create Project**



Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

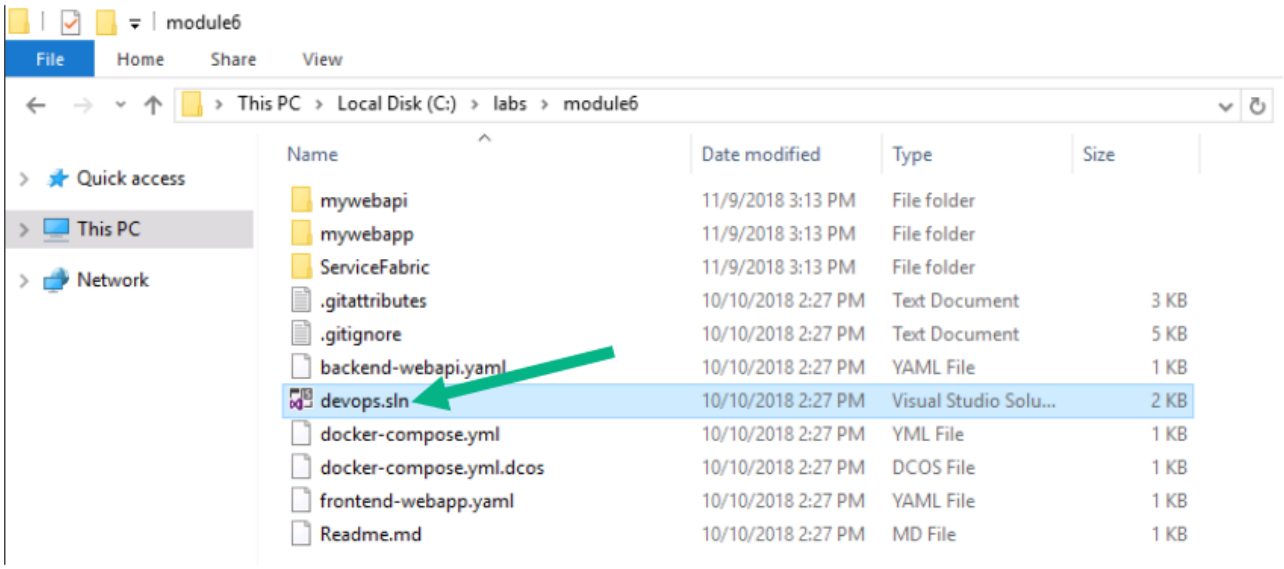
Exercise 2: Setup a project and Azure Container Registry (ACR)

In this exercise, you are going to complete the first phase of the CI/CD pipeline by pushing the code artifacts for the sample Web App and Web API to Azure Repository. You will also create an Azure Container Registry (ACR) to privately store the Web App and Web API Docker images.

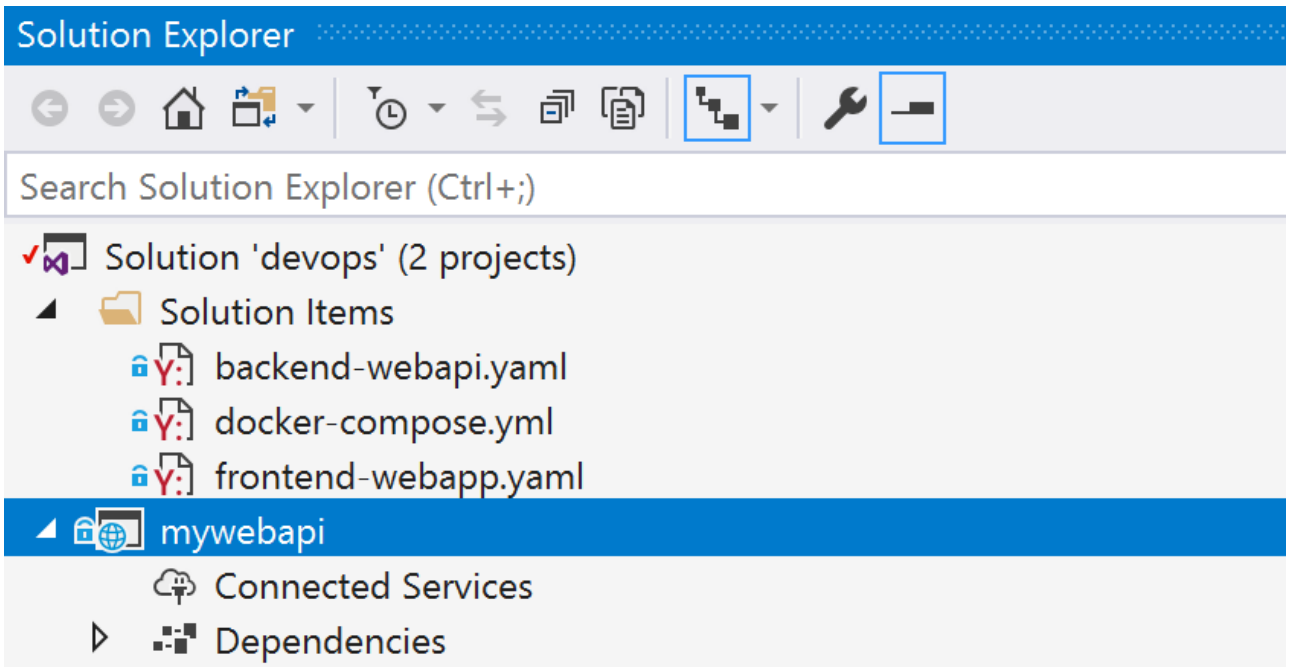
Return to list of exercises - Return to list of modules

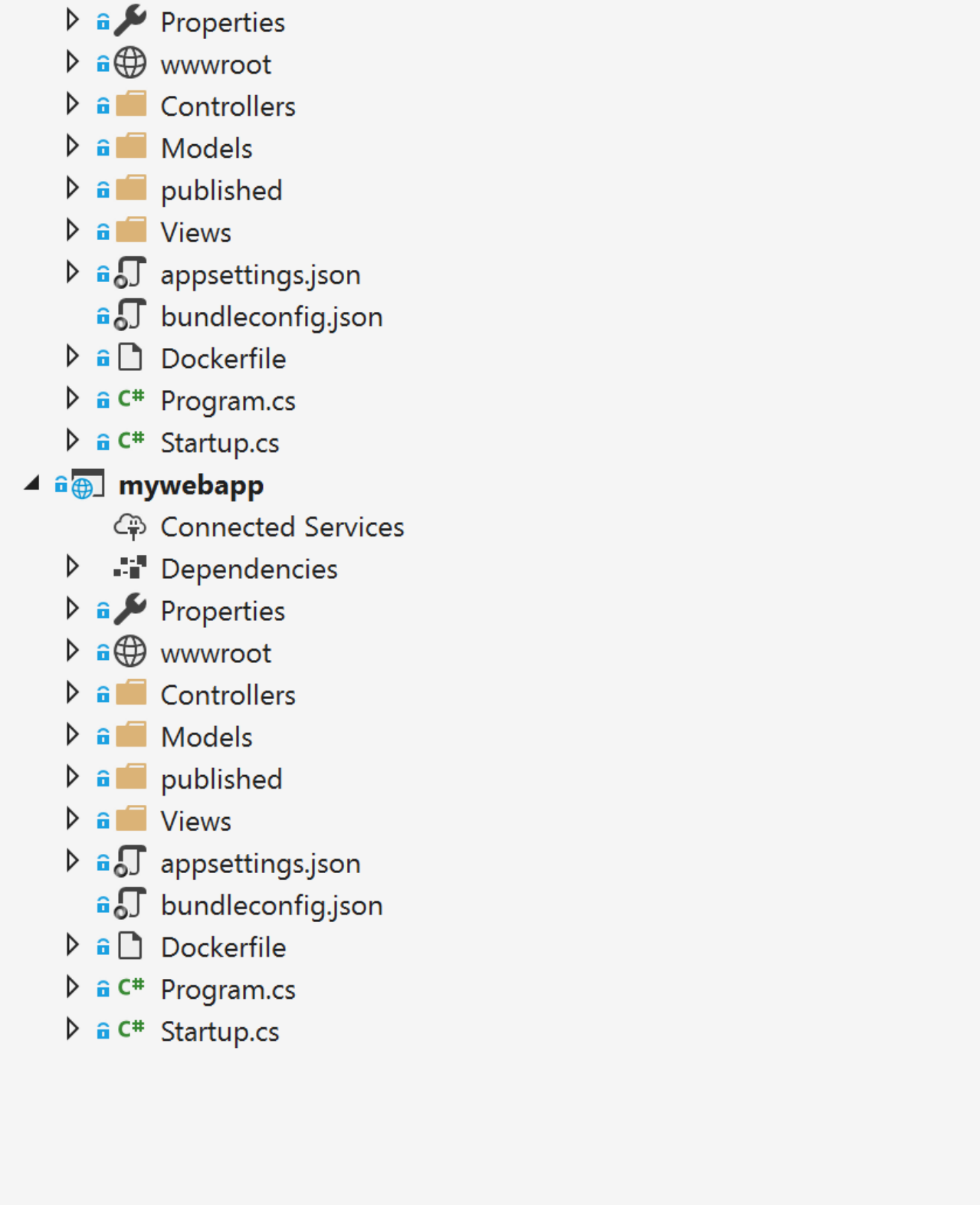
Examine and Build sample application in Visual Studio

1. Navigate to **C:\labs\module6** [+++cd C:\labs\module6;start devops.sln+++] within the Windows VM. Double click on the "devops" solution file which will open the project in Visual Studio.

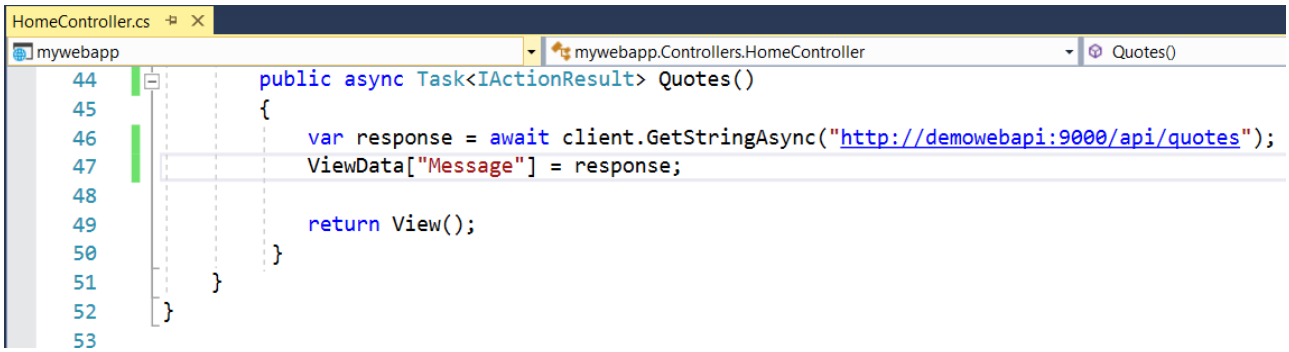


2. On the right, you will see the **Solution Explorer**. You should see two ASP. NET Core applications, a Web API backend project and an MVC Core frontend project.

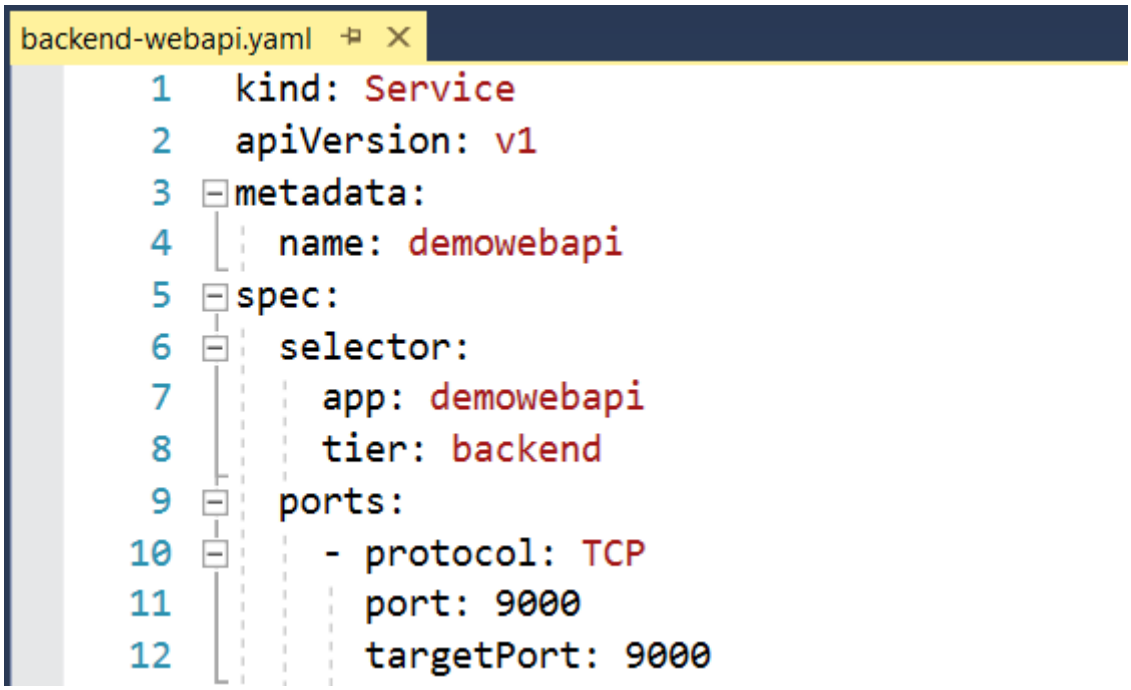




3. Open **HomeController.cs** in the **mywebapp** project, under the **Controllers** folder. You will see that the **frontend** application retrieves some quotes from the Web API. The URL to reach to the Web API is configured by the container orchestrator used. For example, in Kubernetes, this happens through Labels and Selectors, along with the use of Services.



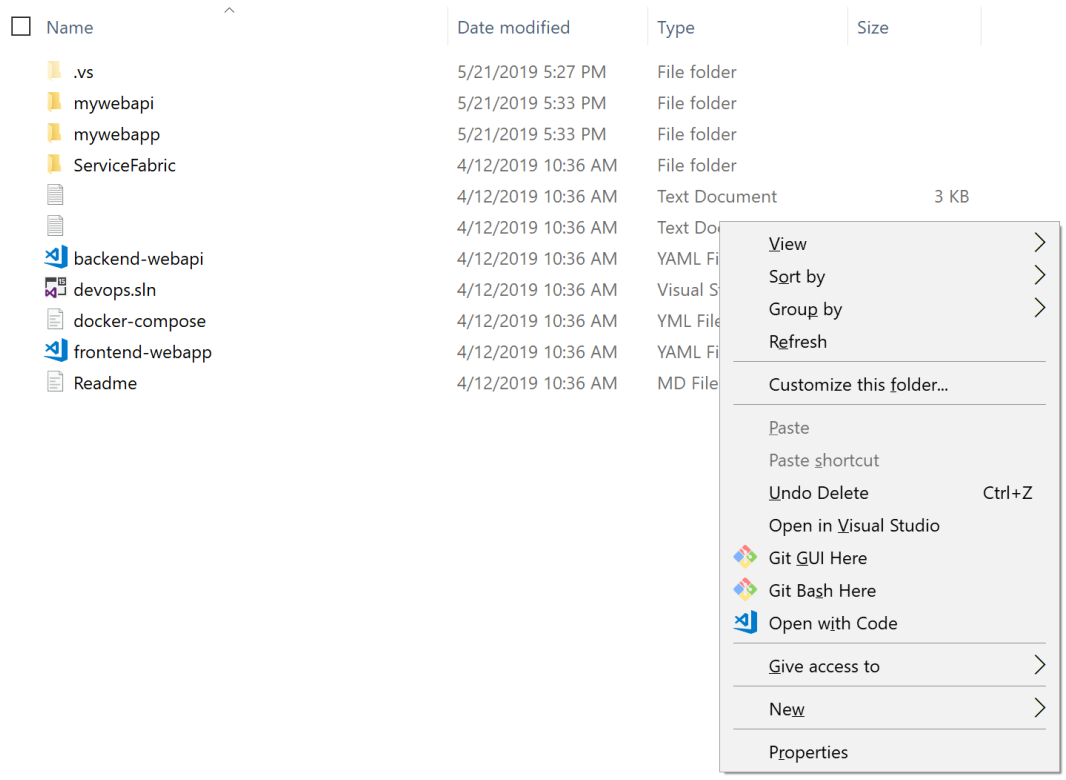
4. In the **Solution Items** folder, open the **backend-webapi.yaml** file which is the deployment configuration file for Kubernetes cluster. Notice that here the backend application is also exposed with the name **demowebapi**.



Add solution to source control (Git and Azure Repository)

1. Open File Explorer and navigate to **C:\labs\module6**.

2. Right click choose **Git Bash Here**.

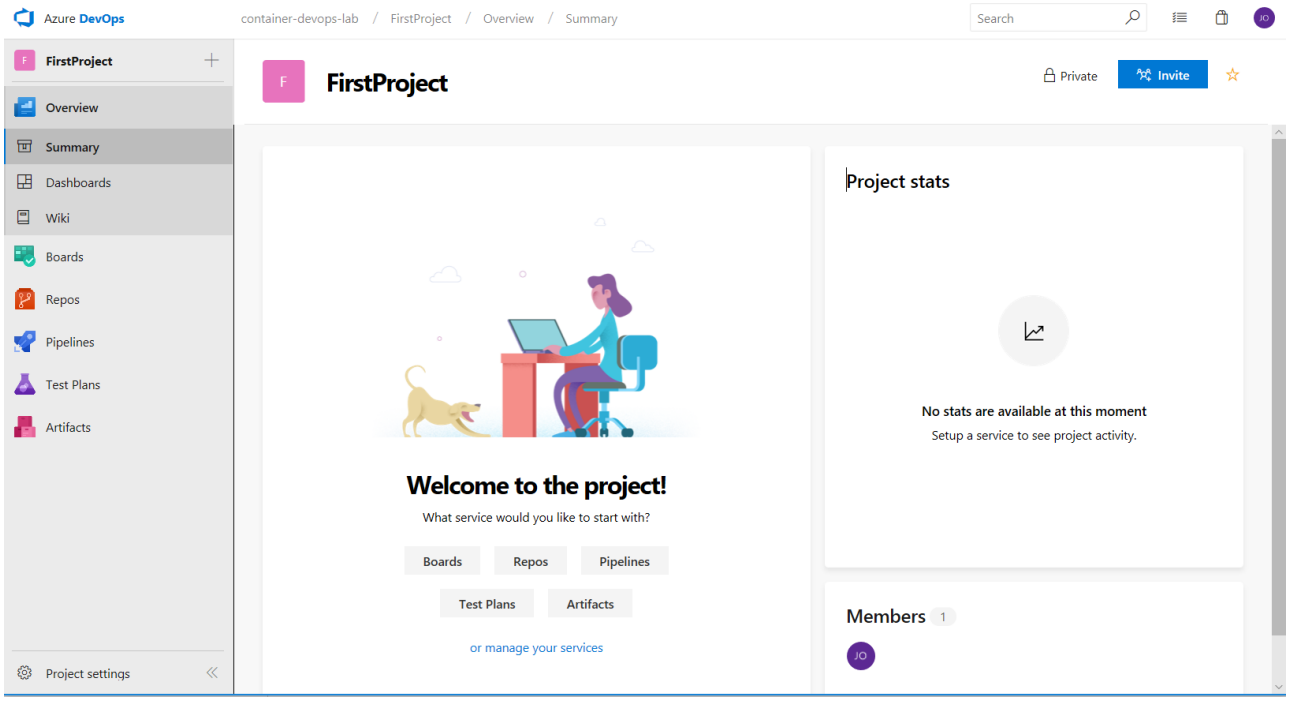


3. Run the following commands, one at a time

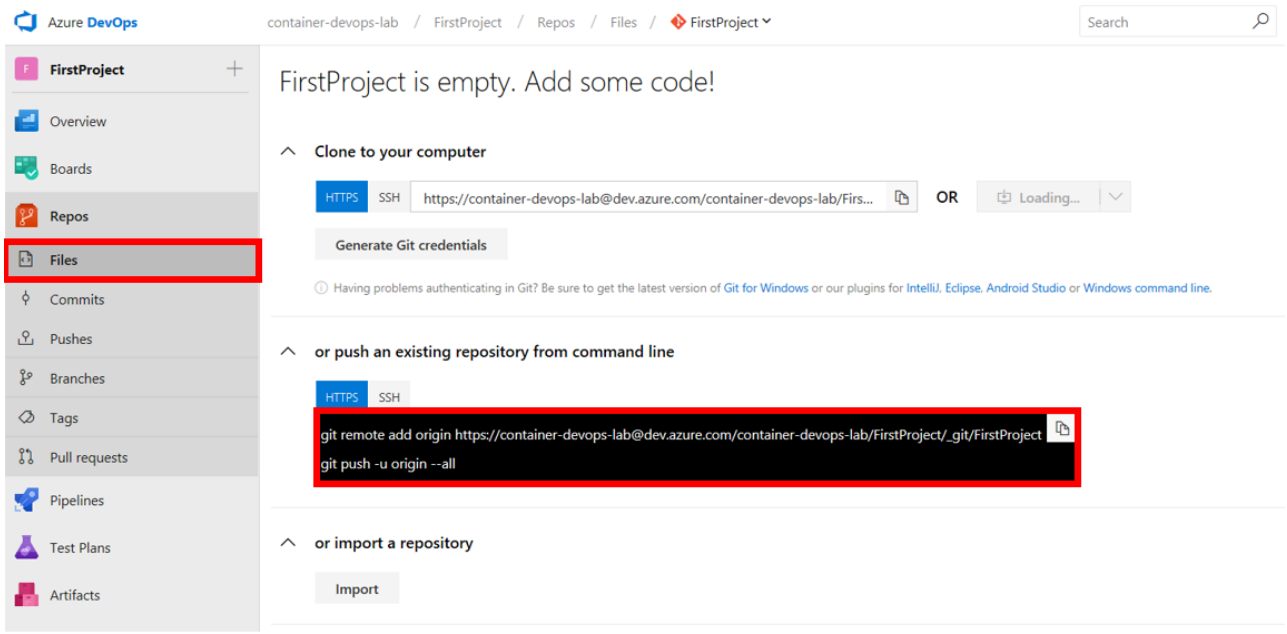
> Note:You can either fill in a name and email or leave it as the default. The two **Git config** commands are a one-time setup. **Git init** will initialize your folder as a git repository and add an empty hidden **.git** folder. **Git add** and **commit** are necessary steps anytime you make a change to your code and want it to be staged (tracked and ready to be committed locally) and committed (ready to be pushed to the server)

```
git config --global user.email &lt;you@example.com&gt;
git config --global user.name "Your Name"
git init
git add .
git commit -m 'adding my web project'
```

4. Navigate to https://dev.azure.com/**YourAccountName/YourProjectName** to see the environment. A sample link is https://dev.azure.com/container-devops-lab/FirstProject



5. Now you are ready to push your local Git repository to a remote repository in Azure DevOps. Navigate to your Code repository by clicking on **Repos - Files**. You will see that your Git repository is empty, as expected. In order to push your local code files into Azure DevOps, copy the commands under **"push an existing repository from command line"**



6. Paste them into Git Bash by pressing **Shift + Insert** (or right click and paste into the Git Bash command line). The first line will be run automatically; hit enter to run the second line.

> Knowledge: **Git remote add origin** means to add a "remote URL such as your Azure repo URL" with the alias **origin** to your local git settings for this project folder. You could name it any alias you want, **origin** is just the default. **Git push** means to interact with the server and push code to that URL you provided as origin. The **-u**" will "set-upstream" which means that it will default all pushes and pulls to the **origin** URL and you don't have to specify **origin** in all your future commands.

```
super@5dokdkszzx3za MINGW64 /c/labs/module6 (master)
$ git remote add origin https://container-devops-lab@dev.azure.com/container-devops-lab/FirstProject/_git/FirstProject

super@5dokdkszzx3za MINGW64 /c/labs/module6 (master)
$ git push -u origin --all
```

You will see a popup to authenticate to your Microsoft account. Be sure to use your Lab Azure Account on the resources tab. Once authenticated, your project will be pushed to the remote Git repository in Azure DevOps. Return to Azure DevOps in the browser and refresh the page to see your code files populate **Repos - Files**.

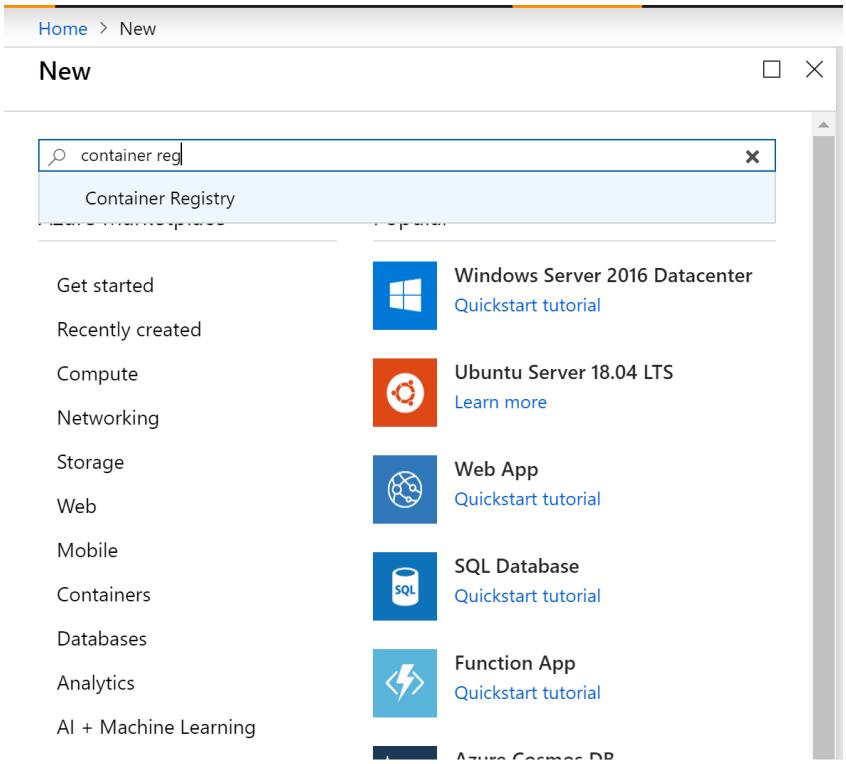
```
![](content/mod6image12.png)
```

Create an Azure Container Registry (ACR)

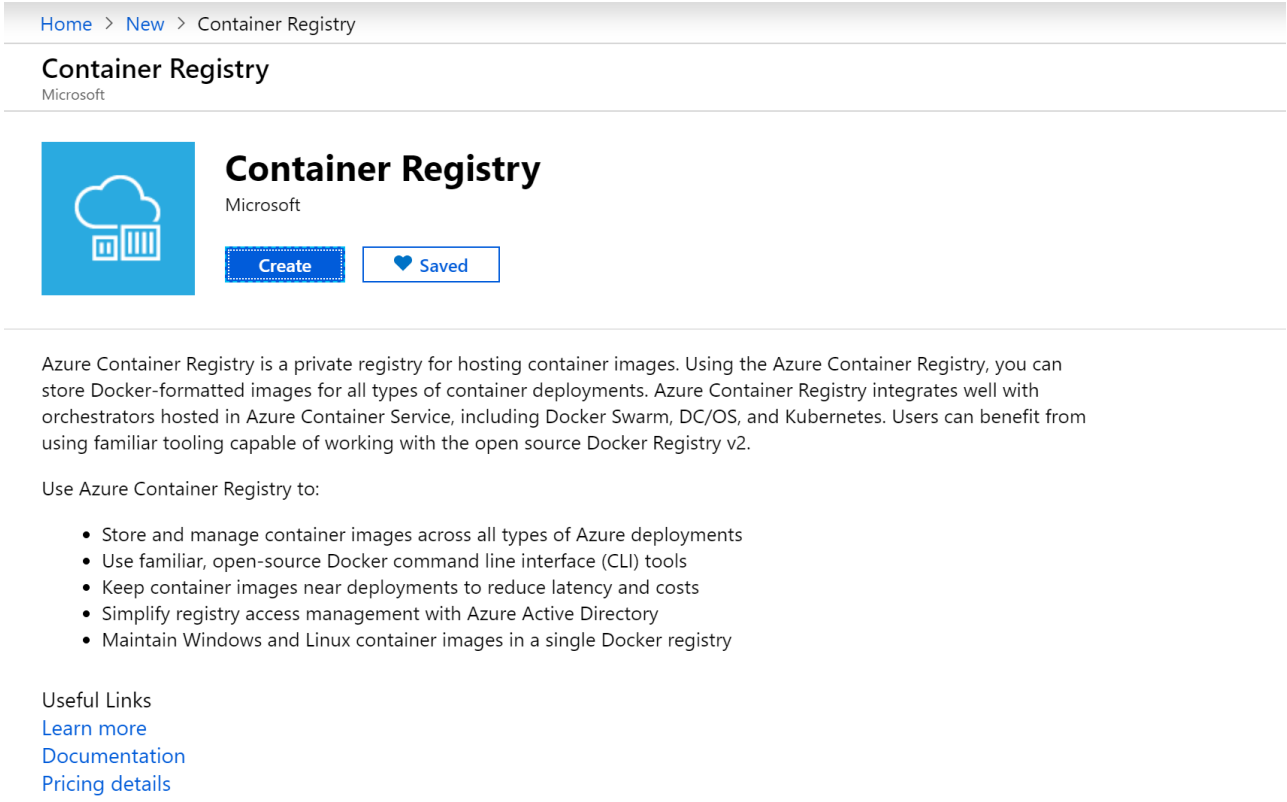
A critical component of the Containerized DevOps lifecycle is the container registry. The container registry allows you to store and manage your container images. You can leverage public registries such as Docker Hub or private registries either installed on-premise or hosted by a cloud provider. In this task, you are going to use Azure Container Registry to manage your Linux and Windows container images

Return to list of exercises - Return to list of modules

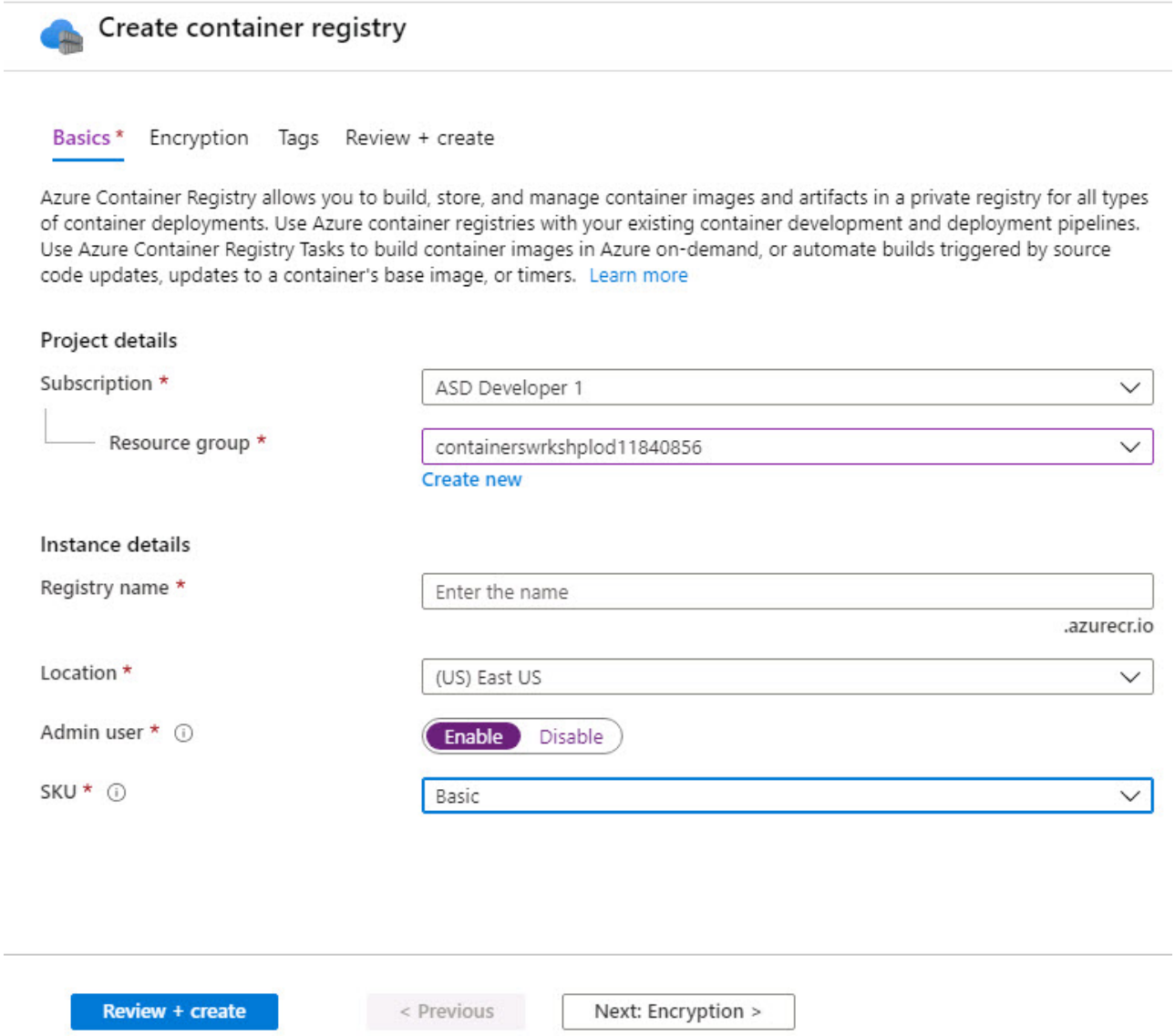
1. In the Azure Portal, click the **plus** icon to add a new resource. Type **"Container Registry"** in the search textbox and click on the result for **Container Registry**.



2. Click **Create**.

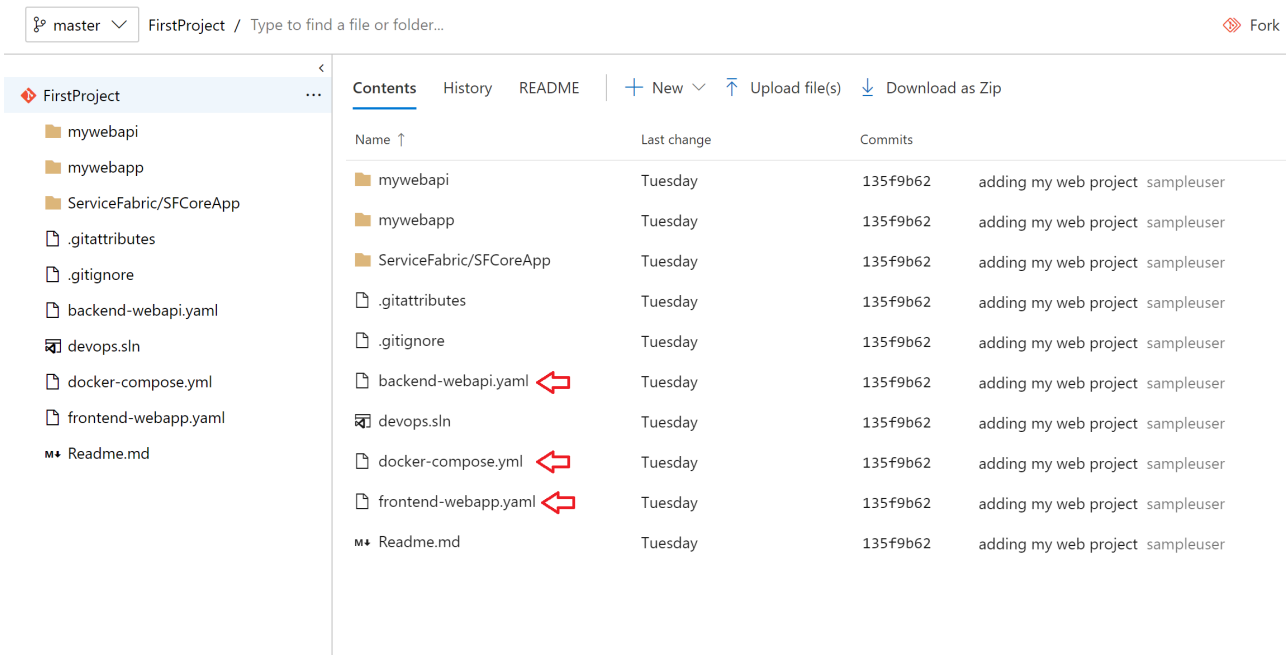


3. Select a unique registry name and specify a resource group. You can select the resource group used in the previous labs if you would like to keep the resources grouped together. Make sure to enable **Admin user** so that build agents can login to ACR and push container images successfully.



> Note:For future reference, in order to enable features like geo-replication for managing a single registry across multiple regions, content trust for image tag signing, and firewalls and virtual networks (preview) to restrict access to the registry, you must use the Premium SKU.*

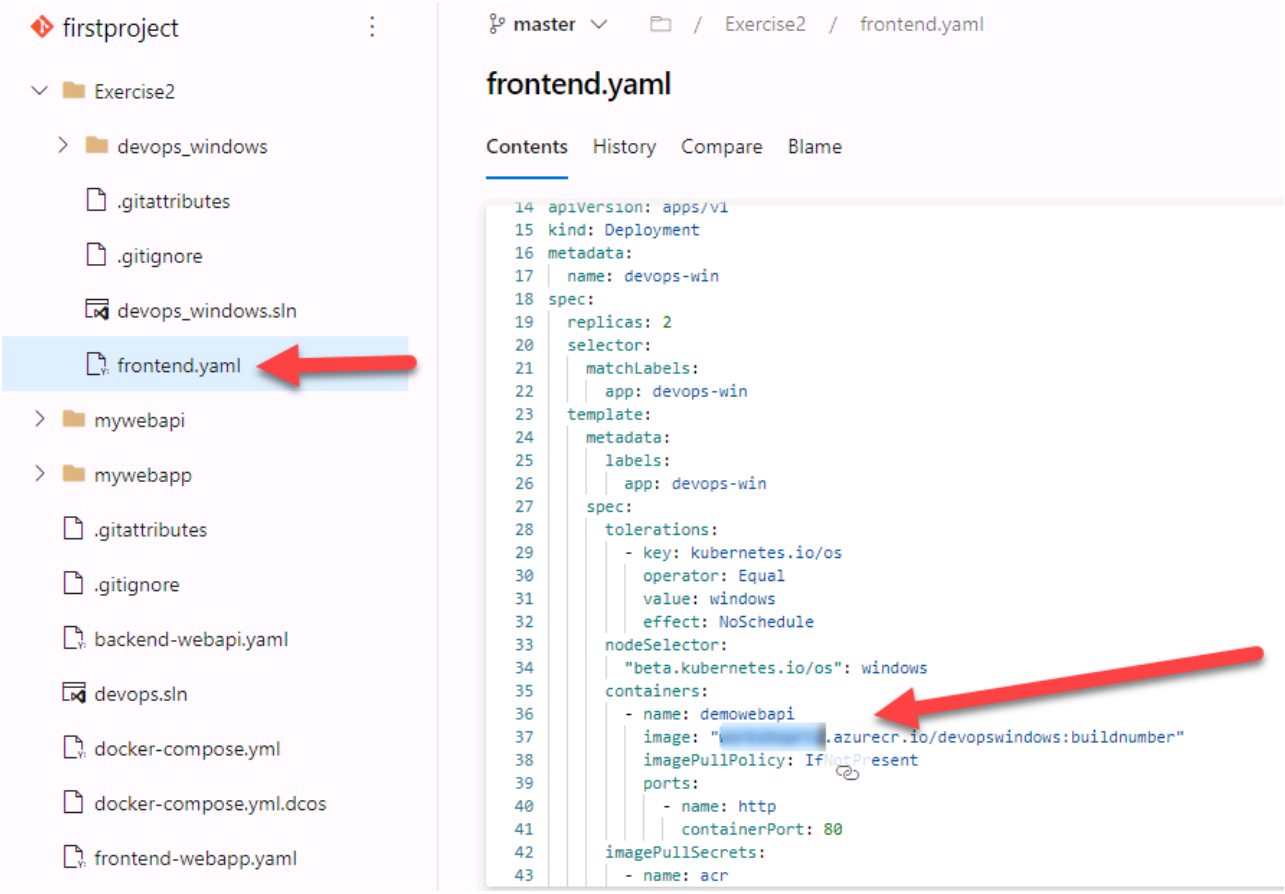
4. While the ACR is being built, you can update your Kubernetes configuration files to use the proper container registry name. Go into Azure DevOps and hit the **Repos** > **Files** tab. The files that you will need to be edited are: **backend-webapi.yaml, docker-compose.yml, and frontend-webapp.yaml**.



5. Go into each of the 3 files individually and replace all instances of the existing ACR name **devopslabacr** with the new ACR name that you just created **(enter your ACR name in all lowercase, otherwise you will get an error later!)**.

> Alert:There should be a total of 4 replacements. **Once again, make sure you write your ACR name in lowercase**, e.g. **devopslabacr.azurecr.io demo-webapp:BuildNumber**. If you don't, you may face authentication issues when AKS tries pulling your container images during deployment. The Kubernetes task in the Azure Pipeline will create a secret using the lowercase ACR address, therefore the ACR name in the yaml needs to match.

6. Similarly, navigate to the **frontend.yaml** file located in **Exercise2** directory. Replace the image name prefix with the new ACR that you just created instead of the default value **devopslabacr**.



Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

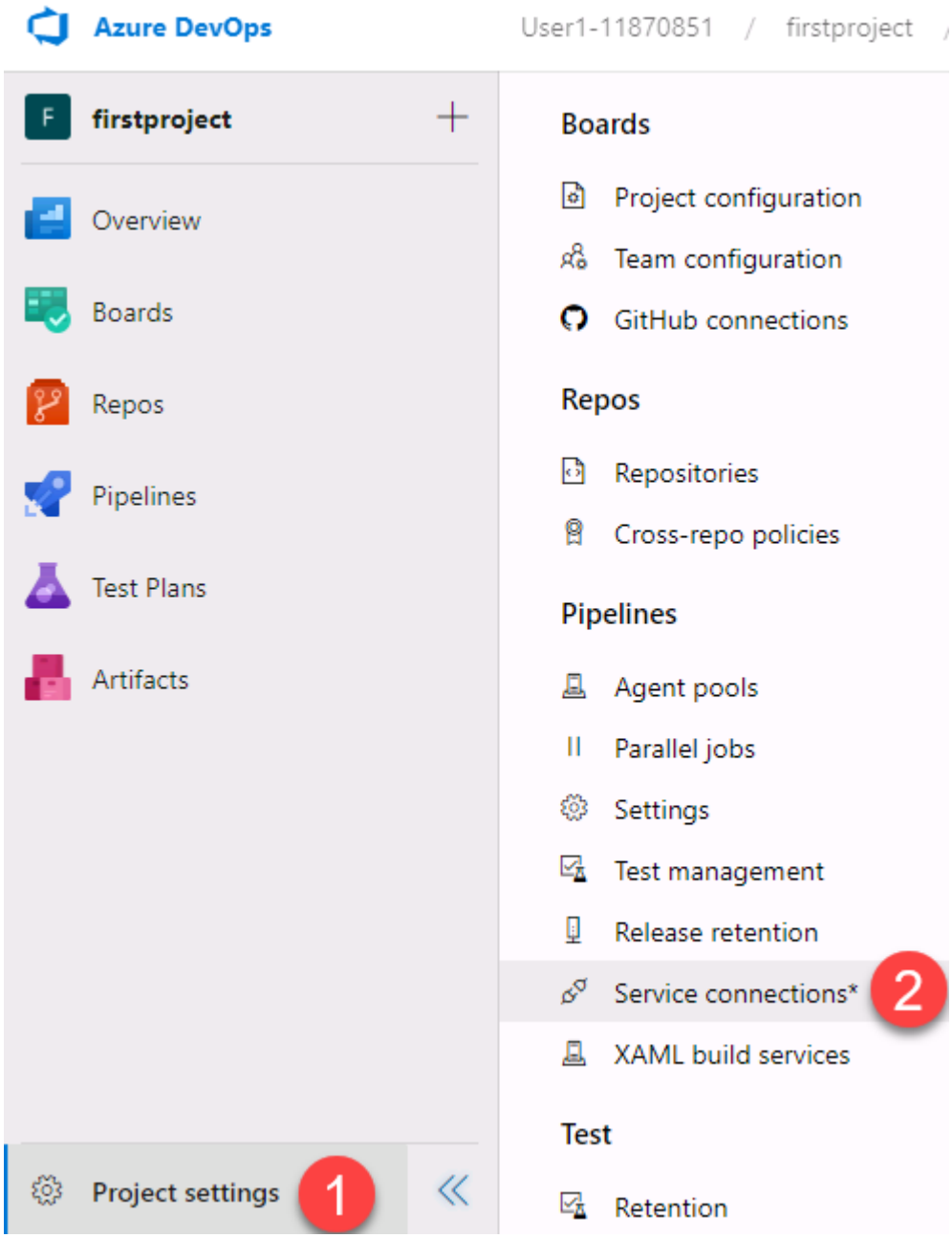
Exercise 3: Create Build Pipeline for Linux Containers

Return to list of exercises - Return to list of modules

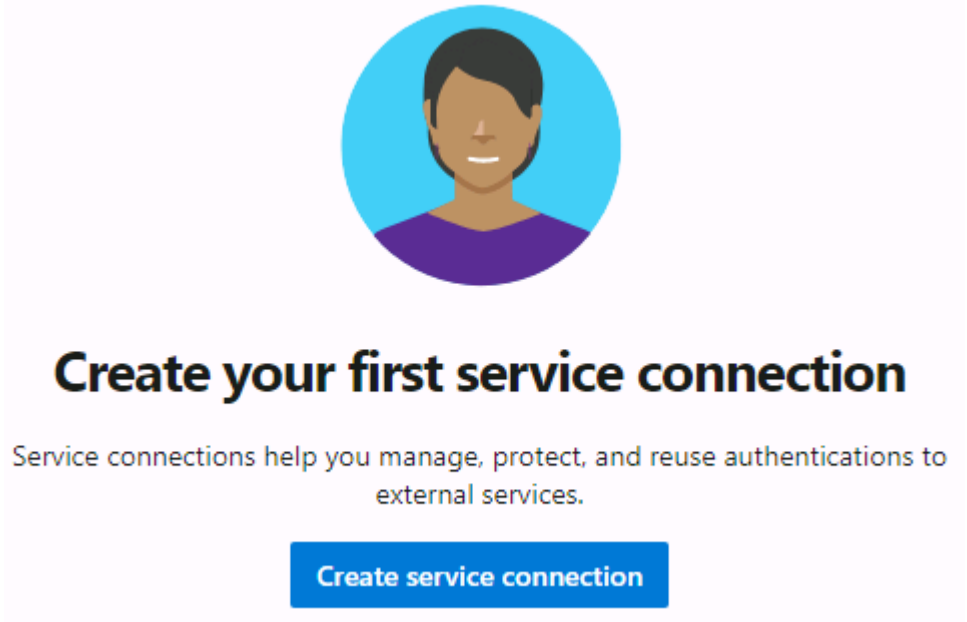
Create a connection to the Azure Container Registry

In this task , you will create a connection the Azure Container Registry (ACR) created in the previous exercise. You will use this connection in the build pipeline to push your new images to the Container Registry and in the release pipeline to have your Kubernetes Cluster pull the images.

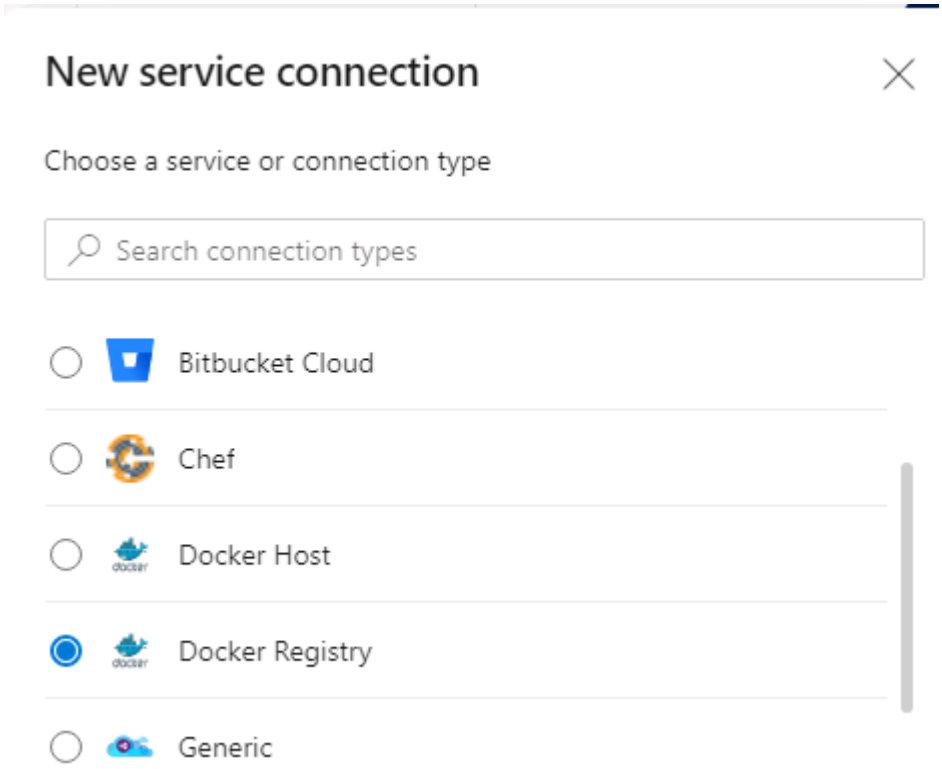
1. Navigate to https://dev.azure.com/<YourAccountName>/<YourProjectName>, click on **Project Setting** then **ServiceConnections**



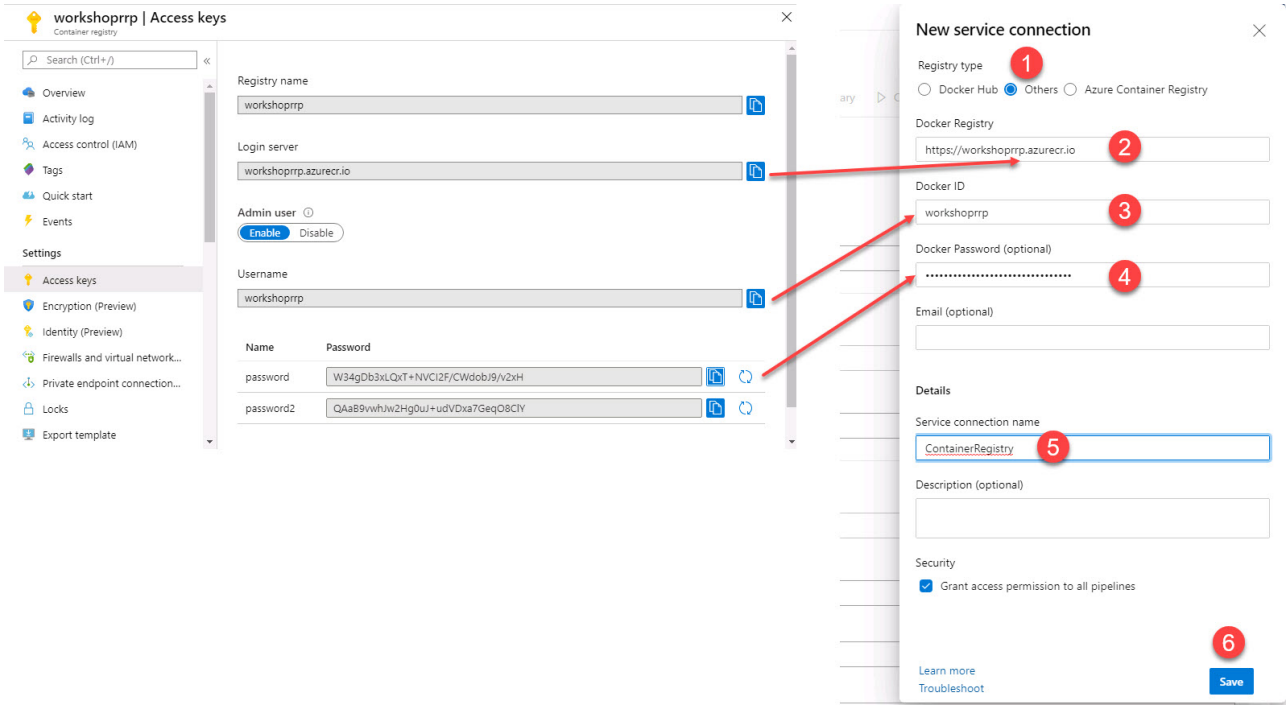
2. Next select **Create Service Connection**



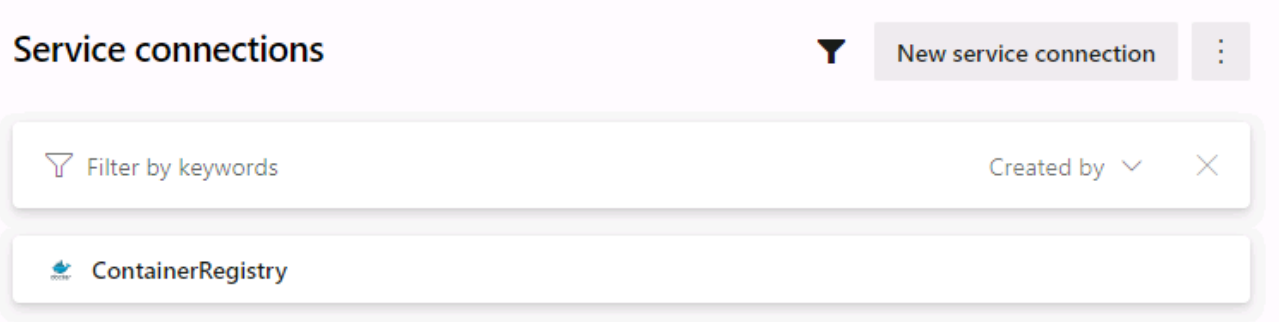
3. Select **Docker Registry** for New Service Connection Type and click next



4. Copy values from the new Azure Container Registry to the Service Connection



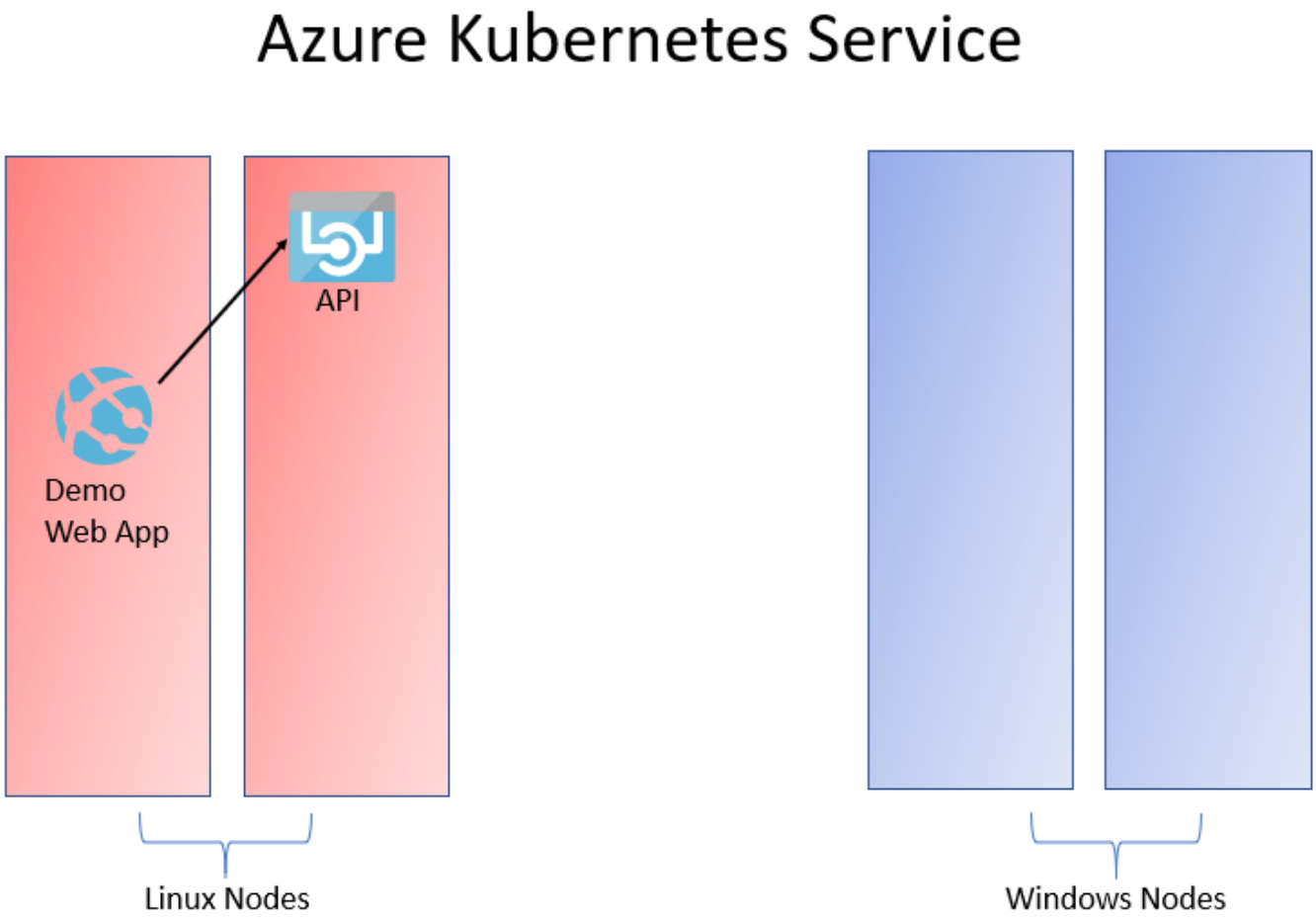
Use `ContainerRegistry` for the **Service Connection Name** and press **Save**



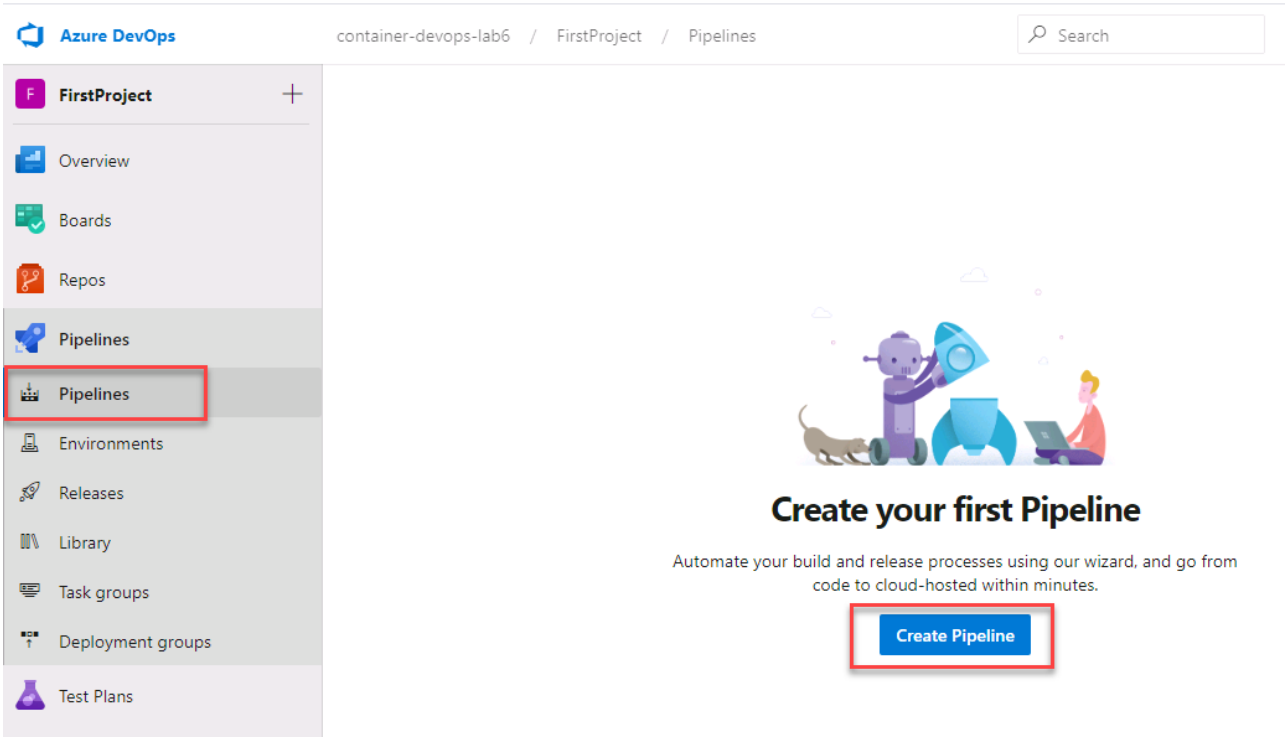
Create new build pipeline

> Note:If you want to avoid creating the pipeline step by step, you can skip this task and jump to Optional: Import a build pipeline.

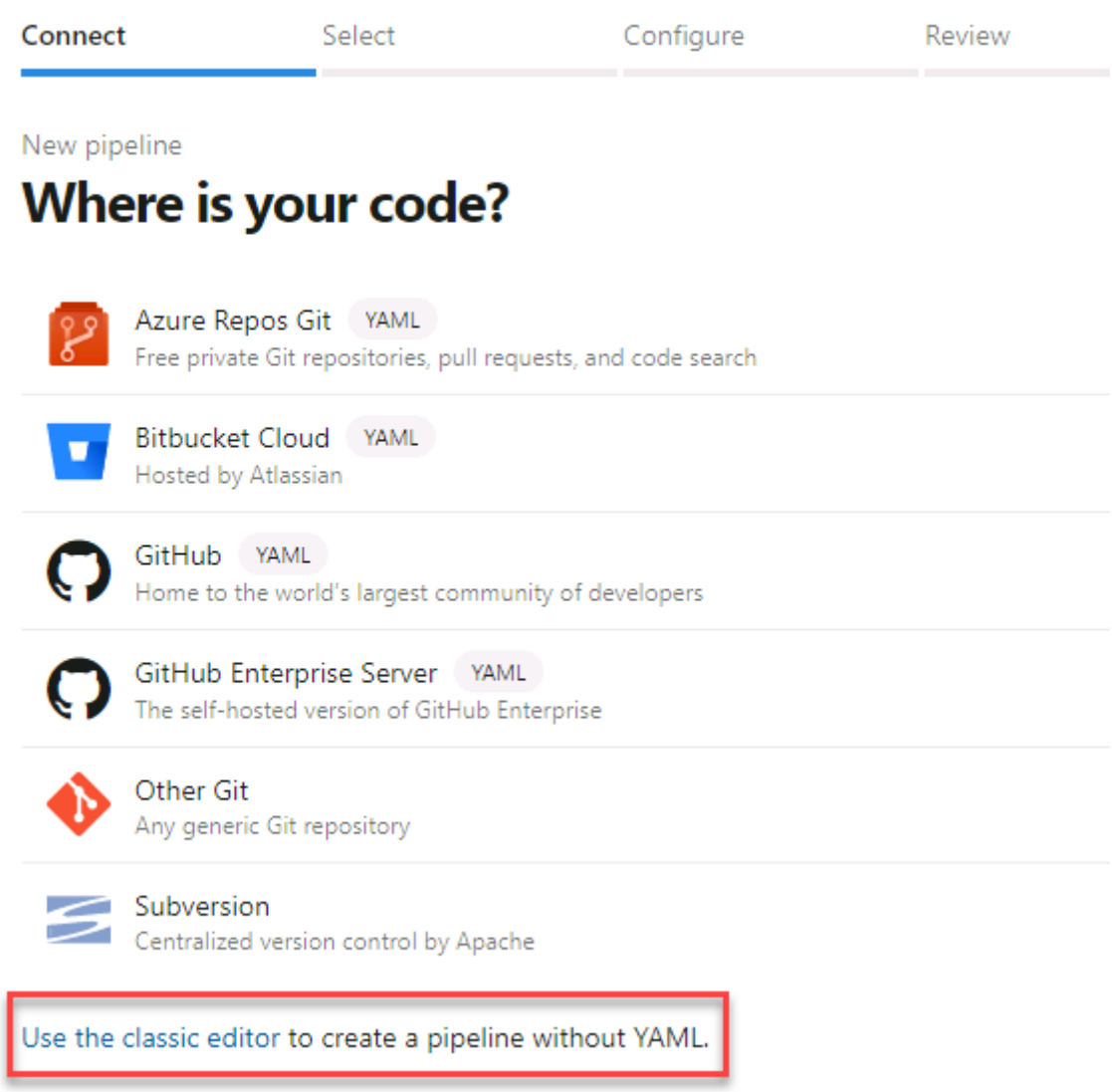
In this task, you are going to create a build pipeline which will be executed on a Linux agent and will produce two Linux container images for your Web App and Web API applications. These images will be tagged automatically with the appropriate build number and then pushed into your Azure Container Registry (ACR) instance.



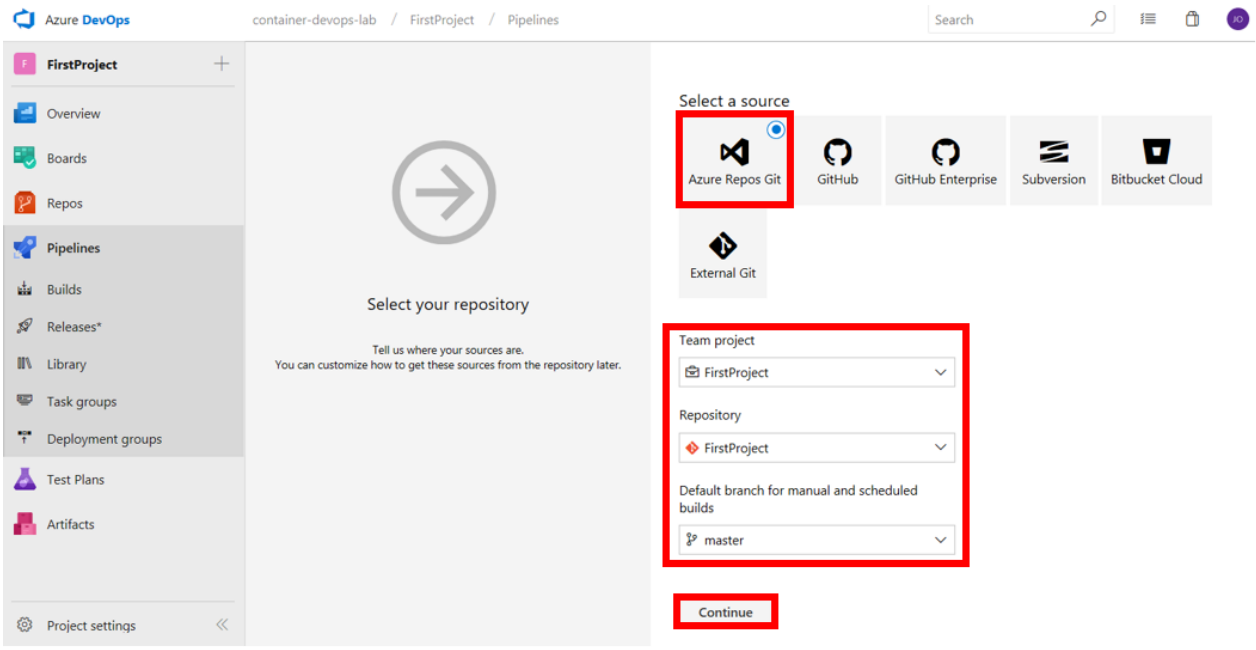
1. Navigate to **Pipelines** - **Builds** page and then click **New pipeline**.



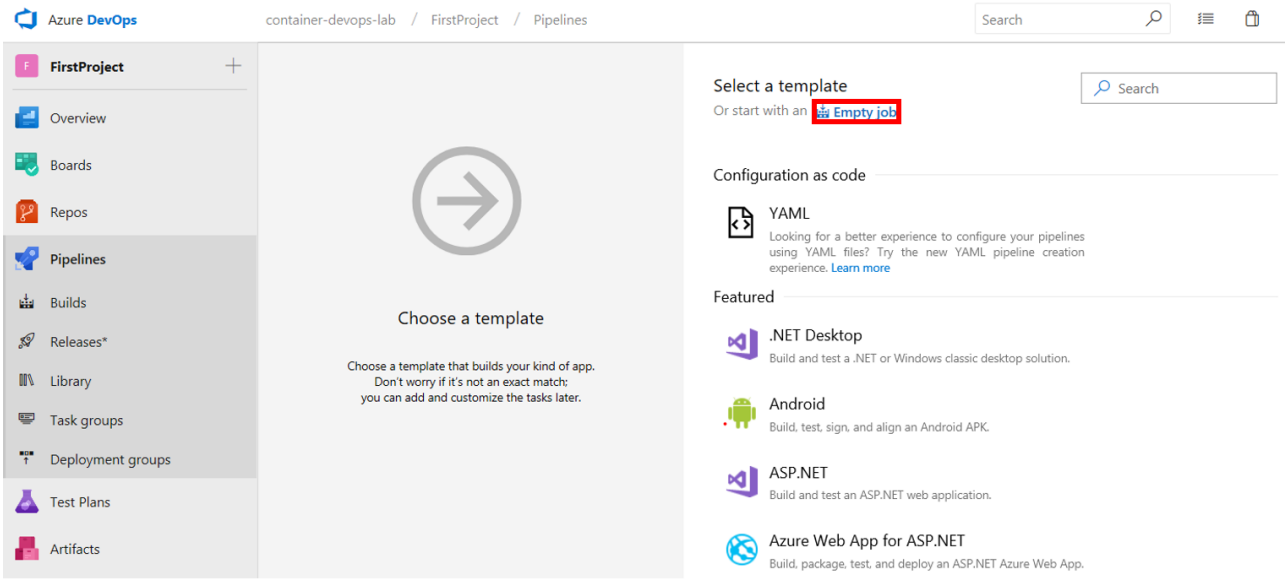
2. Select **Use the classic editor to create a pipeline without YAML** at the bottom of the "Where is your code" menu.



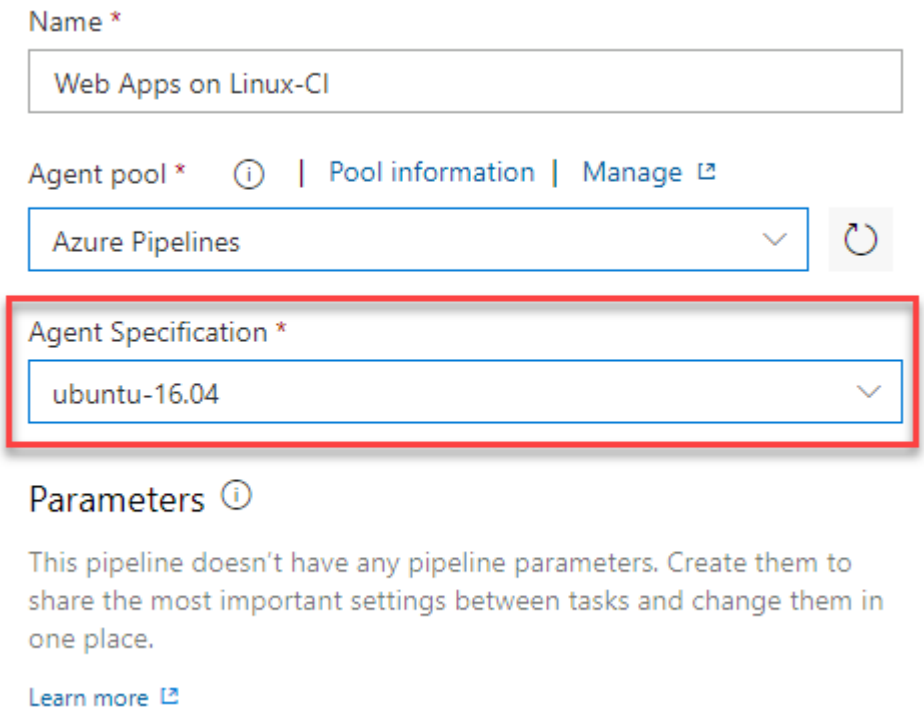
3. Select **Azure Repos Git** as your source and then select **Continue** to proceed.



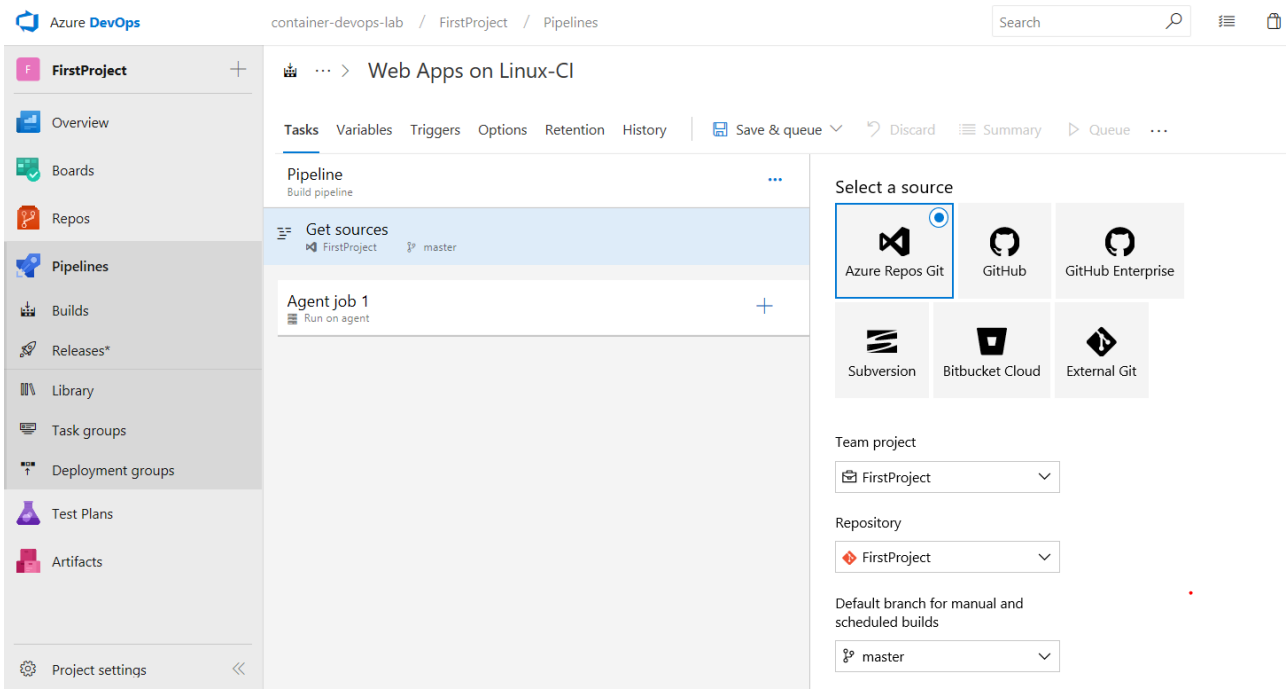
4. Select the **Empty job** option to start from a clean state.



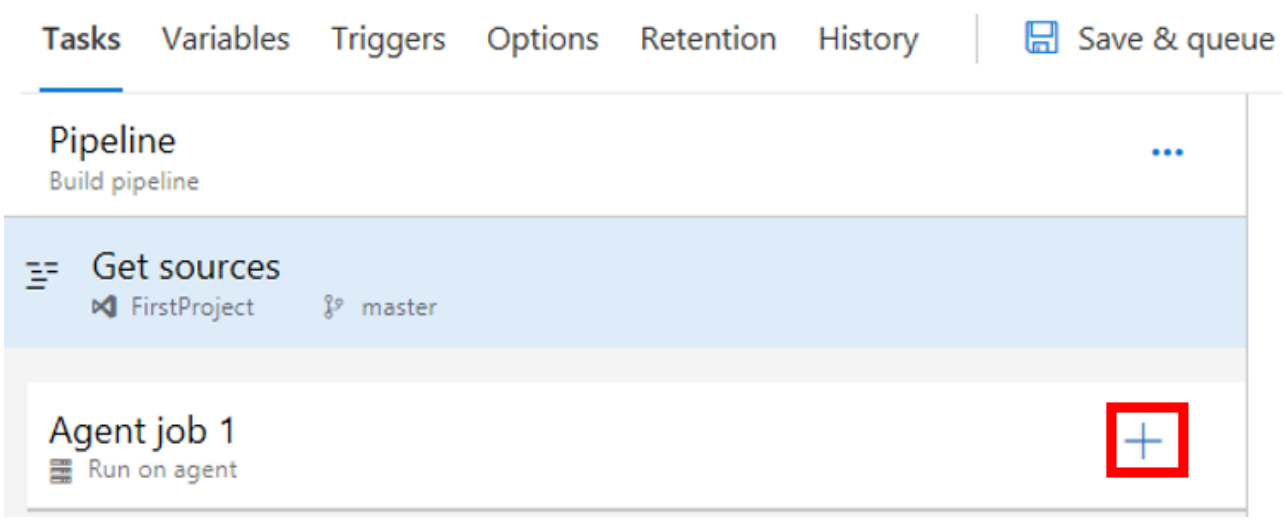
5. Before adding build tasks to the pipeline, you are going to select an agent queue containing agents which will be assigned to handle build requests. Select **ubuntu-1604** agent queue which contains agents installed on Linux OS.



6. Click on **Get sources** task and leave the default options.



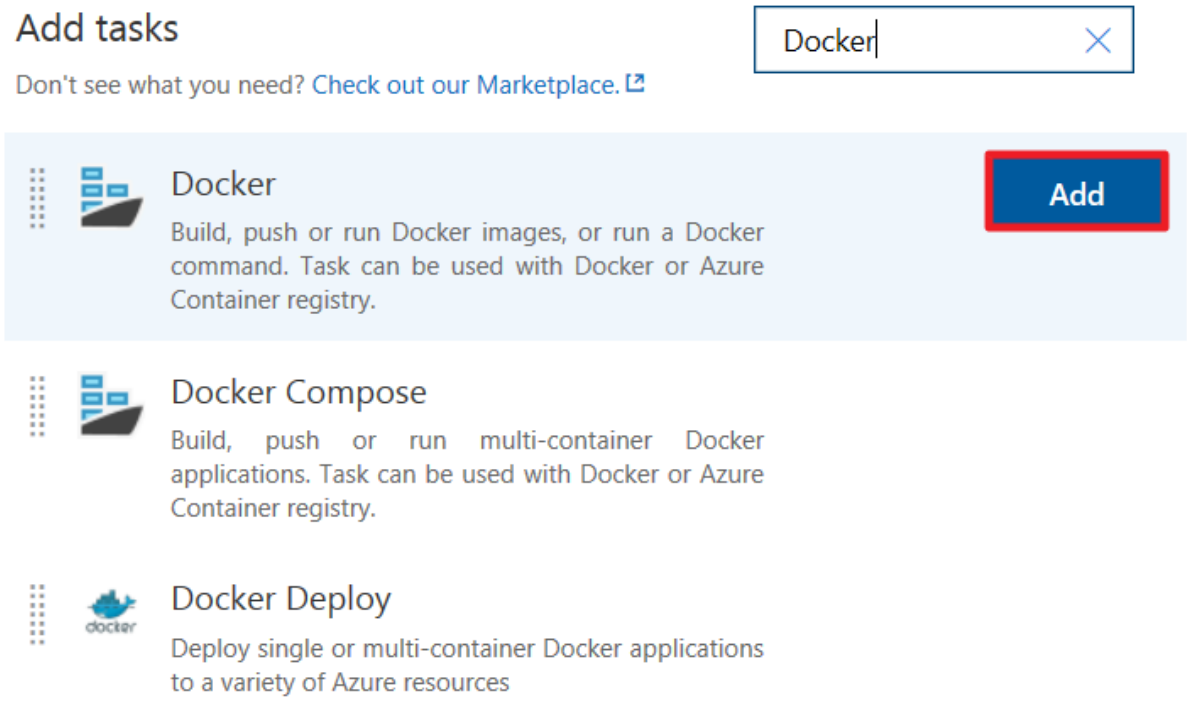
7. Now you will begin adding the necessary tasks to build the web application and web API. You will click the **+** sign to add new tasks:



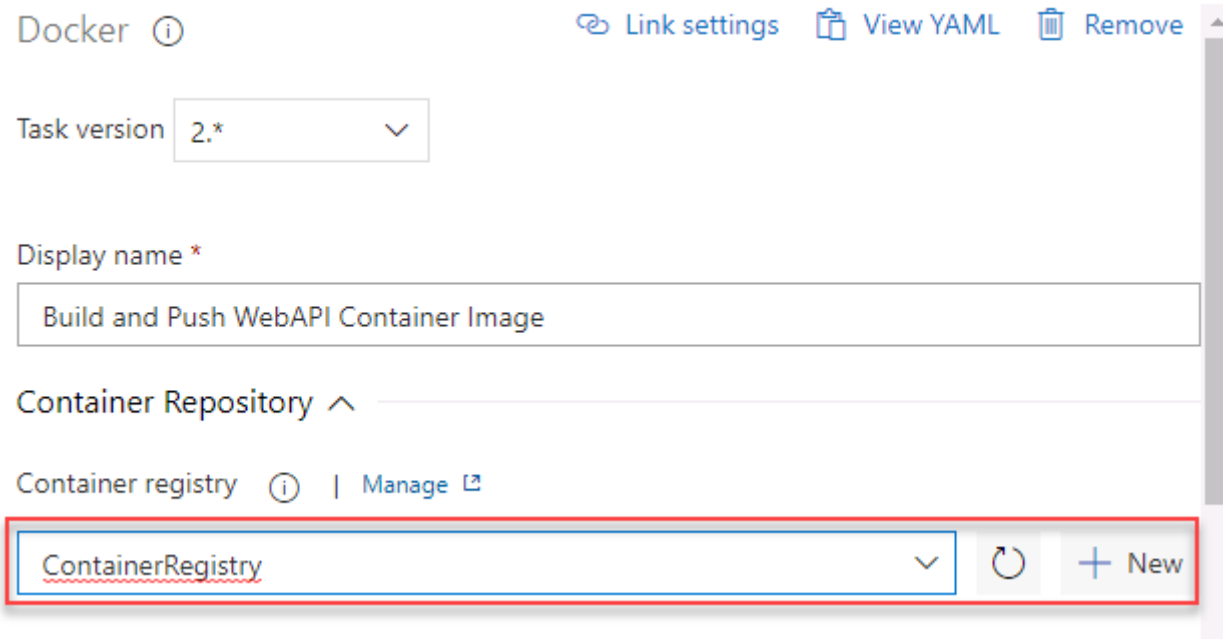
8. Normally, you might need to add **dotnet build** and **publish** steps, however these are included inside of the Dockerfile and are completed as part of the Docker build process. Just remember when you are creating your own CI/CD pipelines to either include the dotnet build and publish steps in the Dockerfile or the CI build pipeline. For your reference, here is the Dockerfile:

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build-env
WORKDIR /app
# Copy csproj and restore as distinct layers
COPY \*.csproj ./
RUN dotnet restore
# Copy everything else and build
COPY . ./
RUN dotnet publish -c Release -o out

# Build runtime image
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT \["dotnet", "mywebapp.dll"\]
```

9. Add a **Docker** task to build container image for web API application.



10. Select the newly added **buildAndPush** task and then proceed to the **Container registry** field. In the Dropdown select the previously created service connection **ContainerRegistry**

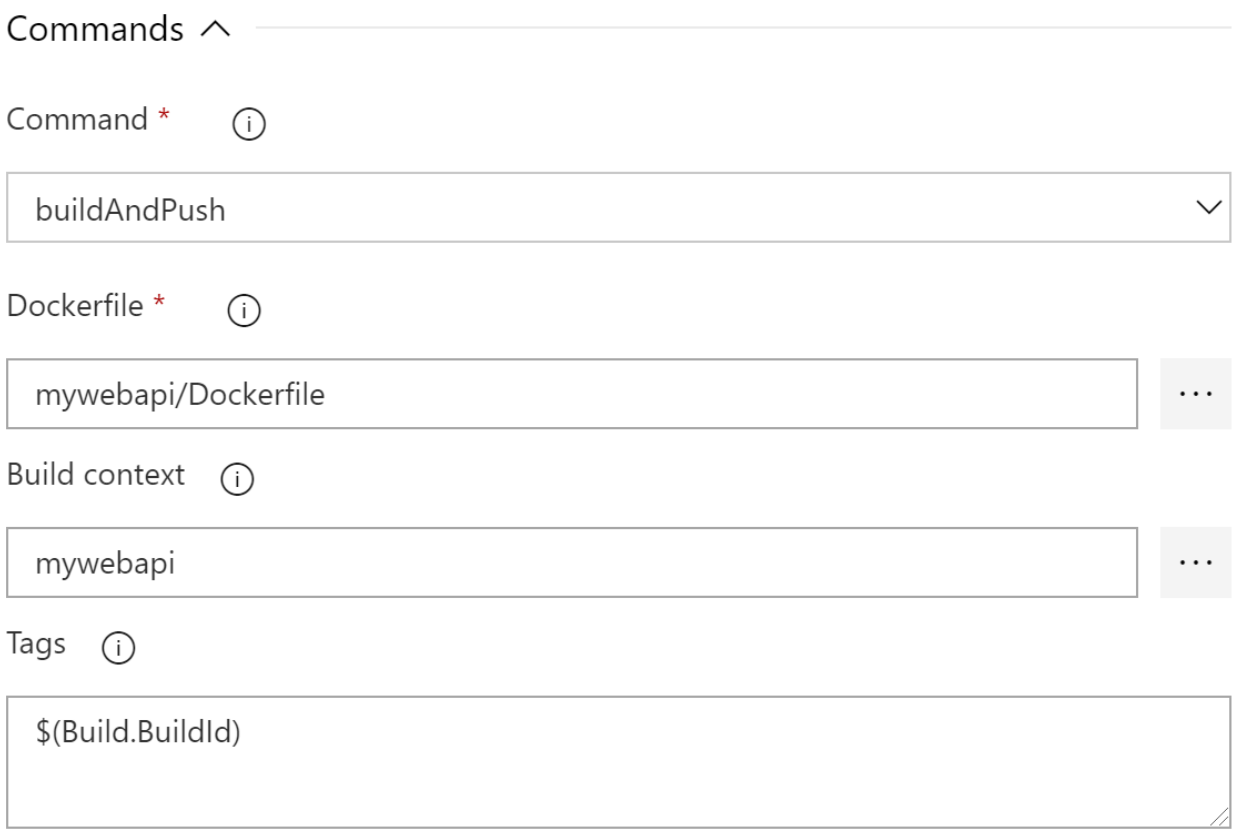


11. Enter the name of the specific repository, demo-webapi, within your ACR that can be used to retrieve the Web API image.

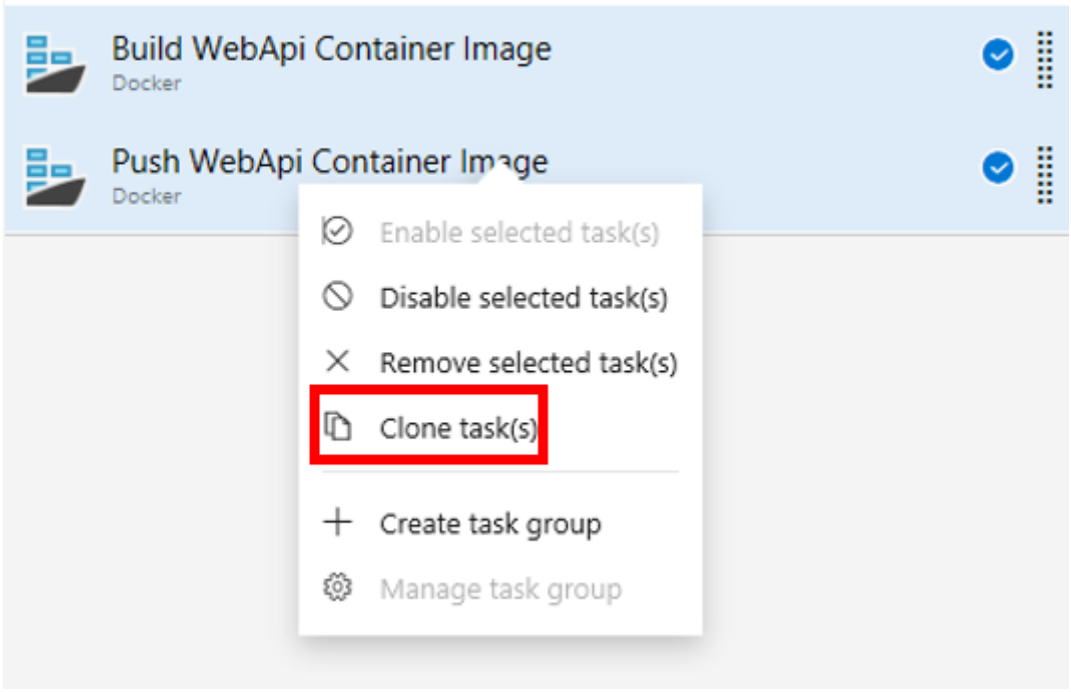
12. Configure this task's other properties as follows:

Property	Value
Display Name:	**+++Build and Push WebAPI Container Image**+++
Container Repository:	**+++demo-webapi**+++

Property	Value
Command:	**+++buildAndPush**+++
Dockerfile:	**+++mywebapi/Dockerfile**+++
Build Context:	**+++mywebapi**+++
Tags:	**+++$(Build.BuildId)**+++



13. Now you will add another Docker task for the **webapp** project. **Right-click** the build and release task, then choose **Clone task(s).**

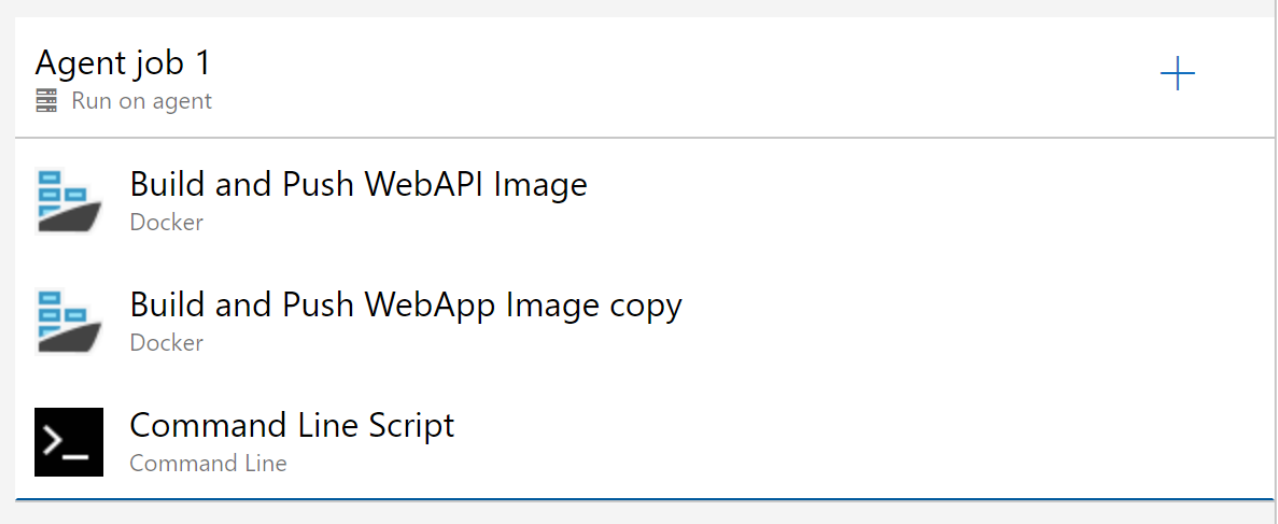


14. Change the **Build Web API Container Image copy** task to the following attributes:

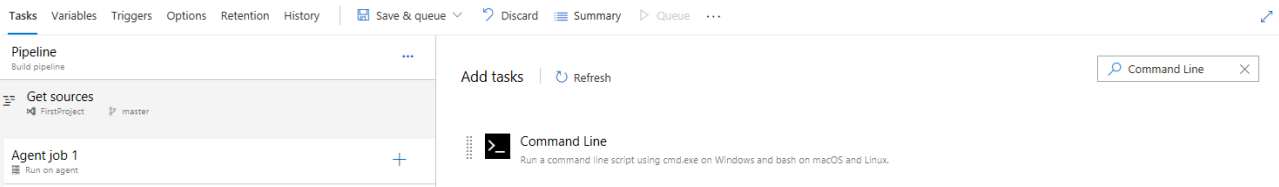
Property	Value
Display Name:	**+++Build and Push WebApp Container Image**+++
Container Repository:	**+++demo-webapp**+++
Command:	**+++buildAndPush**+++
Dockerfile:	**+++mywebapp/Dockerfile**+++
Build Context:	**+++mywebapp**+++
Tags:	**+++$(Build.BuildId)**+++

> Note:In order to retrieve these images from ACR and deploy them to the Kubernetes container orchestrator, you will need to use the same build number in the release pipeline. Therefore, you need to place the build number in the Kubernetes deployment configuration files: **backend-webapi.yaml** and **frontend-webapp.yaml**

15. Add a **Command Line Script** task after the newly created Docker tasks.

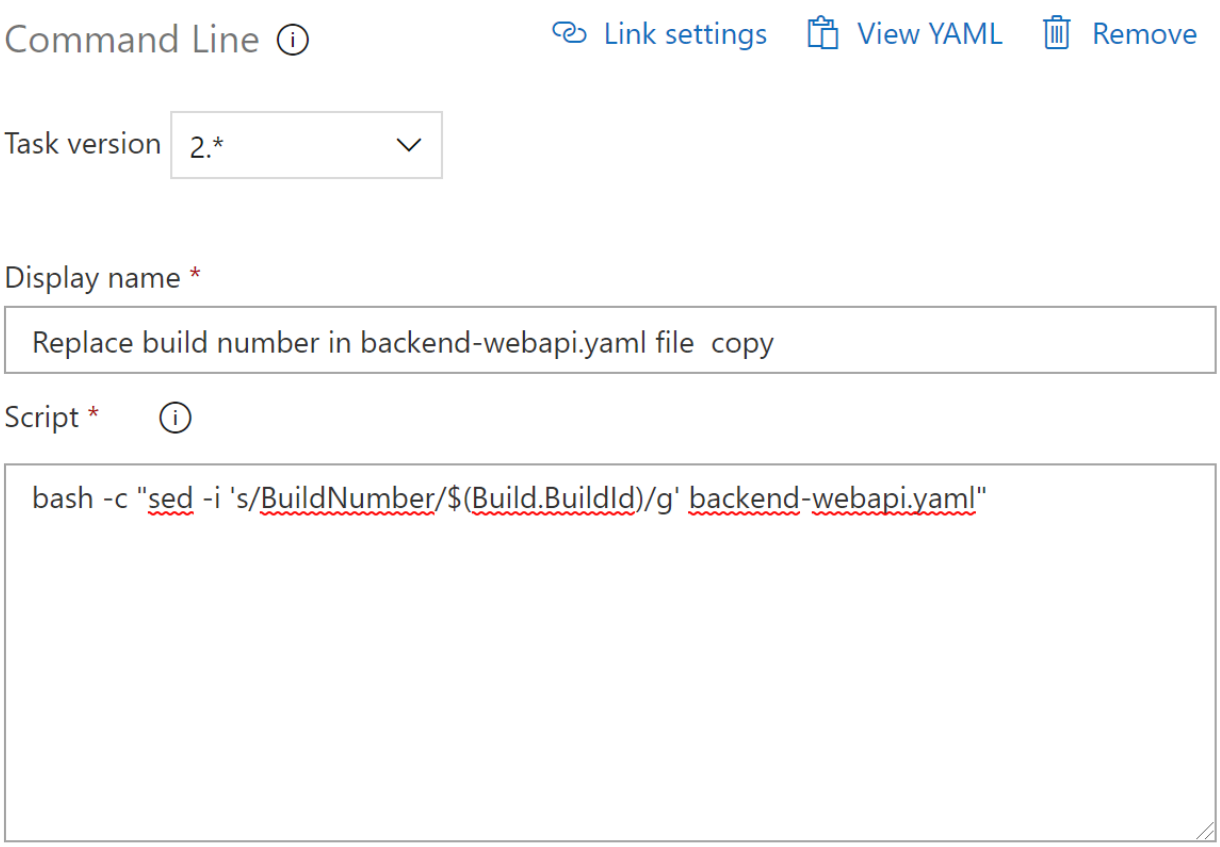


Configure the first **Command Line Script** task as follows:



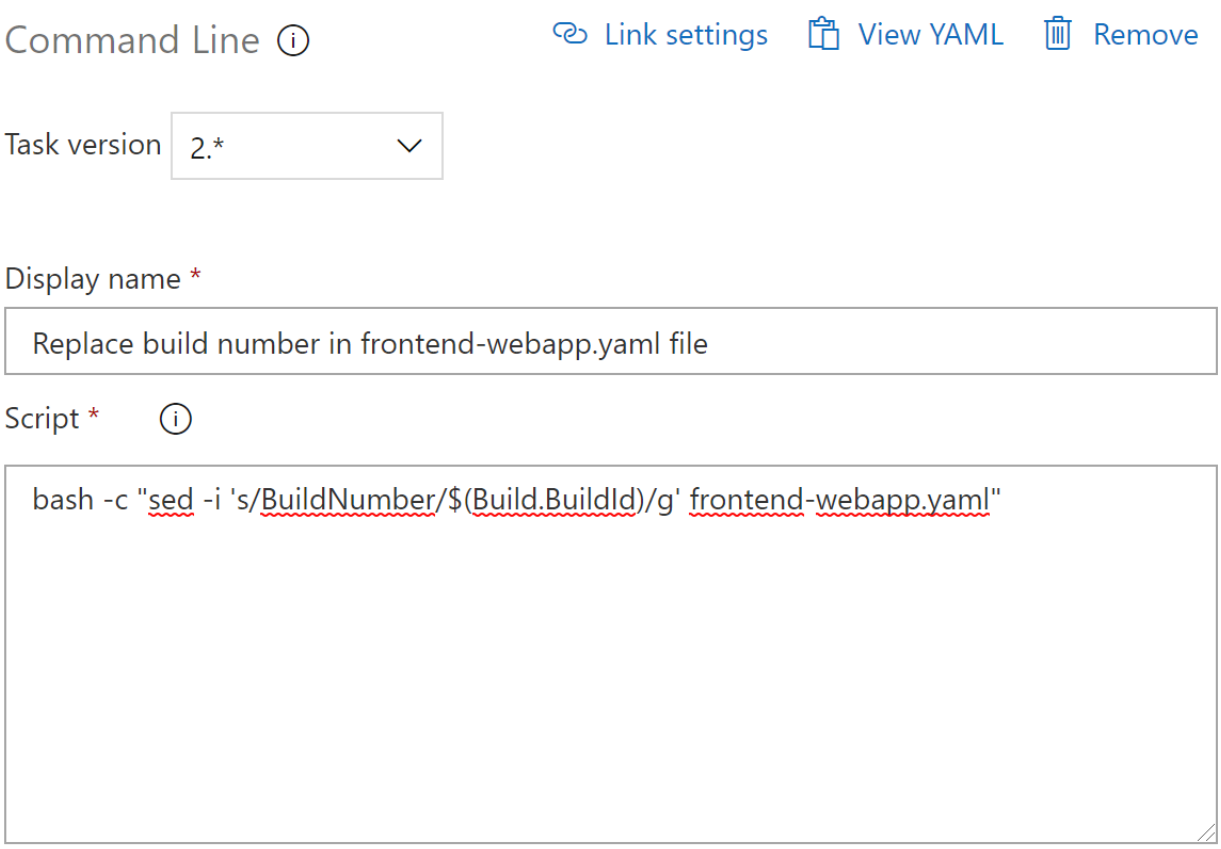
Configure the first **Command Line task** as follows:

Property	Value
Display Name:	**+++Replace build number in backend-webapi.yaml file+++**
Script:	**+++bash -c "sed -i 's/BuildNumber/$(Build.BuildId)/g' backend-webapi.yaml"+++**



16. Clone your first command line task, and configure the second **Command Line task** as follows:

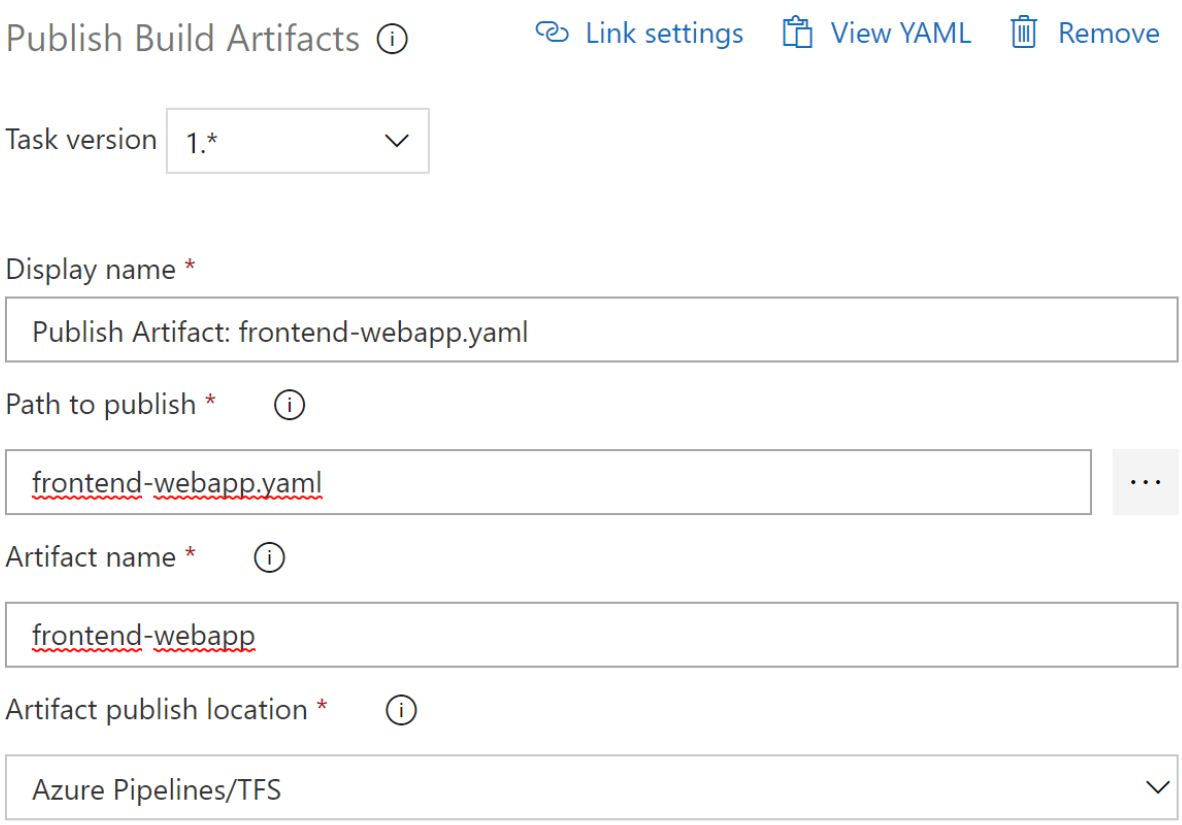
Property	Value
Display Name:	**+++Replace build number in frontend-webapp.yaml file+++**
Script:	**+++bash -c "sed -i 's/BuildNumber/$(Build.BuildId)/g' frontend-webapp.yaml"+++**



> Note:The previous task ensures that the yaml file's content is updated with the appropriate build number, now it is time to upload it as build artifact to be used in release pipeline later.

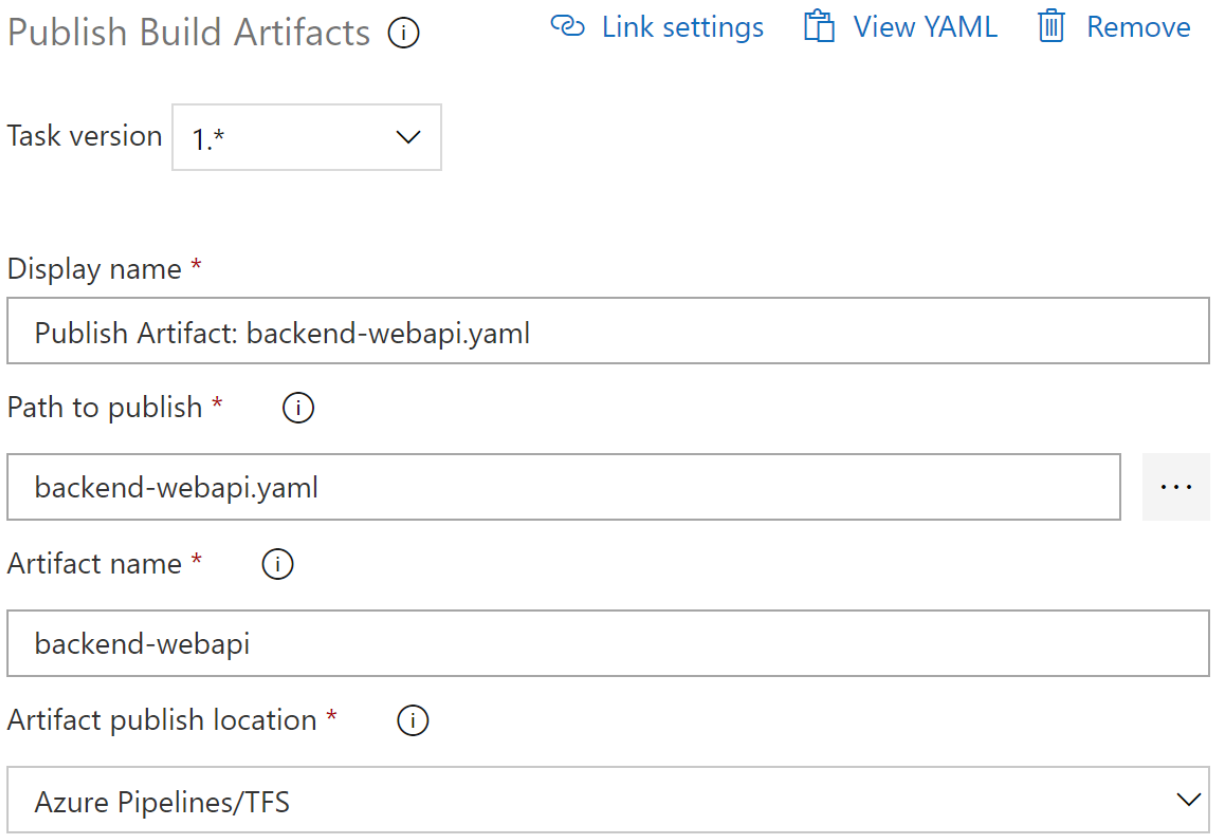
17. Add a new **Publish Build Artifacts** task and configure as follows:

Property	Value
Display Name:	**+++Publish Artifact: frontend-webapp.yaml+++**
Path to Publish:	**+++frontend-webapp.yaml+++**
Artifact Name:	**+++frontend-webapp+++**



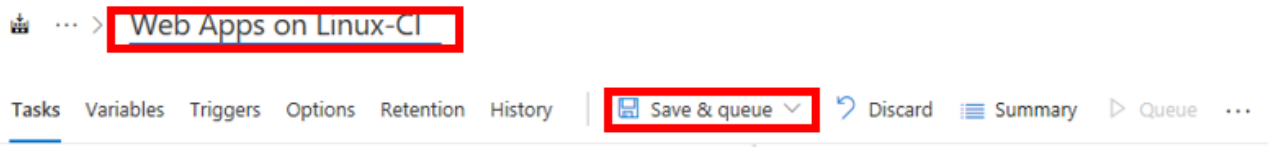
18. Clone your **Publish Build Artifacts** task and modify it as follows:

Property	Value
Display Name:	**+++Publish Artifact: backend-webapi.yaml**+++
Path to Publish:	**+++backend-webapi.yaml**+++
Artifact Name:	**+++backend-webapi**+++



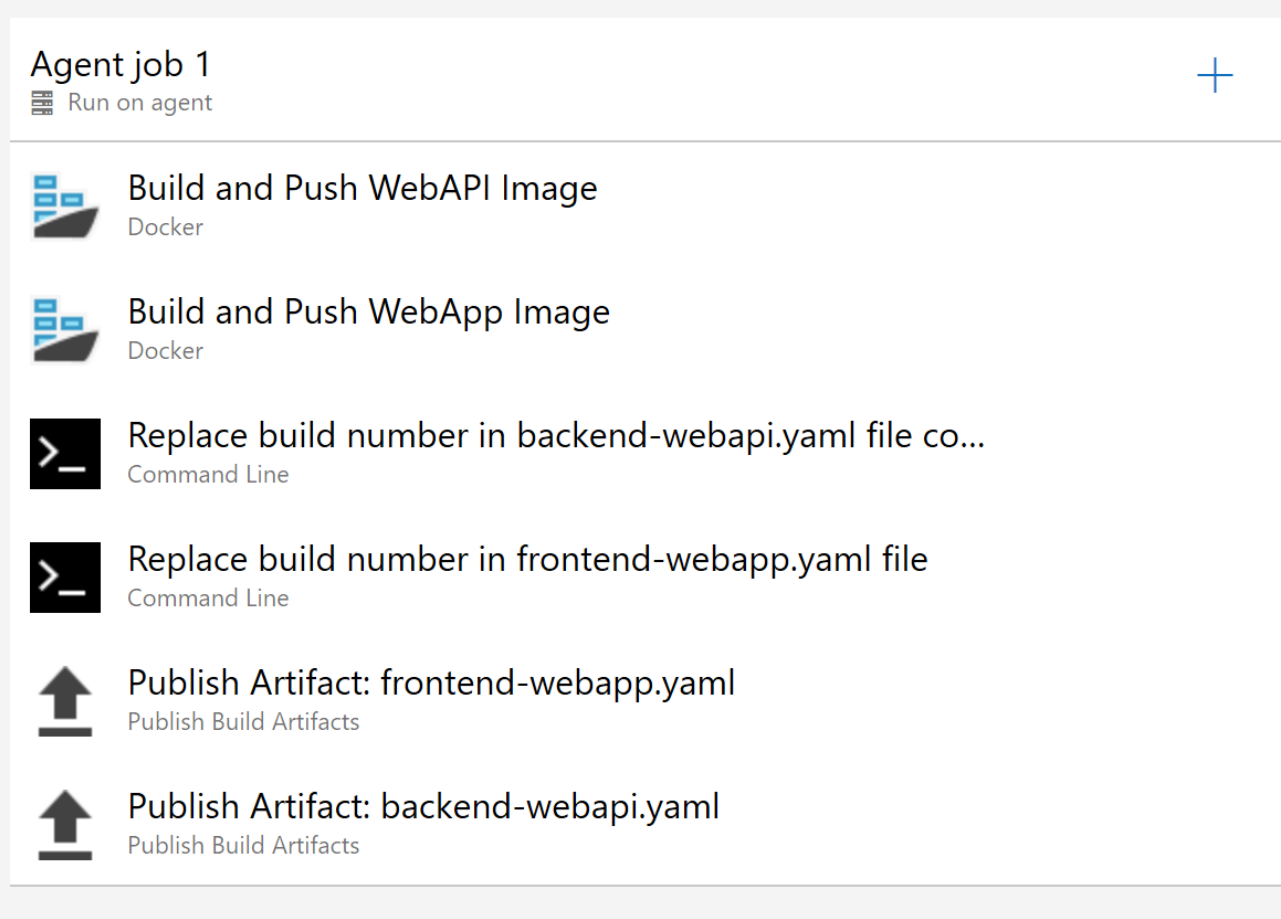
Finally change the build pipeline's name to `Web Apps on Linux - CI`. Click **Save & queue** to save and trigger a build.

> Note: You will see a pop-up asking for a comment, you can leave it blank and press Save & Queue on the popup to continue.



19. Your finished build should look like this:

> Note: If you get an error please click on the step with the error and try debugging or the instructor will come by and help you.



Optional: Import a build pipeline

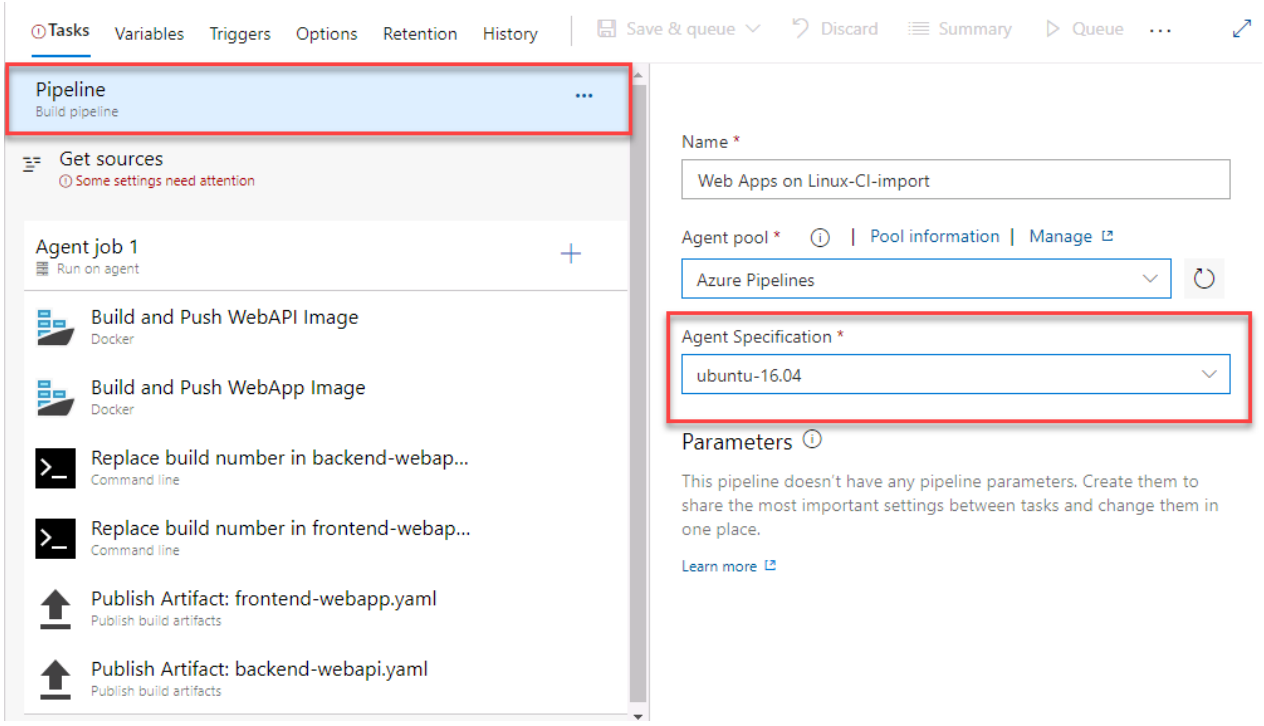
> Note: Skip this optional task if you already created the build pipeline.

1. Navigate to **Pipelines** - **Builds** and click **New** - **Import a pipeline**.

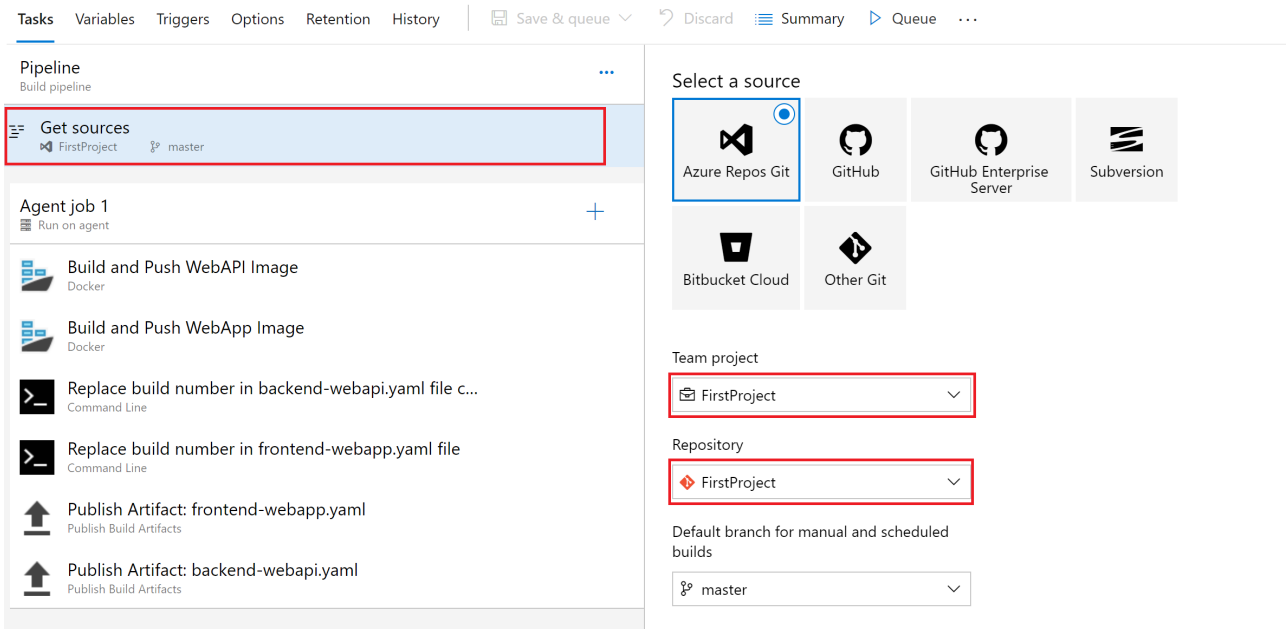
   > Note: Currently, we can only do this when at least one pipeline exists. So, you will need to create an empty pipeline first.

2. Select **C:\labs\module6-ext\Web Apps on Linux-CI.json** and click **Import**

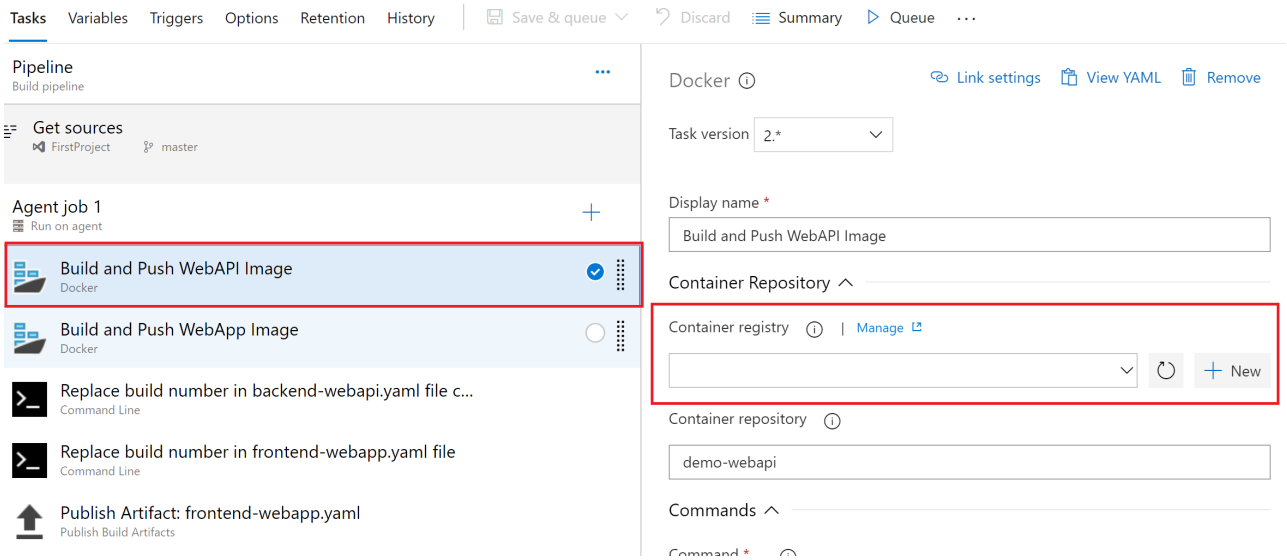
3. Select **ubuntu-1604** as an agent specification



4. Click on **Get sources** and make sure that correct Git repository and branch is selected.



5. Select the docker tasks one by one and enter the subscription and ACR information.



6. Make sure the Build pipeline is named `Web Apps on Linux-CI` to stay aligned with the following screenshots. Click **Save & queue**.

Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 4: Create Release Pipeline for AKS Cluster

In this exercise, you will create a Release Pipeline which will pull container images from ACR (result of a previous build task), and then deploy these images as containers to a Kubernetes cluster on AKS.

> Note:If you want to avoid creating the pipeline step by step, you can skip this task and jump to Optional: import a release pipeline.

The build pipeline is used for compiling the application, building a container image(s), and then pushing the container image(s) to a container registry. The release pipeline, on the other hand, is used to pull the container image(s) from a container registry and then deploy them as containers to the cluster.

Return to list of exercises - Return to list of modules

1. Navigate to https://dev.azure.com/<YourAccountName>/<YourProjectName> and go to the **Pipelines** - **Releases** page and click **New pipeline**.

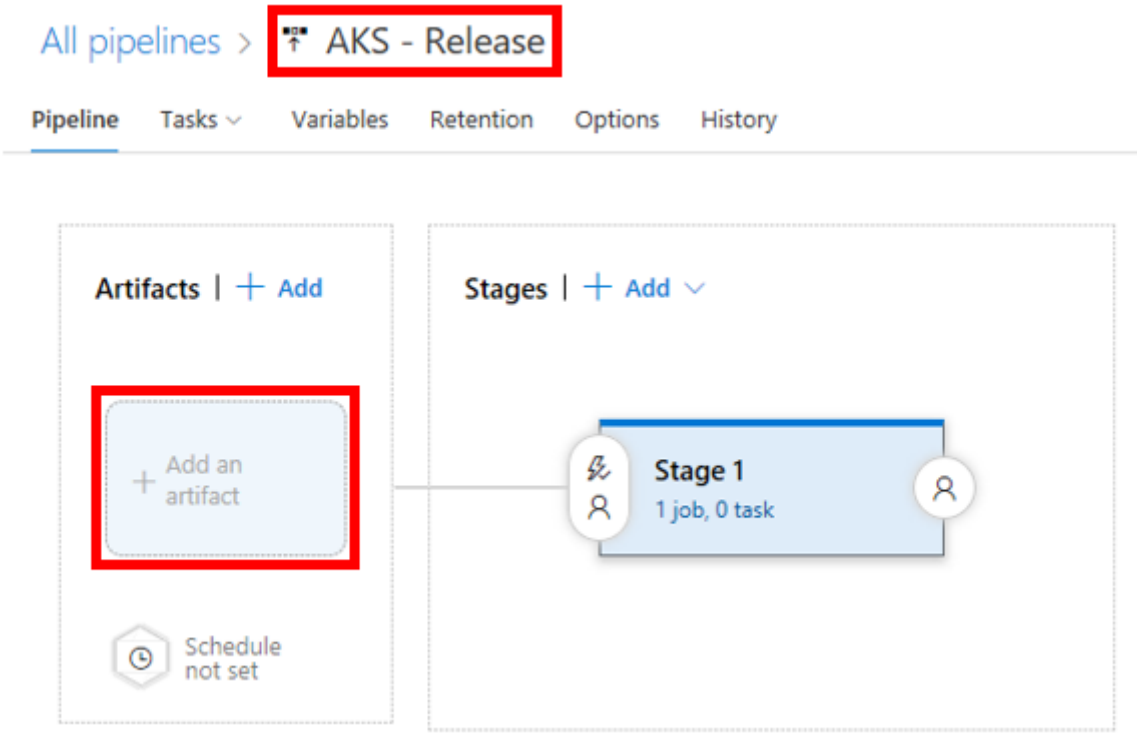
No release pipelines found

Automate your release process in a few easy steps with a new pipeline

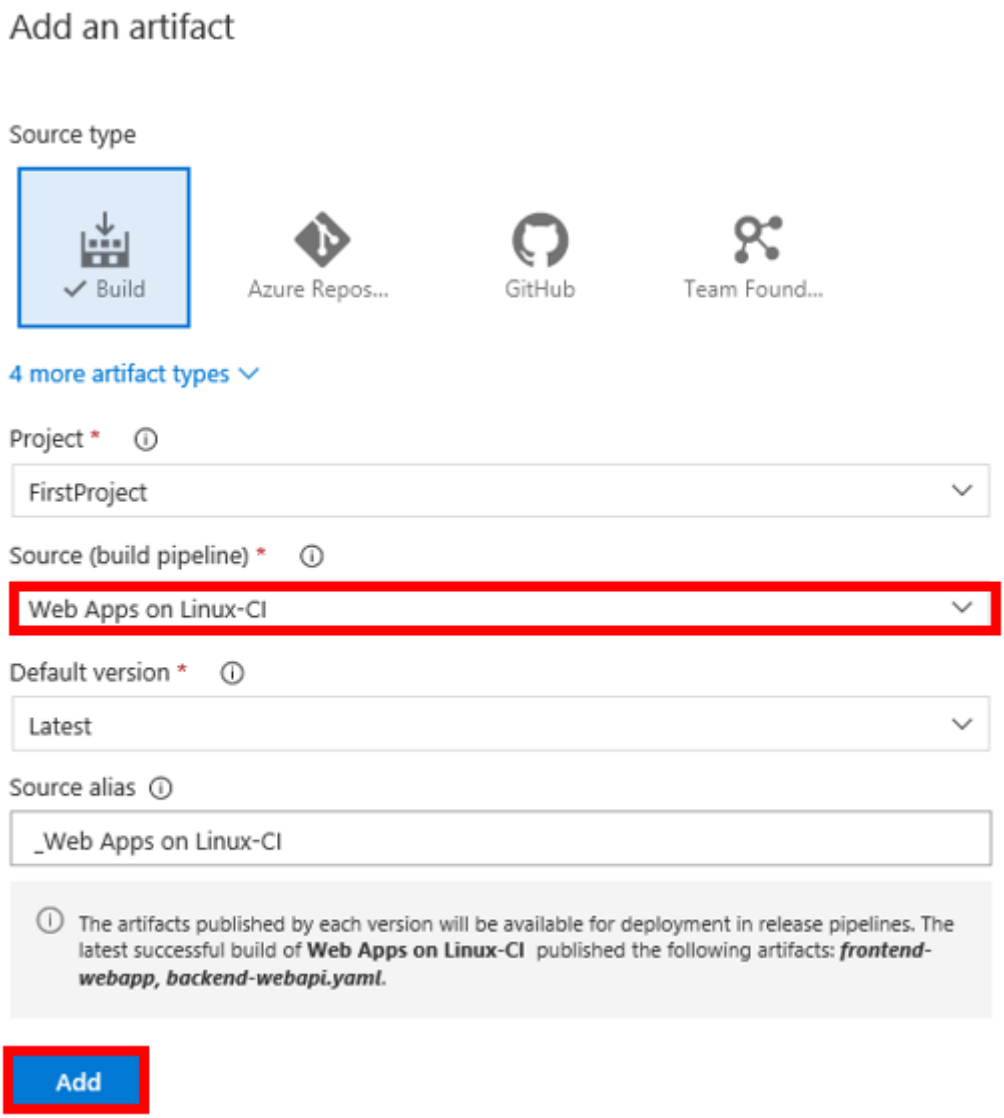
New pipeline

2. Although there are templates for working with Kubernetes deployments, you are going to start with the empty template. Click **Empty job** to create an empty release pipeline.

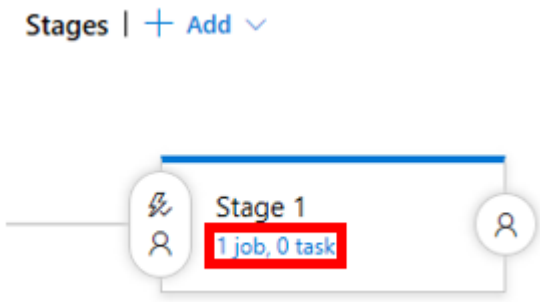
3. Click on the release pipeline name and change it to `AKS - Release`. Then click on **Add an artifact** to link this release pipeline to the build artifacts from the build pipeline.



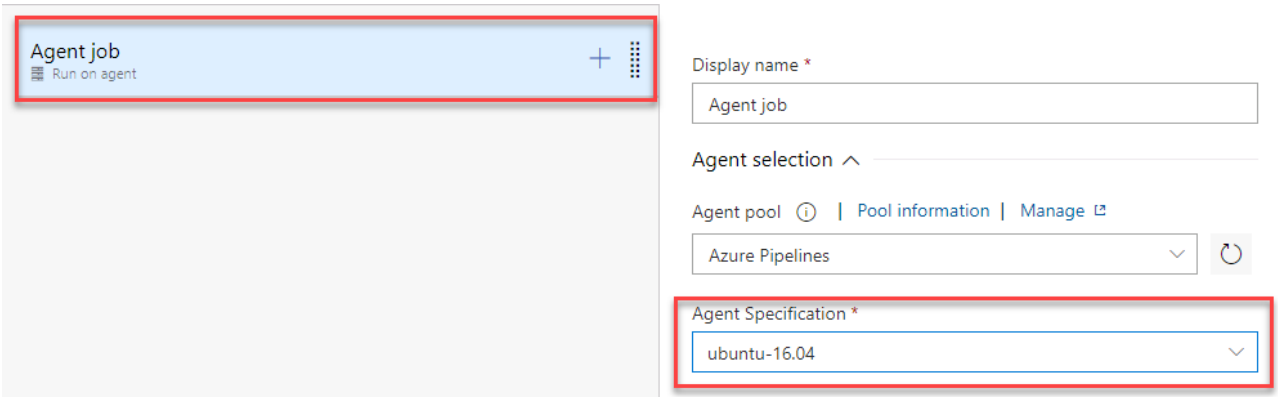
4. On the **Add artifact** popup, select **Web Apps on Linux-CI** build pipeline. Finally, select **Add**.



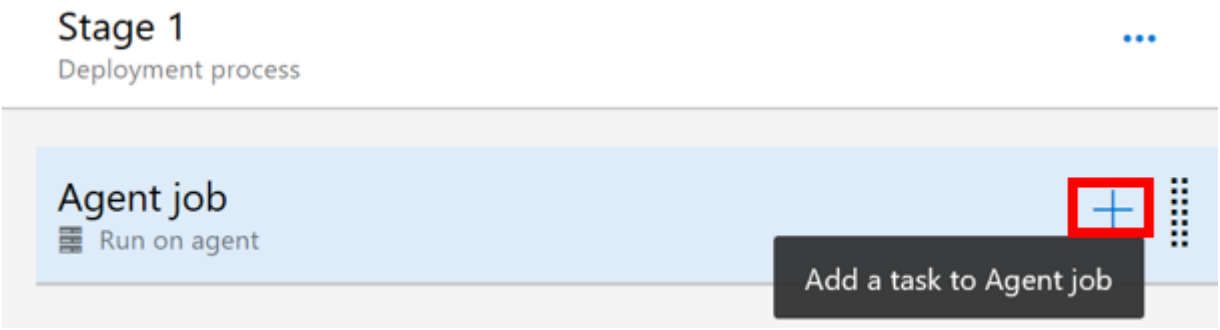
5. Click **1 job, 0 task** link available inside the **Stage 1**.



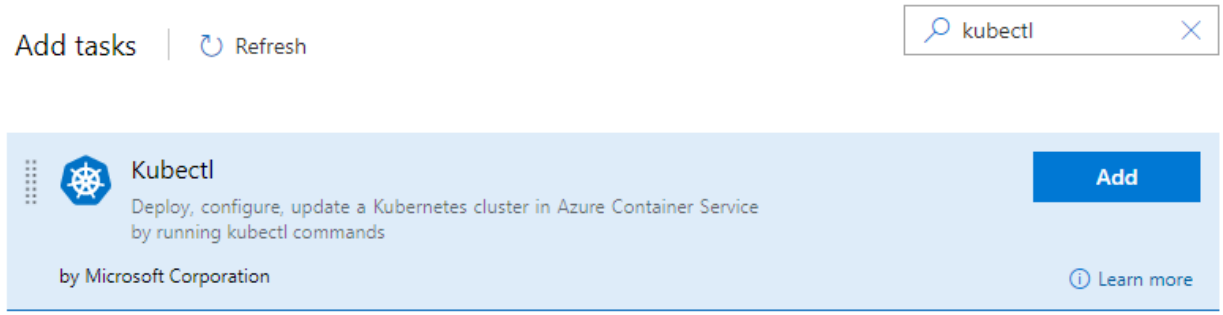
6. Click on the **Agent job** and then select **ubuntu-1604** from the Agent Specification dropdown.



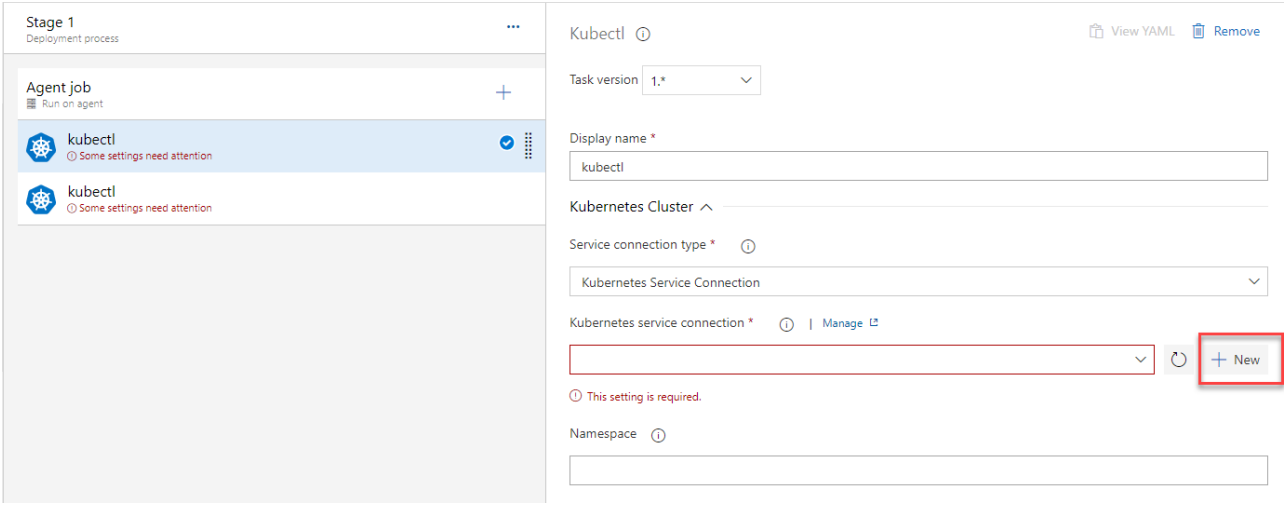
7. Now you are ready to add tasks to deploy your containers to the Kubernetes cluster. Click the **plus** button on **Agent phase**, and then find **kubectl** task.



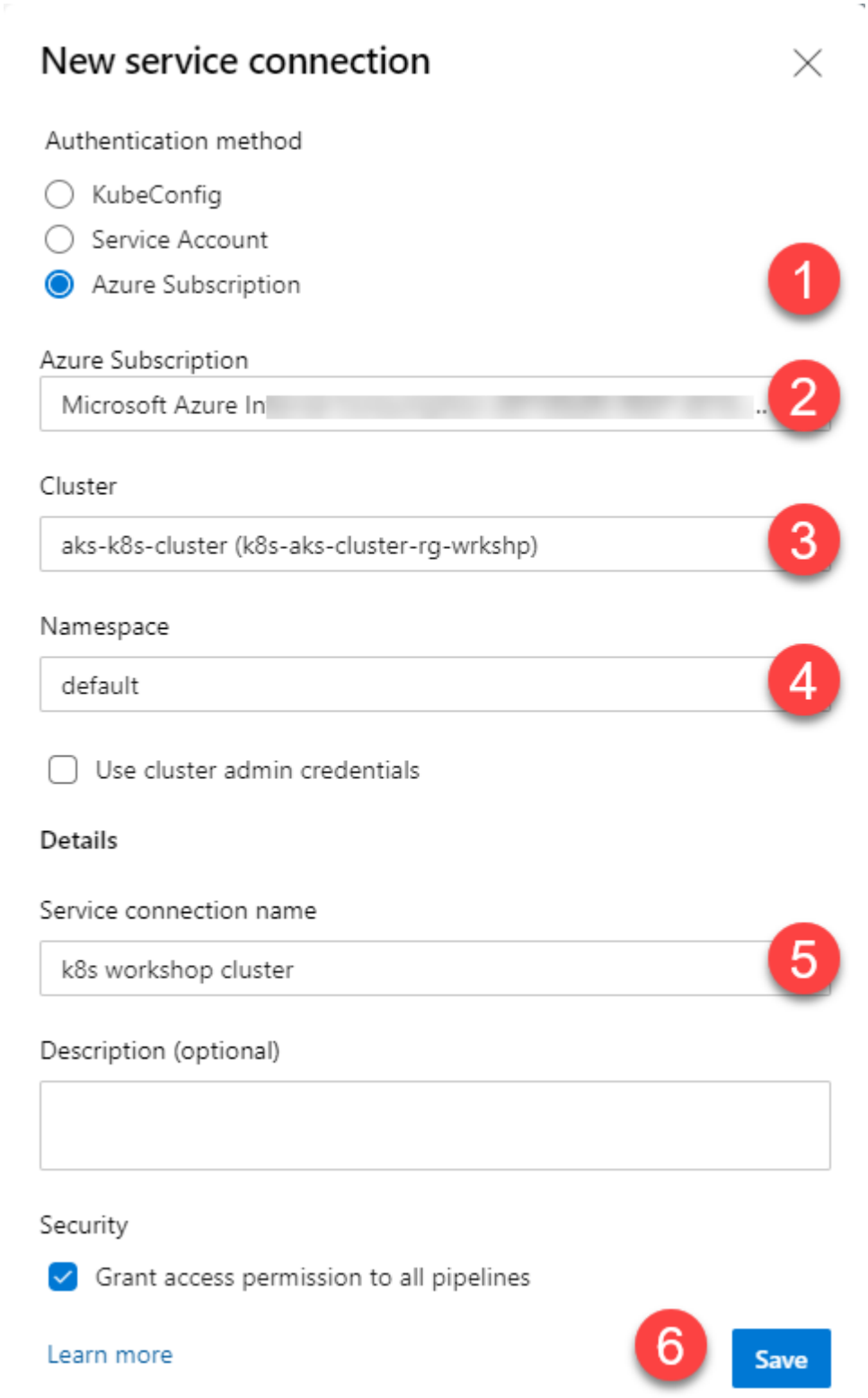
Click **Add** to add the task. Make sure to add two copies of this task, one for deploying **webapi** and another for **webapp**.



8. Change the first task name to `kubectl apply webapp`. Next, add a **Kubernetes Service Connection** by clicking on the "+ New" button



9. Add a connection to your Kubernetes Cluster. Choose your Azure subscription and the Kubernetes cluster created in the previous module. Set the Namespace to `default**+++ and the Connection Name to +++**k8s workshop cluster`. Finally, click **save**

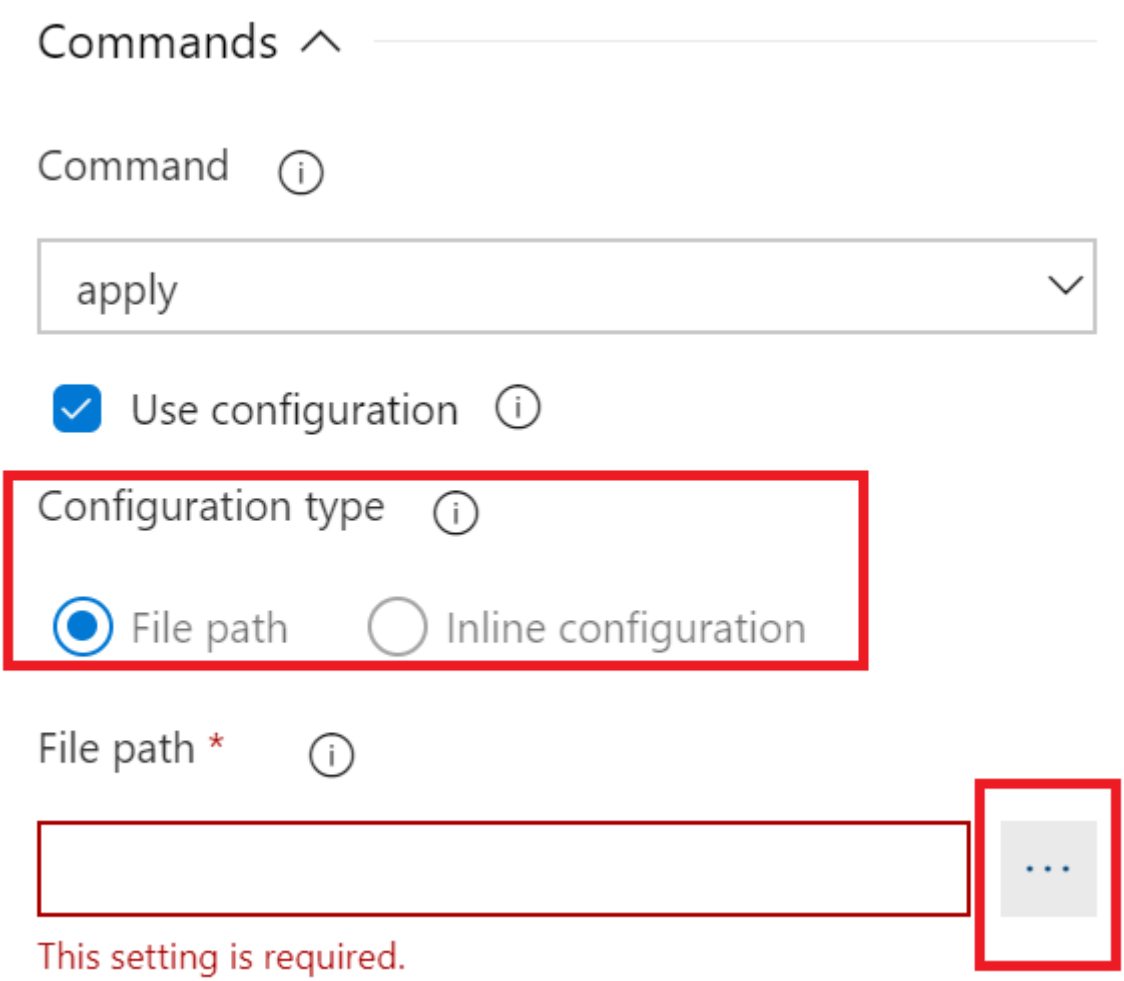


> Knowledge:For more information regarding the various connection types, and using helm charts for deployment, visit https://docs.microsoft.com/en-us/azure/devops/pipelines/apps/cd/deploy-aks?view=azure-devops.

Also note: we are using the default namespace for deploying to our AKS cluster for demo purposes. The best practice is to always specify a namespace. For additional information on AKS best practices, visit https://docs.microsoft.com/en-us/azure/aks/best-practices

10. In the **Commands** section:

- Check the box for **Use configuration files** and click on the 3 dots to the right of the input box to explore files

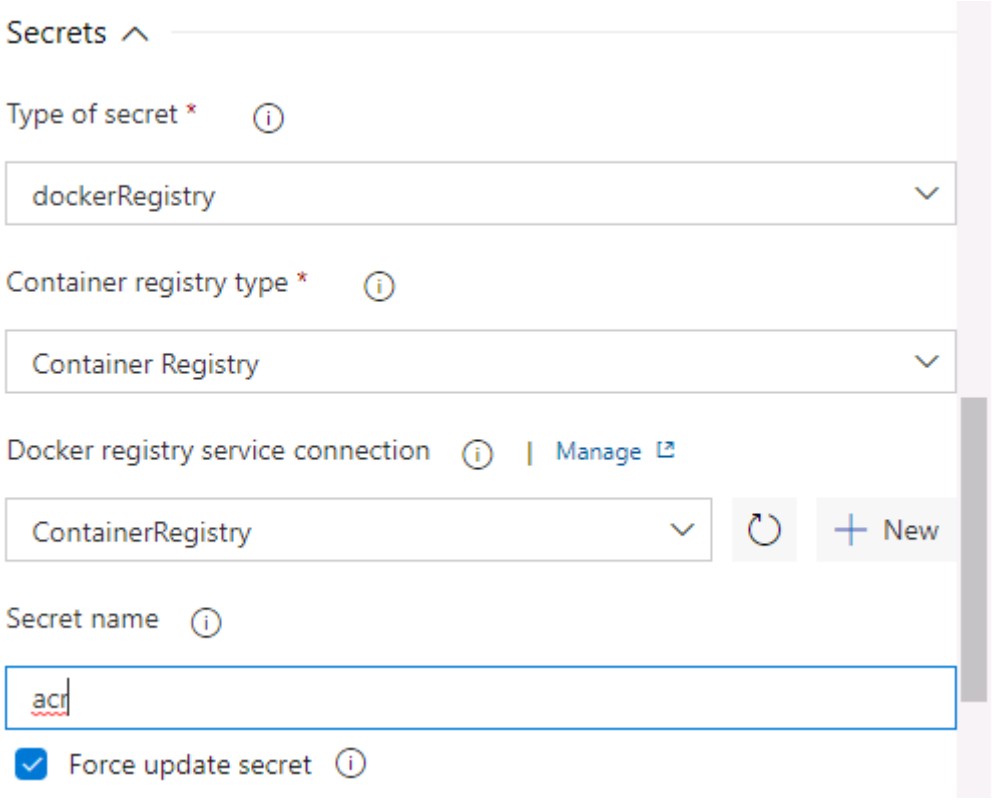


- Select the main folder, and in the **frontend-webapp** folder, select **frontend-webapp.yaml**, then hit **OK**.

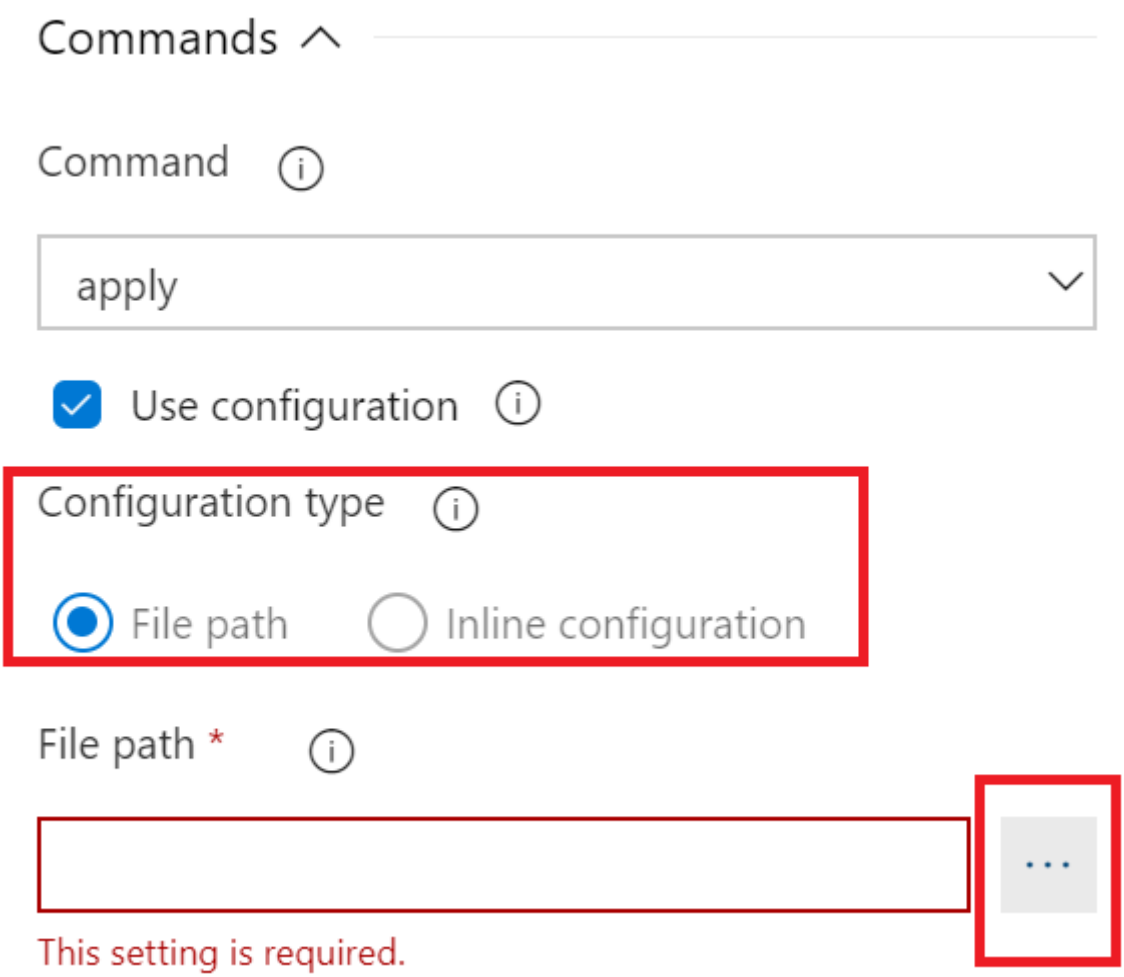
11. Now fill other parameters in the **Secrets section** of the task:

Property	Value
Type of secret:	dockerRegistry
Container Registry type:	Container Registry
Docker Registry service connection:	Select **ContainerRegistry** name created in the build pipeline

Property	Value
Secret name:	`acr` This is the secret which will be created using your ACR credentials, and will be stored in the Kubernetes cluster. You will notice this is same as the **imagePullSecrets** parameter in the yaml deployment files **frontend-webapp.yaml** and **backend-webapi.yaml**
Force update secret:	**Selected** This parameter will delete the secret and recreate it, and prevent getting errors caused by existing secret with same name



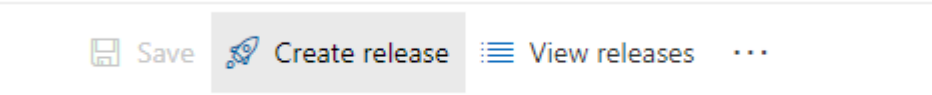
12. Similarly, configure second Kubernetes task:
    - **Display Name**: `kubectl apply webapi`
    - **Select your AKS cluster**
    - Check the box for **Use Configuration files**, and then the below input box will open:



- Click on the 3 dots to the right of the input box to explore files, select the main folder, select backend-webapi folder, select **backend-webapi.yaml**, then hit OK.
- **Container Registry type**: `Container Registry`
- **Docker Registry Service Connection**: Select the name **ContainerRegistry** created earlier in the build pipeline
- > Note:This registry should be the same as the one you used in your build pipeline, since container images will be pulled from this registry
- **Secret name**: `acr`

  > Note:This is the secret which will be created using your ACR credentials, and which will be stored in Kubernetes cluster. You will notice that this is same as **imagePullSecrets** parameter in yaml deployment files **frontend-webapp.yaml** and **backend-webapi.yaml**
- **Force update secret**: Selected

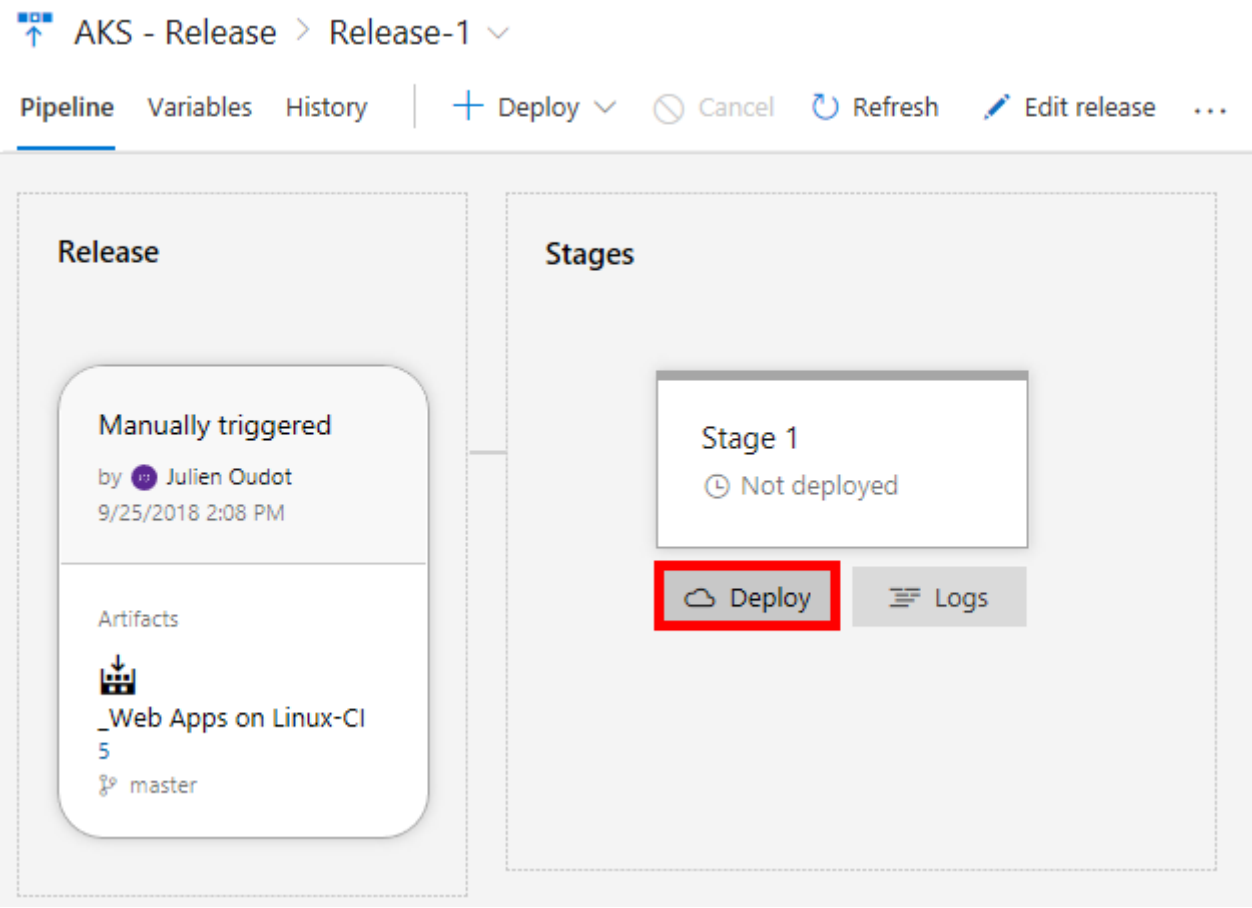
13. Look at the top right of your page and you will see the buttons as in the screenshot below. First, Save the release pipeline. Then click on the **Create Release** button to trigger a release.



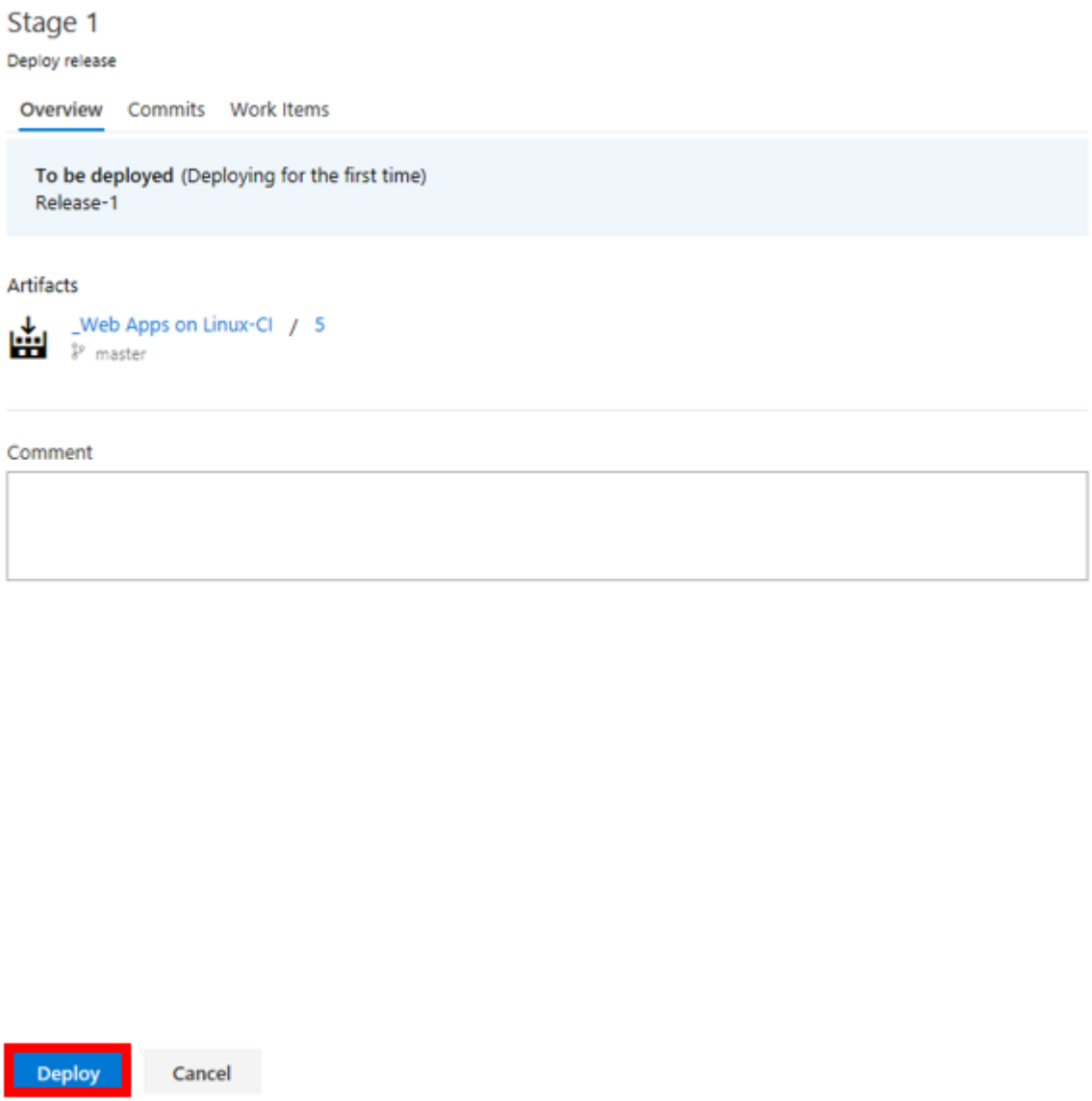
14. On **Create a new release** window select the Stage 1 from the dropdown. Keep all defaults and press **Create**.

```
![](content/mod6image56.png)
```

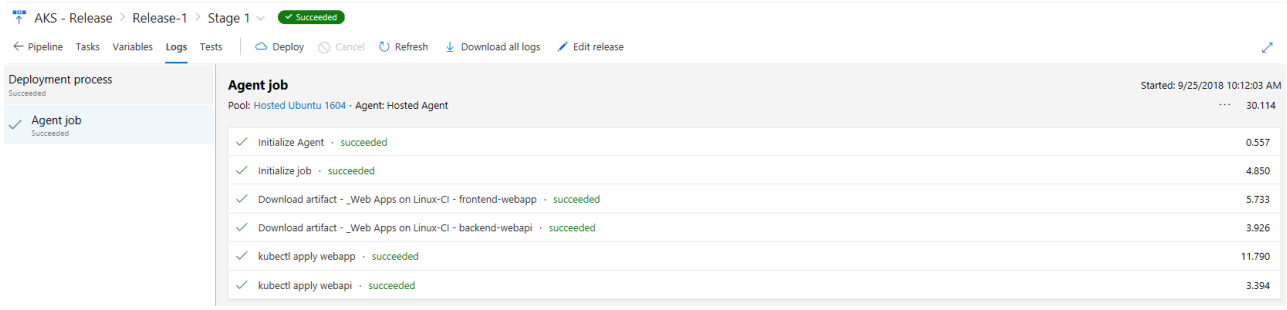
1. Go back to the Release page by clicking on the name of the created release. Click **Deploy**.



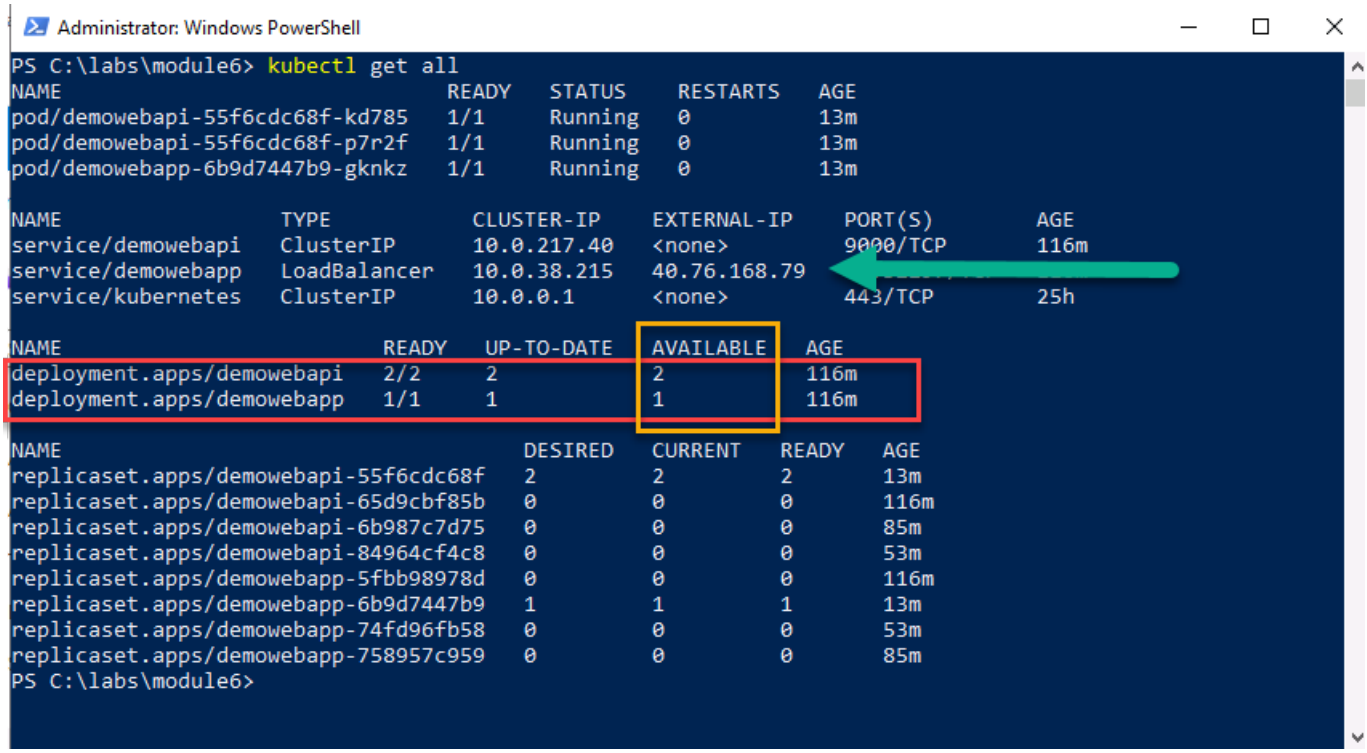
2. Finally, press **Deploy** to kick off the deployment to AKS cluster.



When it succeeds you will see a similar log as below:

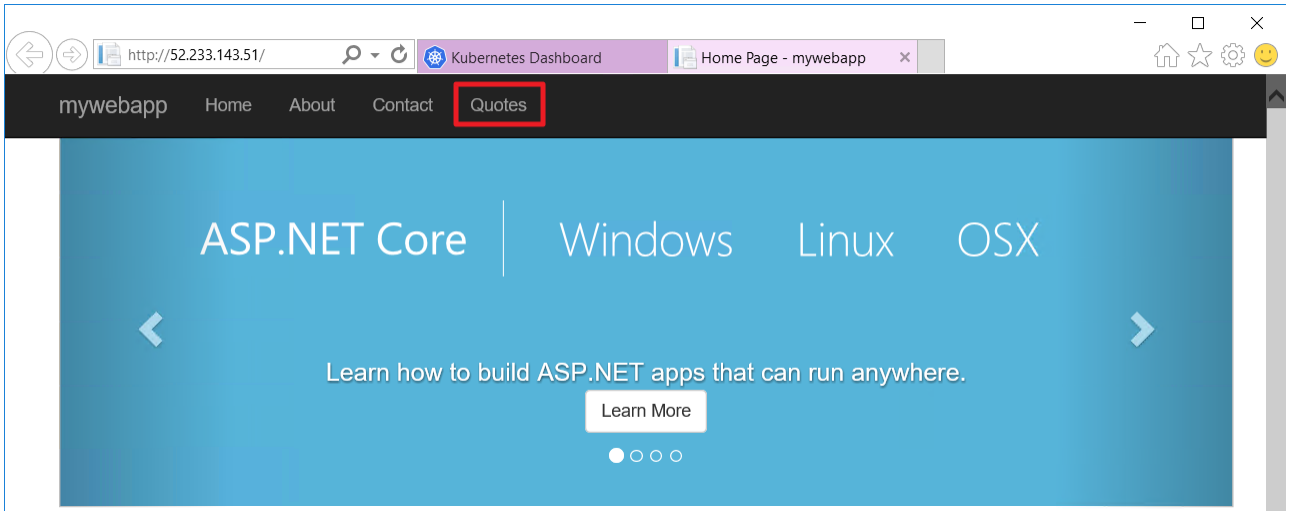


3. Now your deployment in Azure Pipelines is completed, but the deployment into the Kubernetes cluster is just beginning. In Powershell type `*kubectl get all*`. Wait until your two new deployements **deployments.apps/demowebapi** shows 2 available and **deployments.apps/demowebapp** shows 1 avaialble.

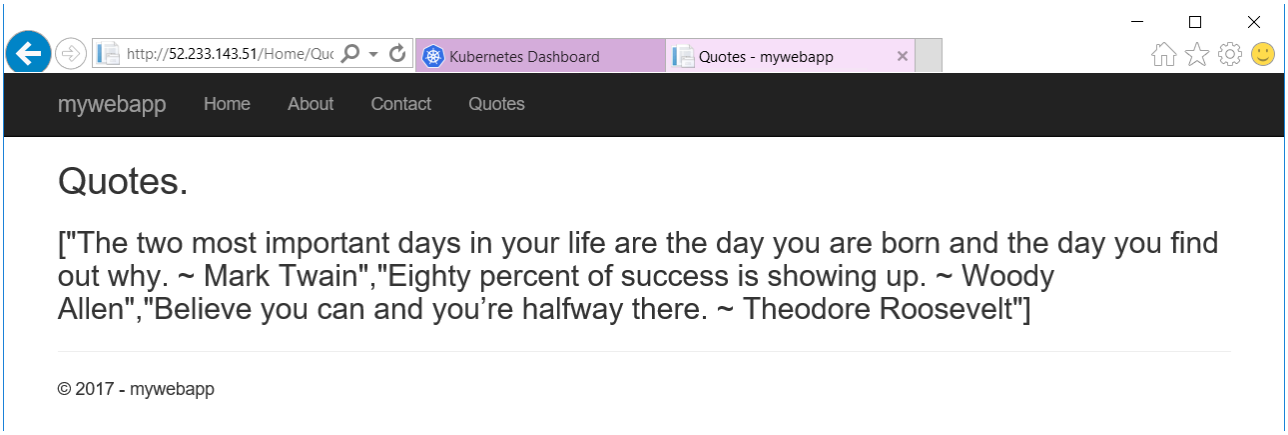


1. Open the browser using the **EXTERNAL-IP** for the **service/demowebapp** +++start http://$(kubectl get service demowebapp -o=jsonpath='{.status.loadBalancer.ingress[*].ip}')+++

2. When the web site is loaded, click on **Quotes** link.



> Note: Notice that frontend web application is successfully fetching quotes from the backend Web API. You can check **HomeController.cs** in the **webapp** project, and see that the frontend application is connecting to the backend Web API using the following URL http://demowebapi:9000/api/quotes. The URL's hostname is **demowebapi** is the service name of the backend application configured in the **backend-webapi.yaml** file.



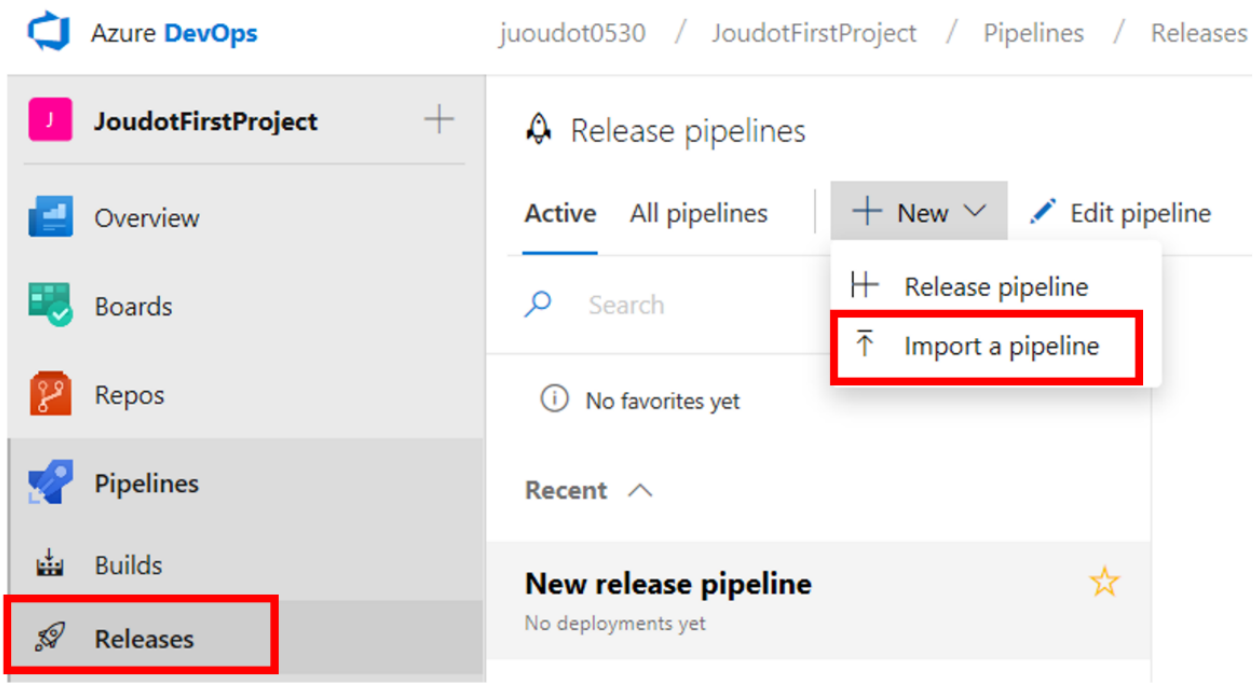
Optional: Import a release pipeline

> Note:Skip this optional task if you already created the release pipeline.

Return to list of exercises - Return to list of modules

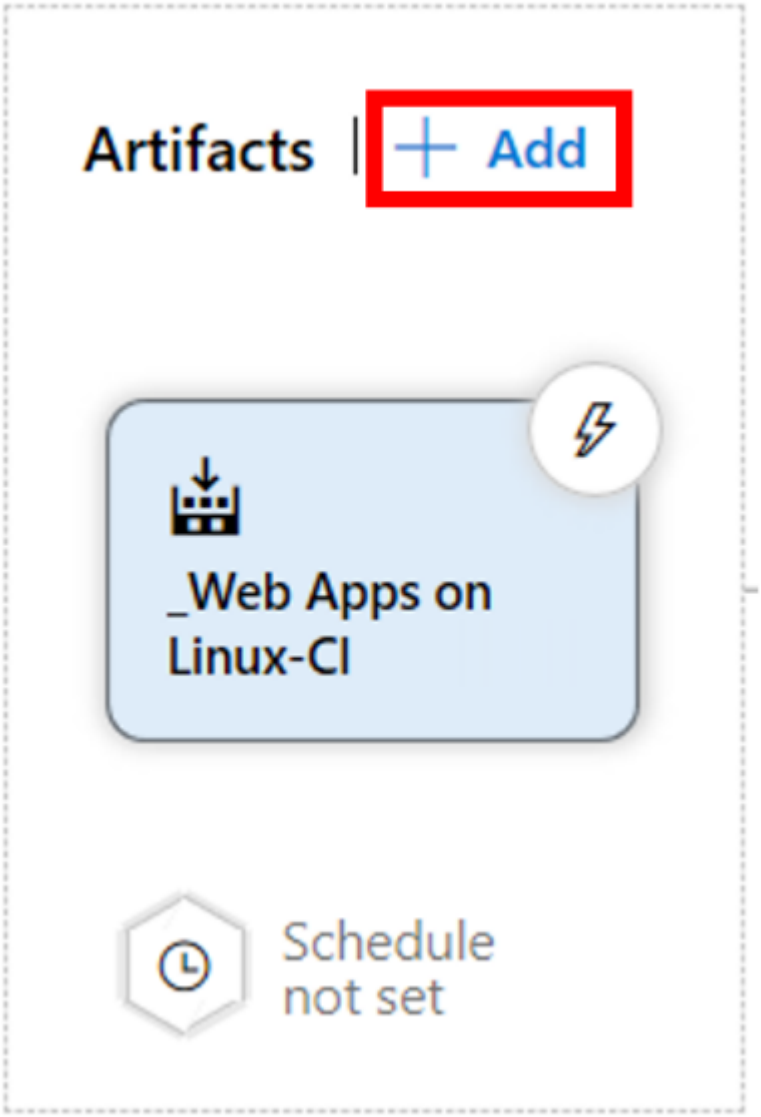
1. Navigate to **Pipelines** - **Releases** and click **New** - **Import a pipeline**.

   > Note: It seems that we can only do that when at least one pipeline exists. So you will need to create an empty pipeline first.

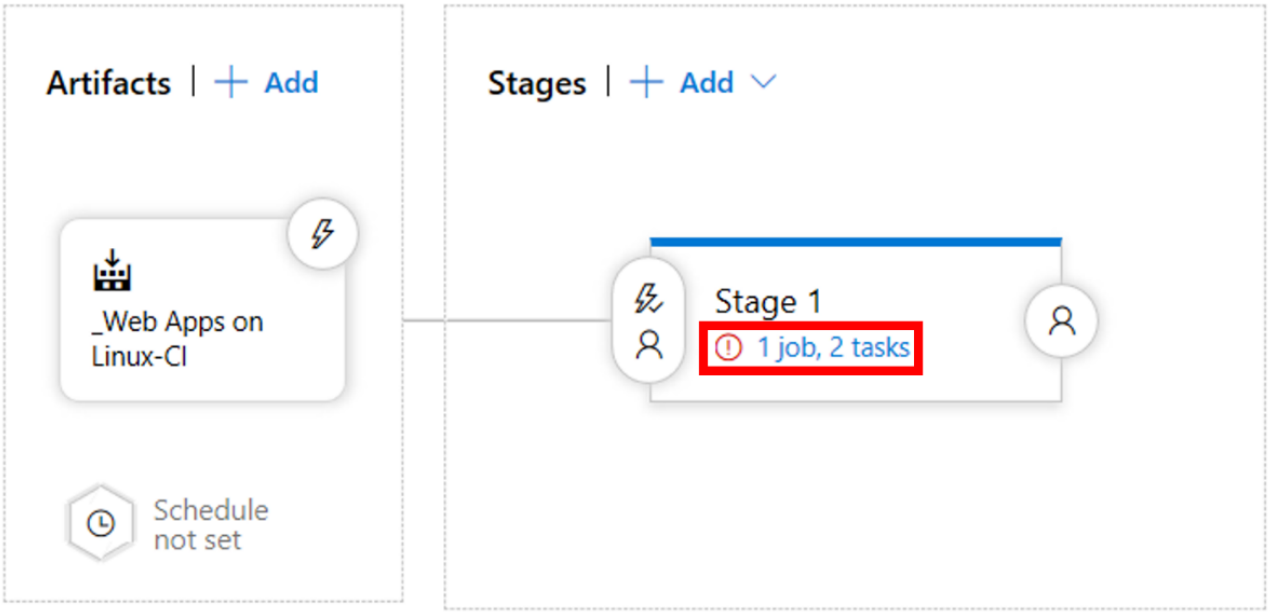


2. Select **C:\labs\module6-ext\AKS - Release-CD.json** and click **Import**

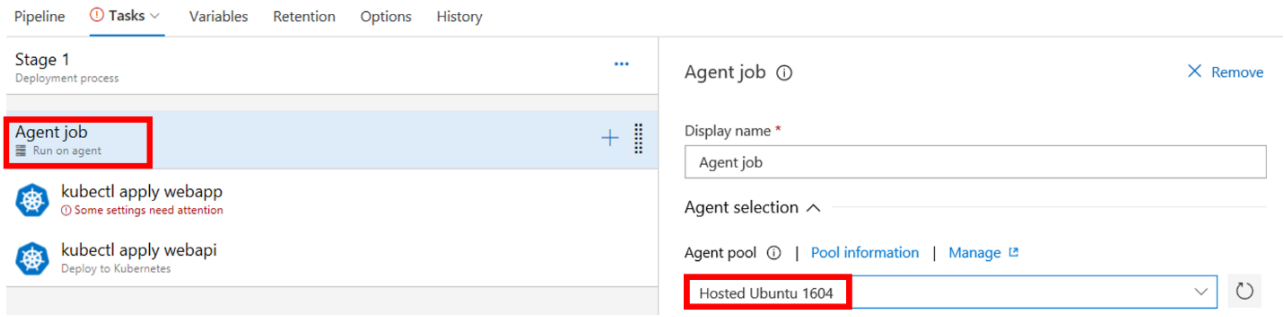
3. Make sure the input Artifact is valid by deleting the default one: click on the Artifact Name and then **Delete**. Finally, click on **Add an Artifact** to add the artifact from your new build.



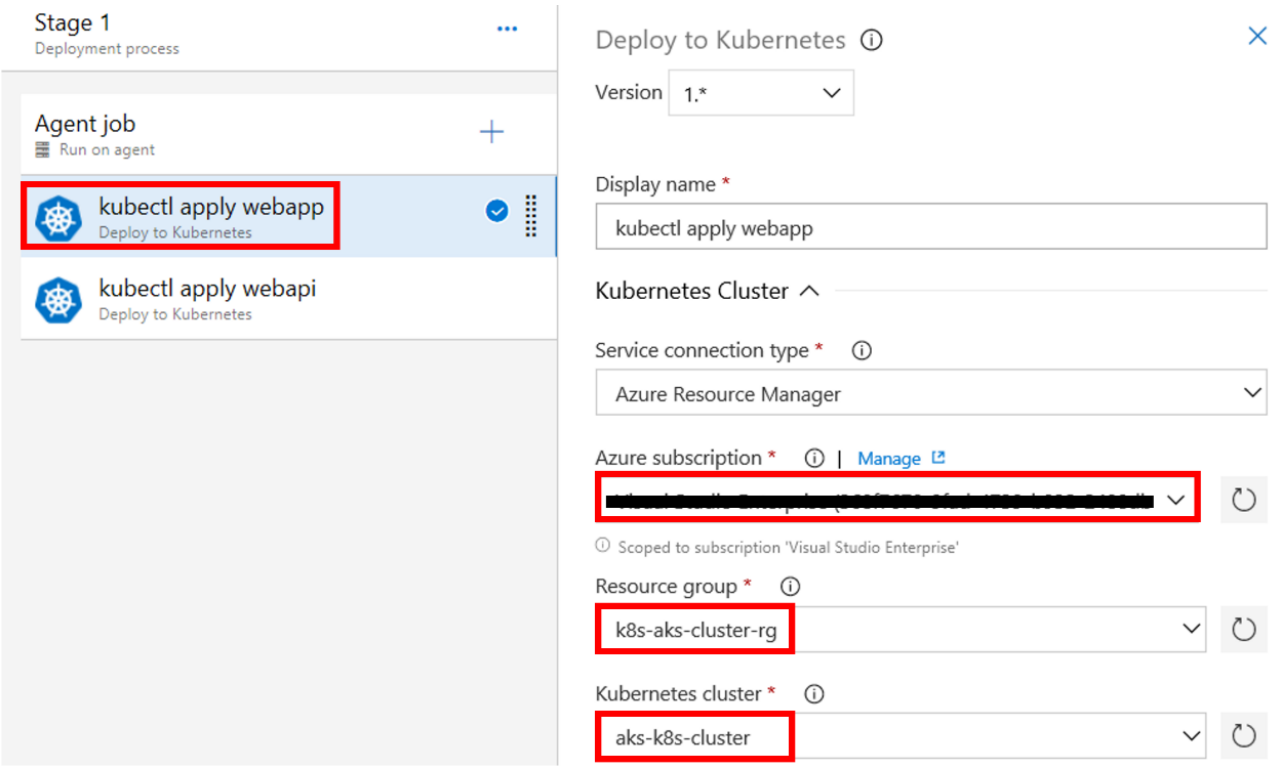
4. Click on the list of tasks under **Stage 1**

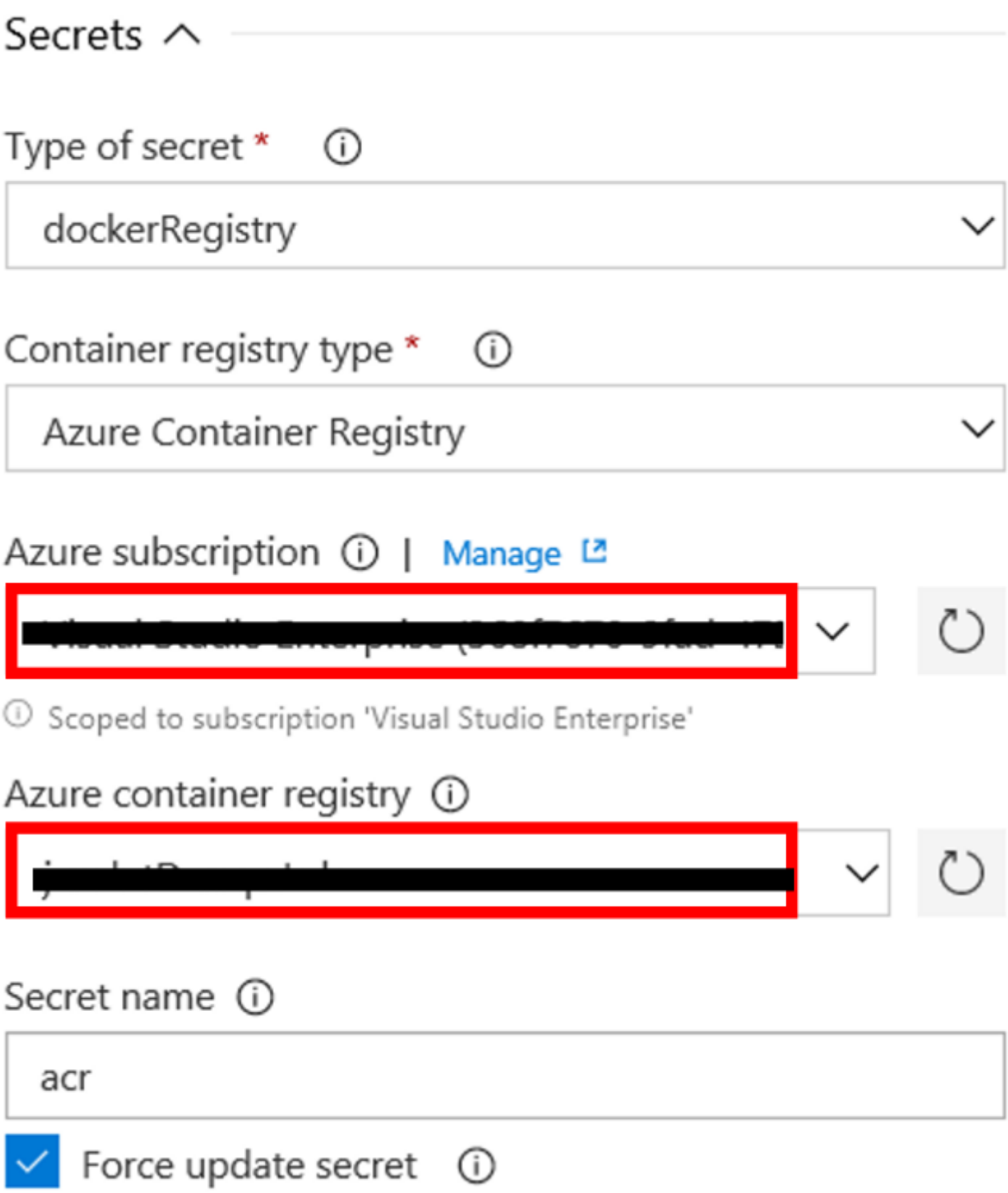


5. Under agent job, select **Hosted Ubuntu 1604** as an agent pool



6. Select the Kubernetes tasks one by one and enter the subscription, resource group and cluster information. Also enter the ACR information under **Secrets**





7. Once all the Kubernetes tasks are updated, **"Some settings need attention"** should disappear and you can save the pipeline. Go back to Step 12 in the previous task to start a release and deploy the application to your kubernetes cluster. Then, you will check status of the deployment using the kubernetes dashboard and subsequently test the application.

See what's new!

During Microsoft Build 2019, Kubernetes Integration for Azure Pipelines was announced. Take some time to read up on this update to see how Kubernetes deployments with Azure DevOps are getting even easier! https://devblogs.microsoft.com/devops/announcing-kubernetes-integration-for-azure-pipelines/

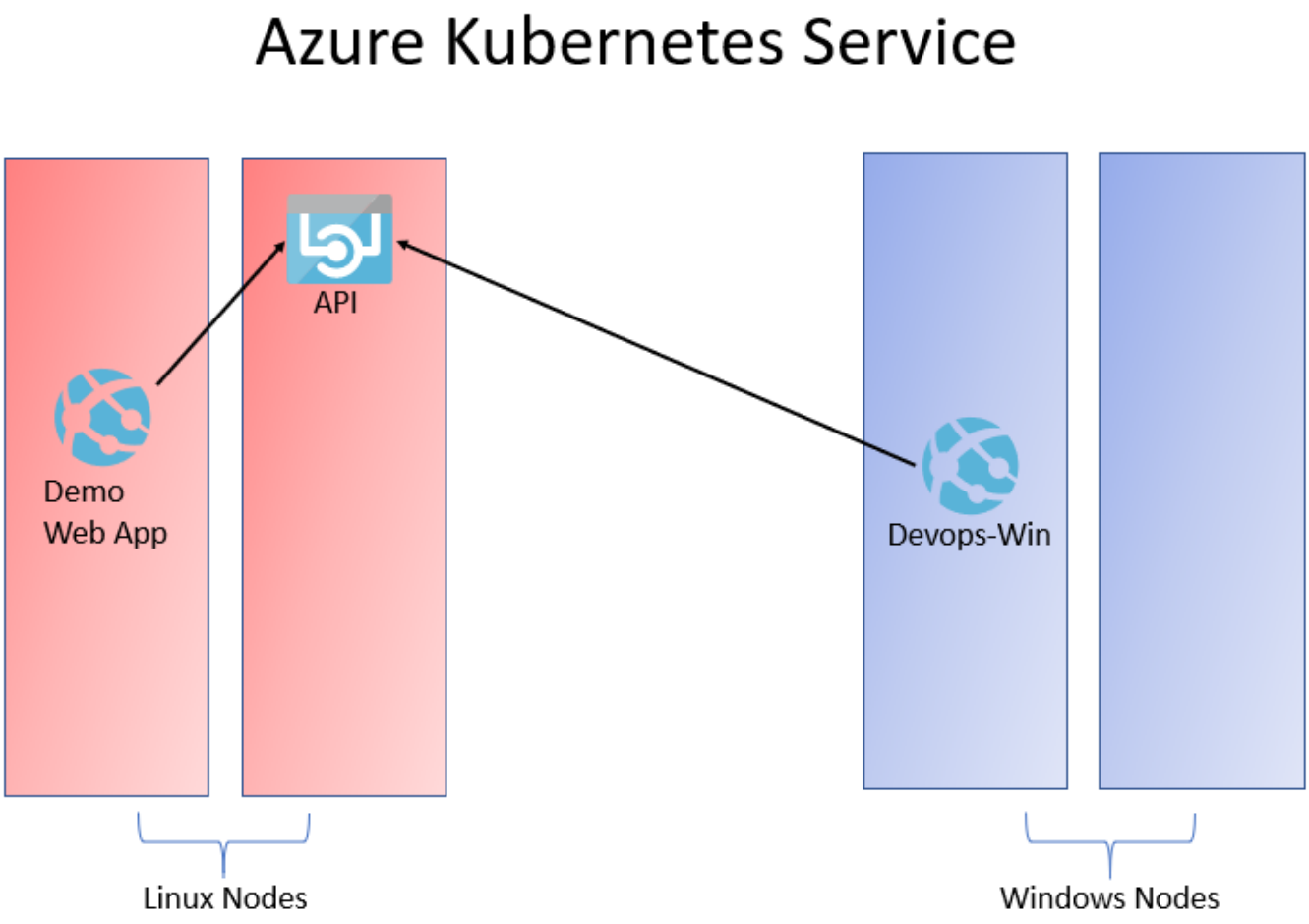
Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 5: Create Build Pipeline for Windows Containers in AKS

In this task, you are going to create a build pipeline which will be executed on Windows agent and will produce one Windows container image for the Web App front-end. The web front-end is an ASP.NET 4.7.x MVC Application hosted in an IIS Windows container and makes a request to the Quotes Linux API Service deployed in the previous exercise. This image will be tagged automatically with the appropriate build numbers and then pushed to the Azure Container Registry (ACR). Instead of creating the pipeline from scratch, we will import a JSON definition of the pipeline.

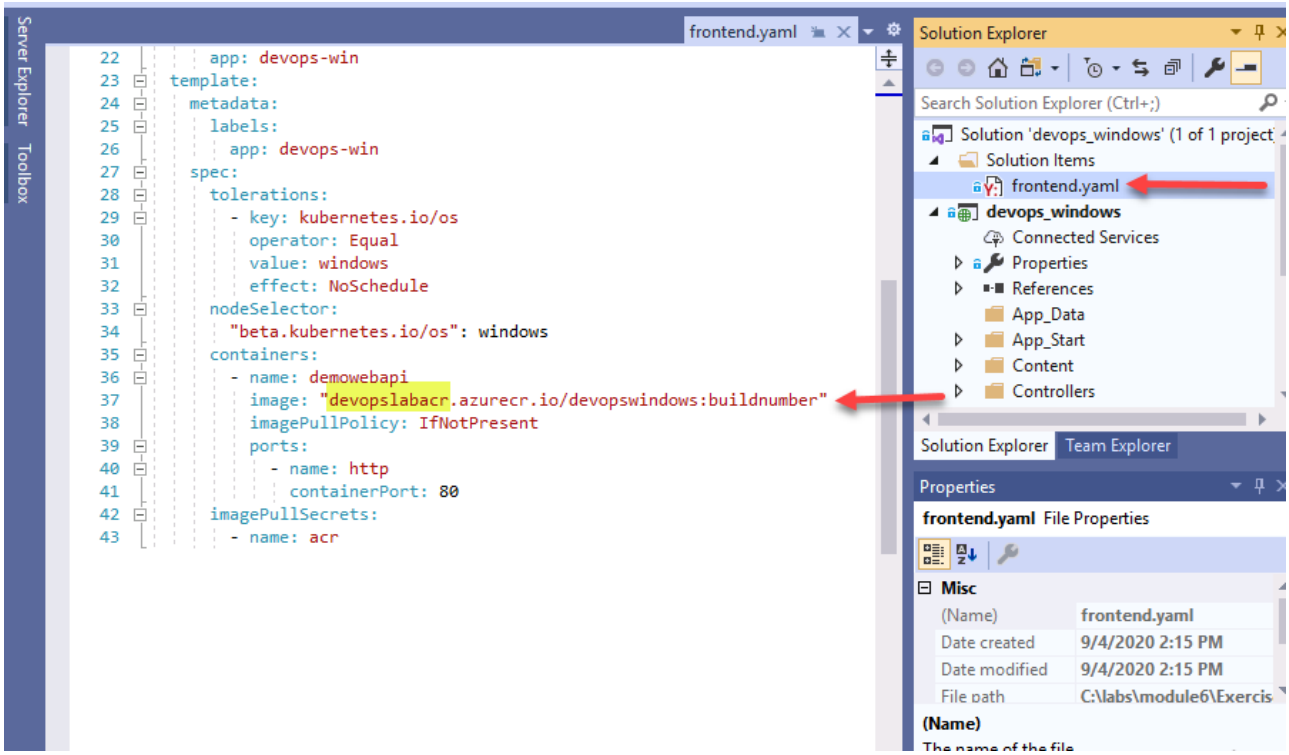
Return to list of exercises - Return to list of modules



Update Project

1. Navigate to +++cd c:\labs\module6\Exercise2+++ and open the **devops_windows.sln** solution in Visual Studio 2019 or greater.

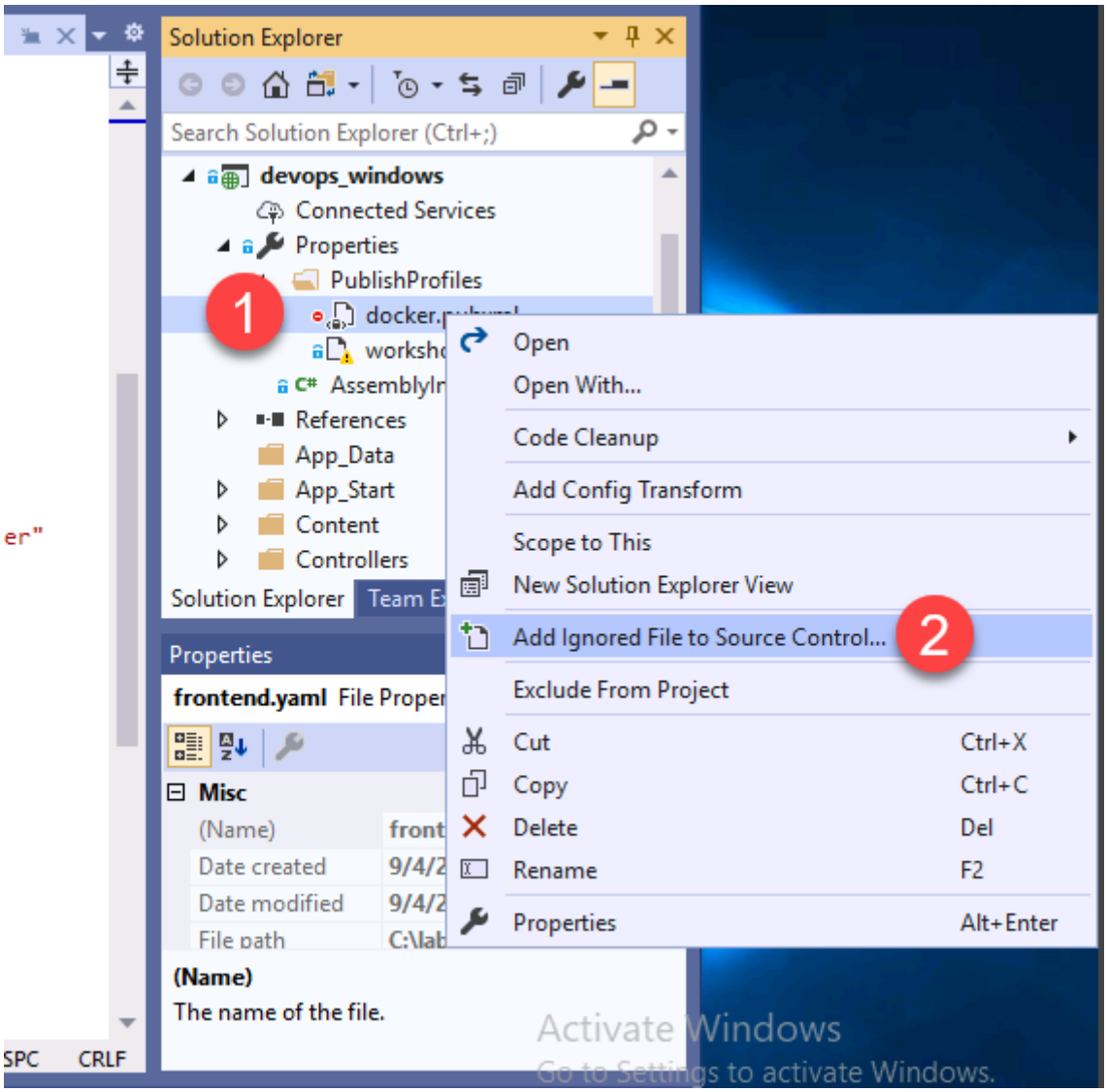
2. In Visual Studio, open **Solution Items -> frontend-yaml** file



3. Replace the default Image Name **devopslabacr** with the name of your Azure Container Registry.

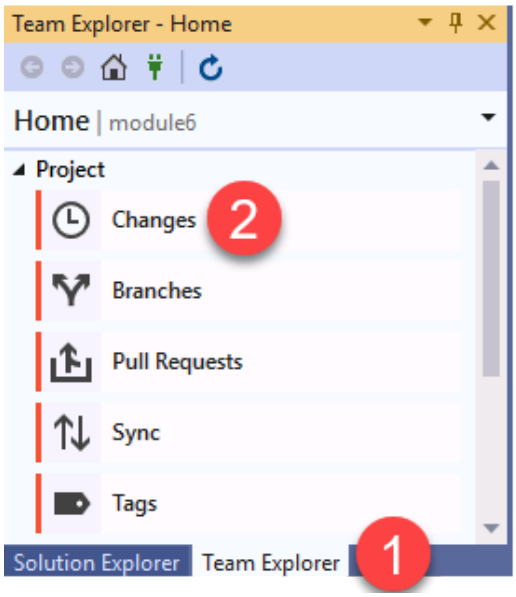
> Knowledge: Updating the Kubernetes YAML file ensures that, when deployed, Kubernetes will pull the image from the correct repository

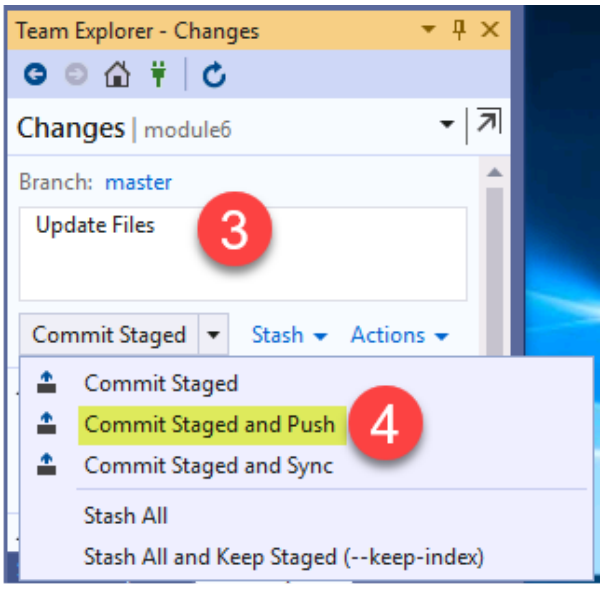
1. Next, in the Solition Explorer, **devops_windows** project, navigate to Properties then PublishProfiles. Right-Click on **docker.pubxml** and choose "**Add Ignored File to Source Control**"



> Knowledge: The Docker publish profile is used by the Build Pipeline to ensure that the IIS Website is correctly prepared for the Docker Image.

1. Finally, Click on **Team Explorer** then **Changes**. Next, add a Comment then click "Commit Staaged and Push".

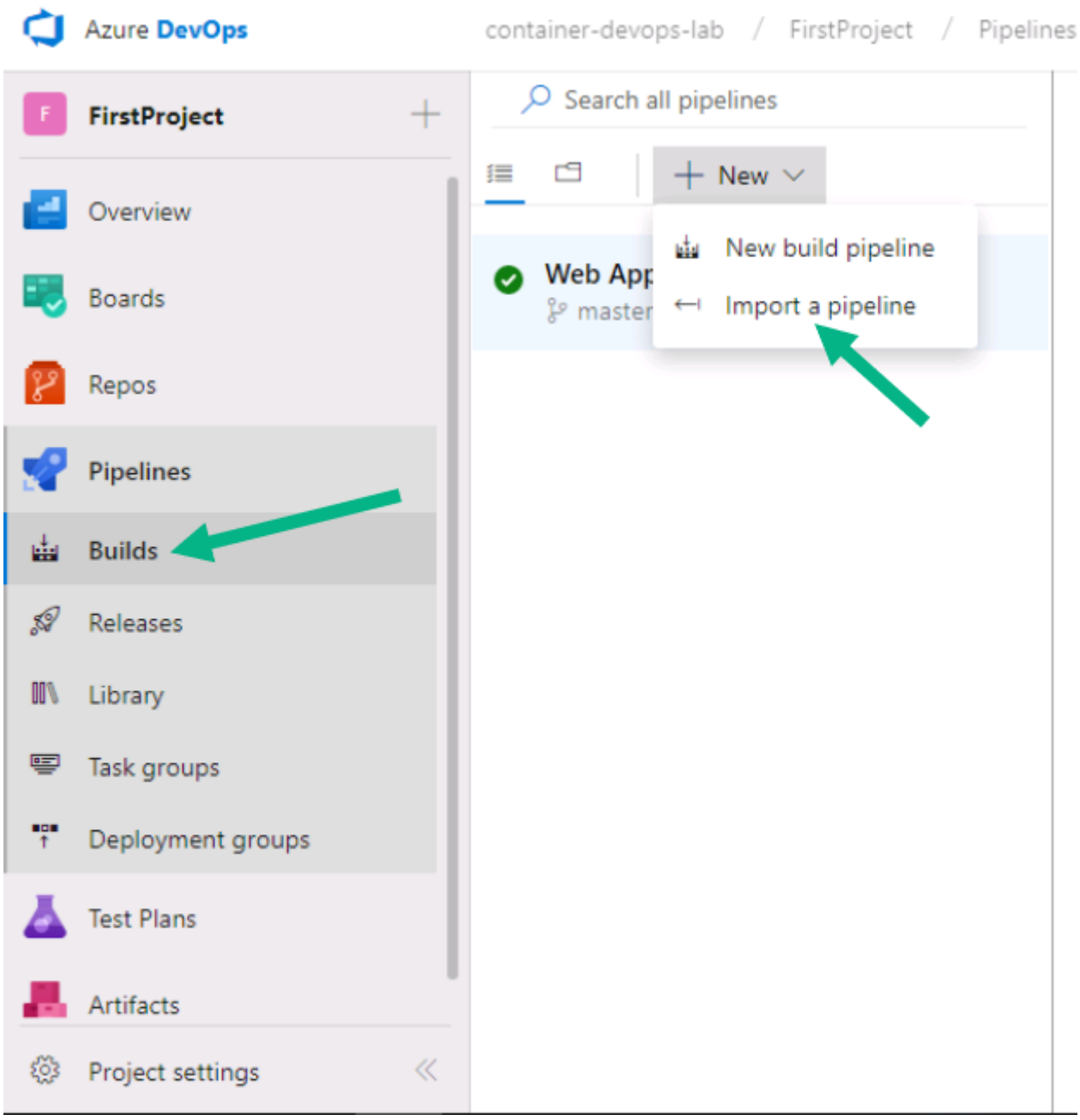




> Alert: If you get an error when trying to push your changes you may need to first do a **pull** to update your local files then **push** again

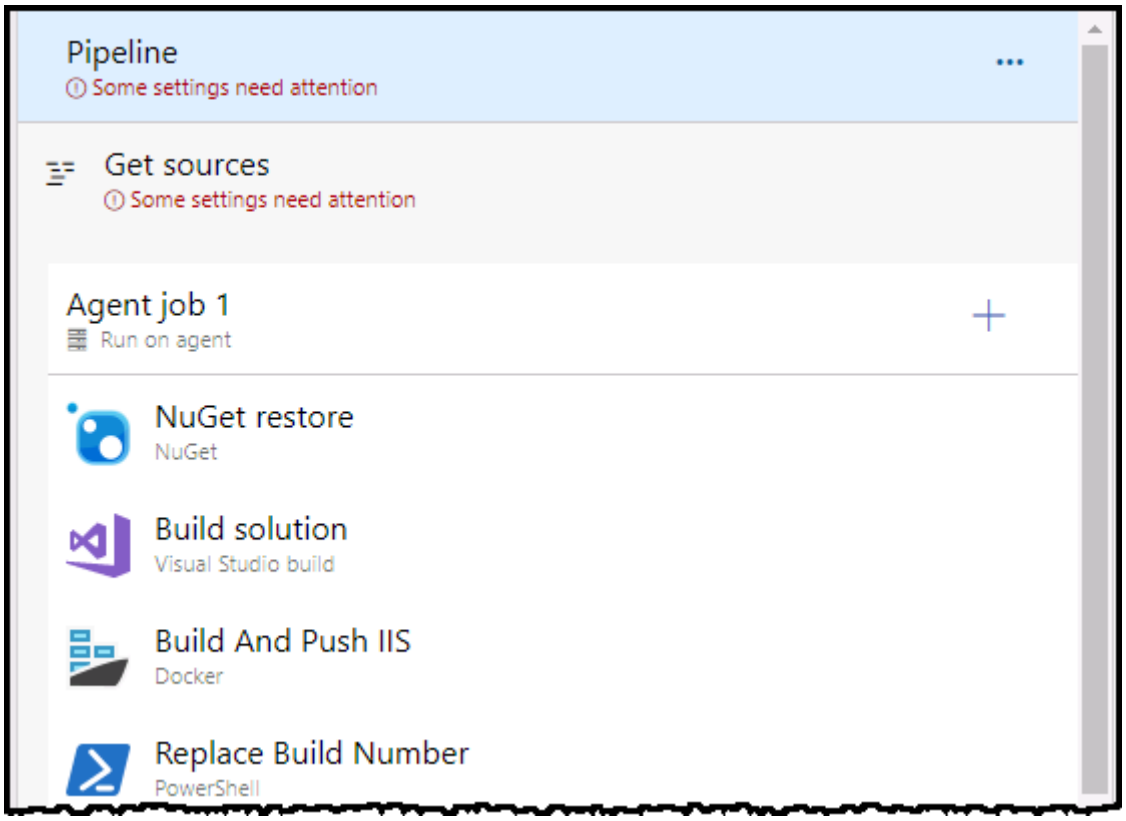
Import a build pipeline

1. Navigate to **Pipelines** - **Builds** and click **New** - **Import a pipeline**.



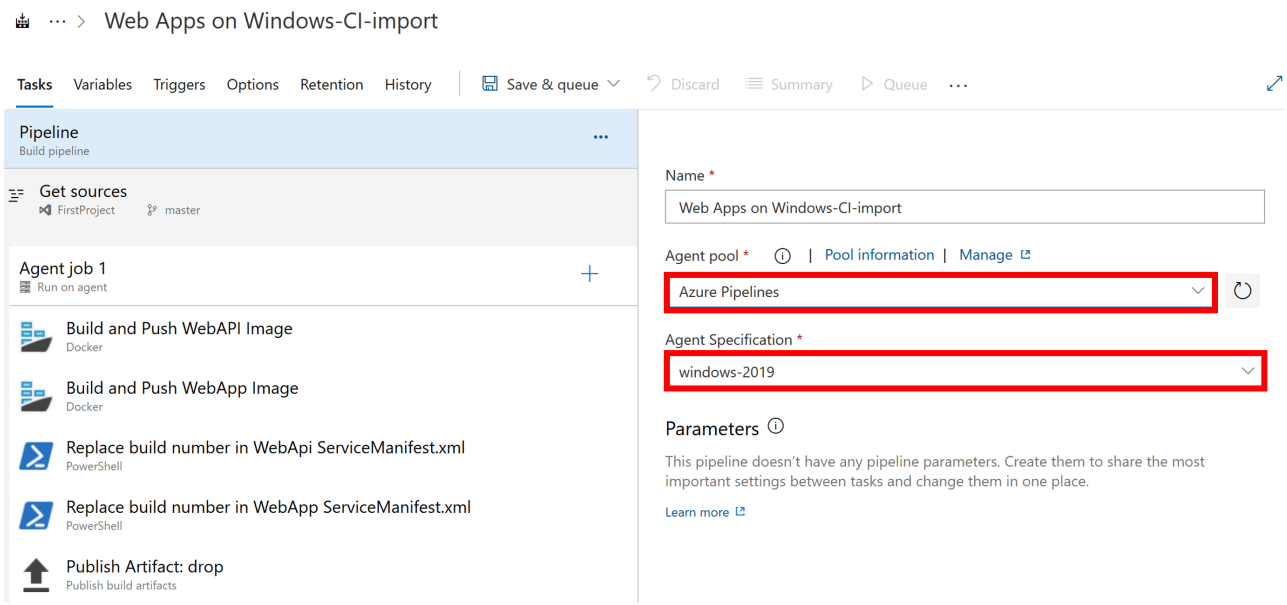
2. Select **C:\labs\module6-ext\Web Apps on Windows-CI.json** and click **Import**

3. You will see that some settings need to be updated. Click on **Pipeline**

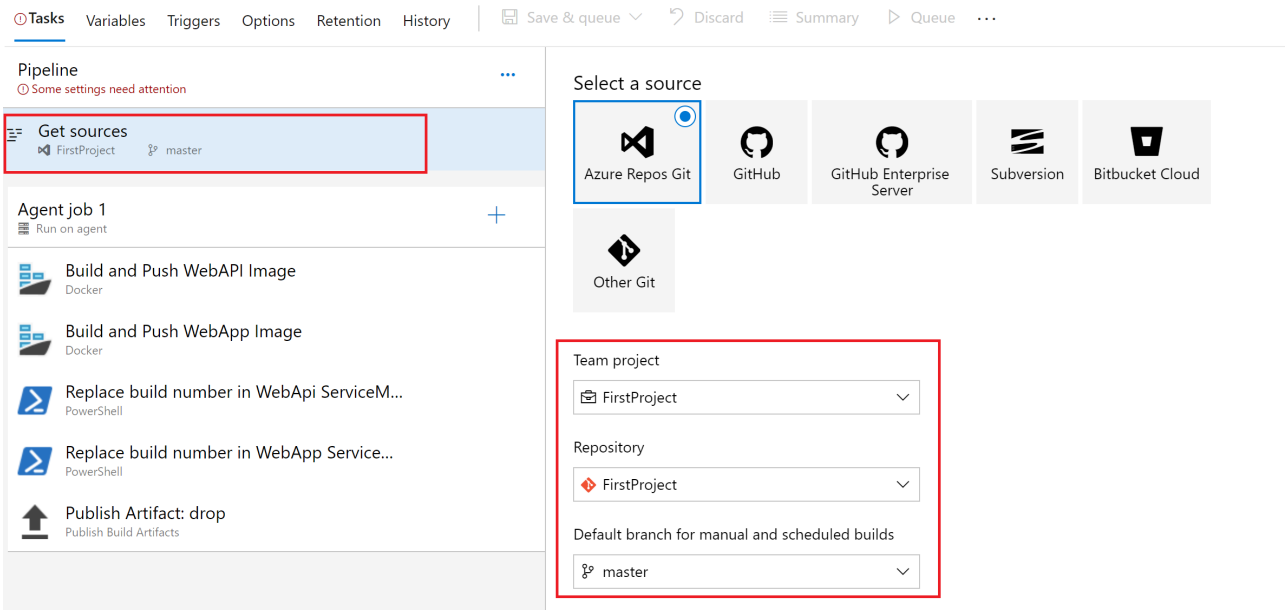


4. Select **Azure Pipelines** - **windows-2019** as an agent pool and specification. Keep confiuration as Release.

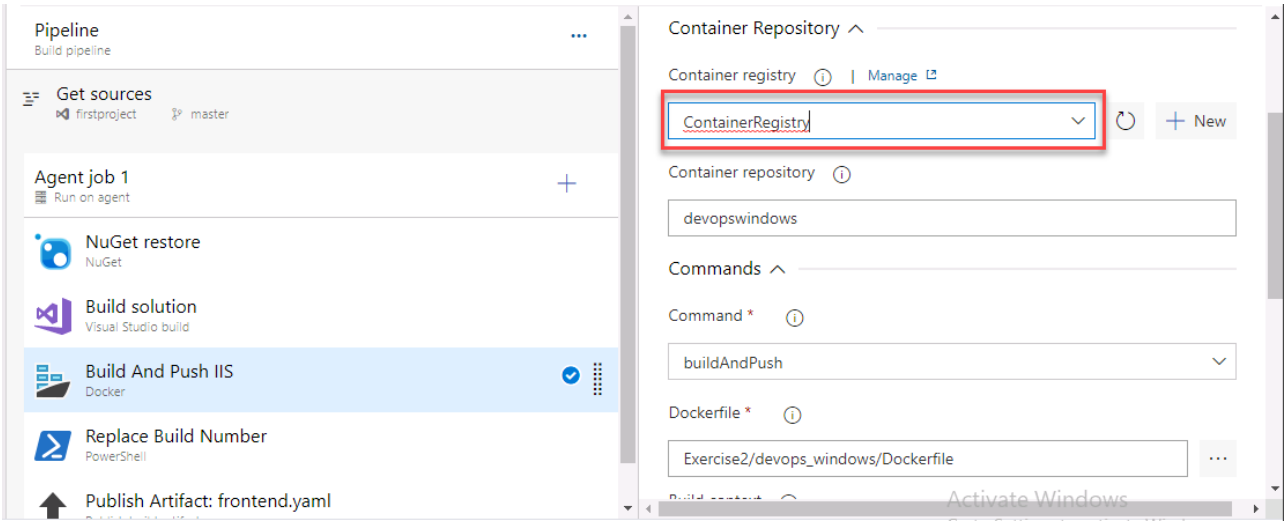
> Note: This is a Windows Server 2019 based agent, hosted by microsoft. This is important since we are about to build Windows Server 2019 container images.



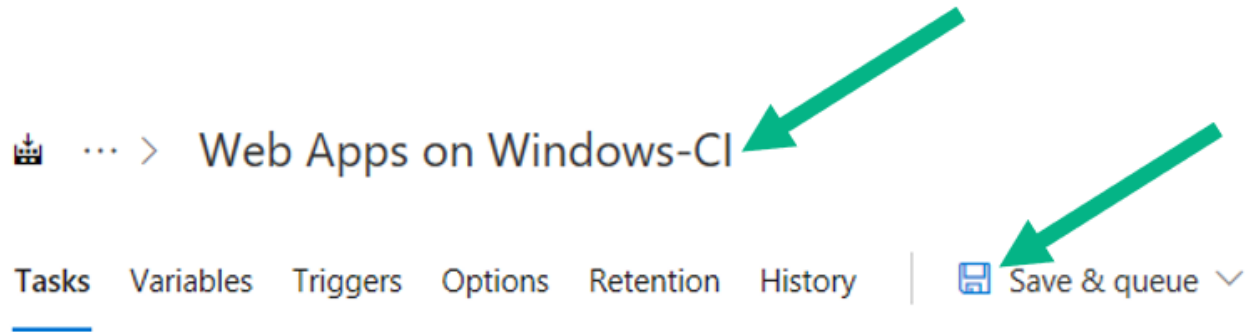
5. Click on **Get sources** and make sure that correct Git repository and branch are selected.



6. Select **Build and Push IIS** task and choose the container registry connection created in Exercise 1 (ContainerRegistry).

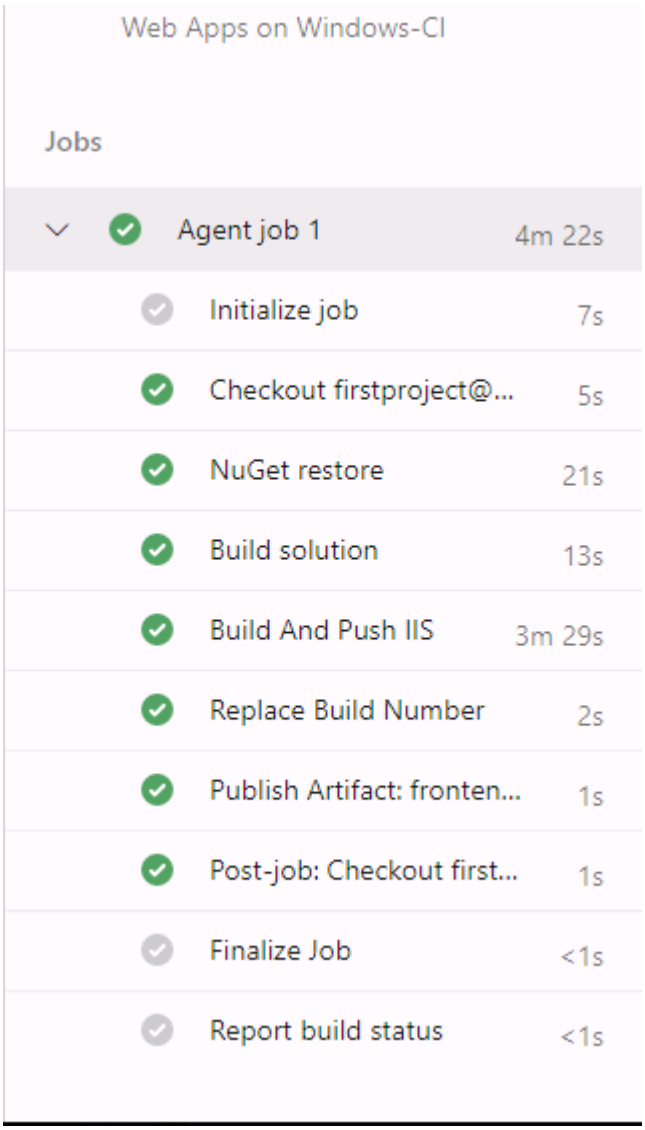


7. Make sure the Build pipeline is named **Web Apps on Windows-CI** to stay aligned with the following screenshots. Then click on **Save & queue** to start the build.



8. Your finished build should look like this

> Note:If you get an error please click on the step with the error and try debugging or the instructor will come by and help you



Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

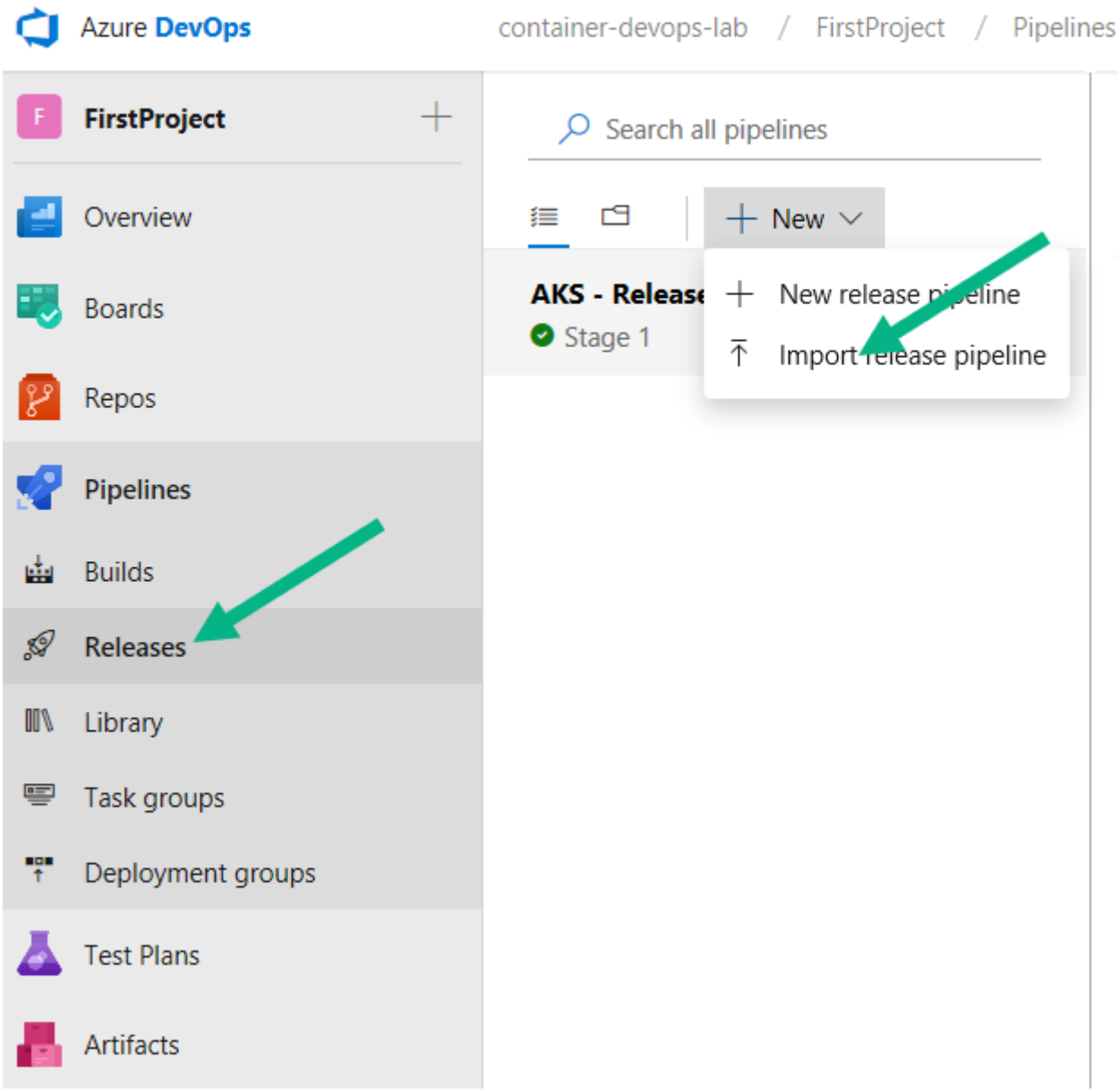
Exercise 6: Create Release Pipeline for AKS Windows Nodes

In this exercise, you will create a Release Pipeline which will pull container images from ACR (result of a previous build task), and then deploy these containers to Windows Nodes in an AKS Cluster.

The build pipeline is used for compiling the application, building a container image, and then pushing container images to the container registry. A release pipeline, on the other hand, is used to pull the container image from the registry and deploy the image as a container to the cluster. Instead of creating the pipeline from scratch, we will import the JSON definition for the pipeline.

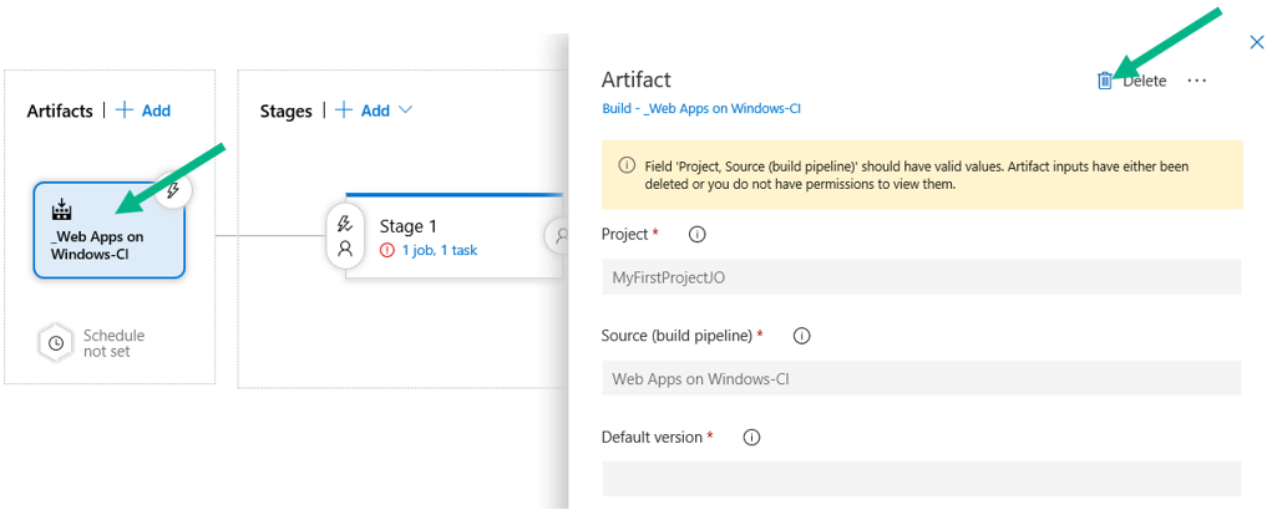
Return to list of exercises - Return to list of modules

1. Navigate to **Pipelines** - **Releases** and click **New** - **Import a pipeline**.

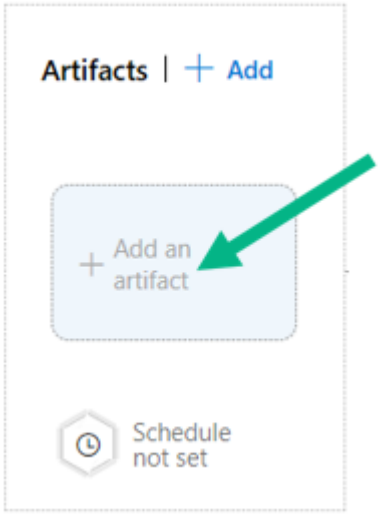


2. Select **C:\labs\module6-ext\AKS-Win Release.json** and click **Import**

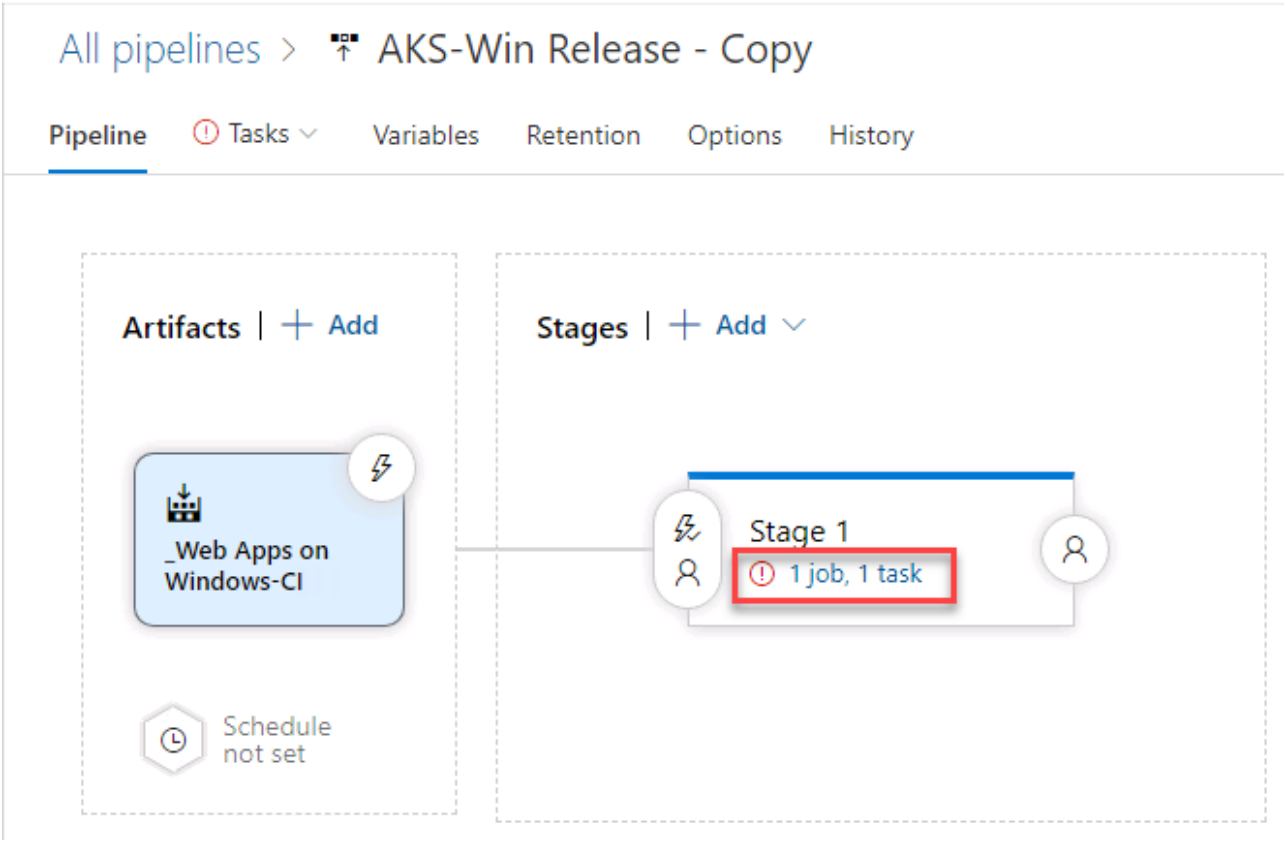
3. Make sure that the input Artifact is valid by deleting the default one: click on the Artifact Name and then **Delete**.



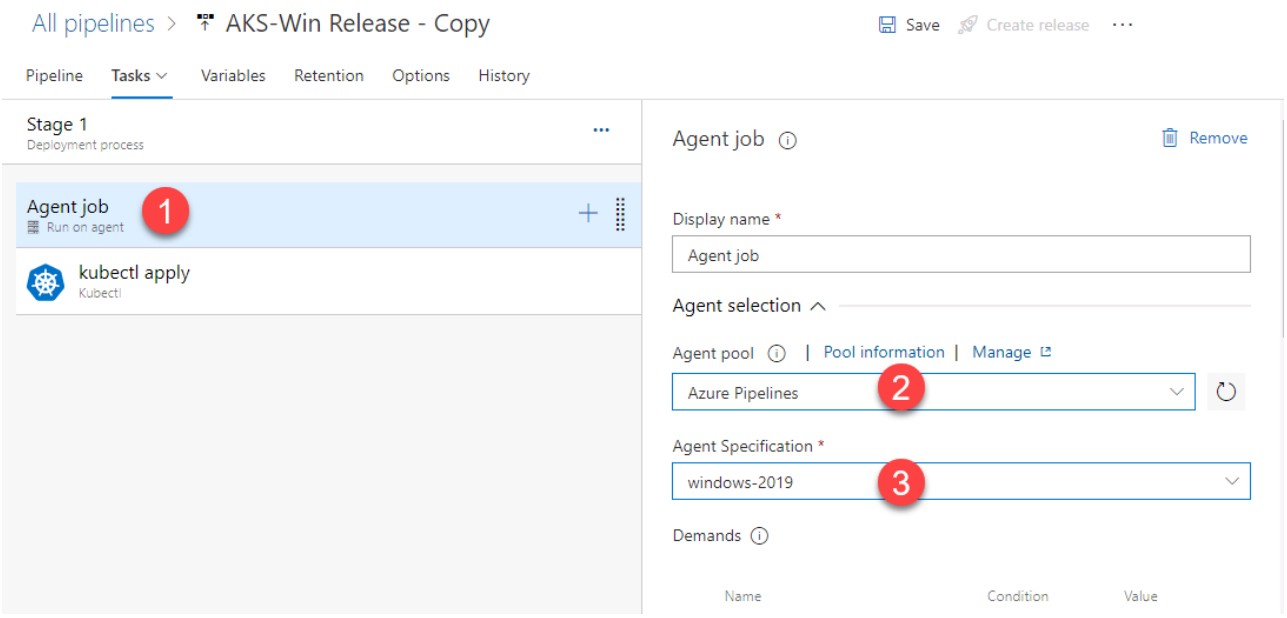
4. Finally, click on **Add an Artifact** to add the artifact coming from your **Web Apps on Windows-CI** build then click **Add**.



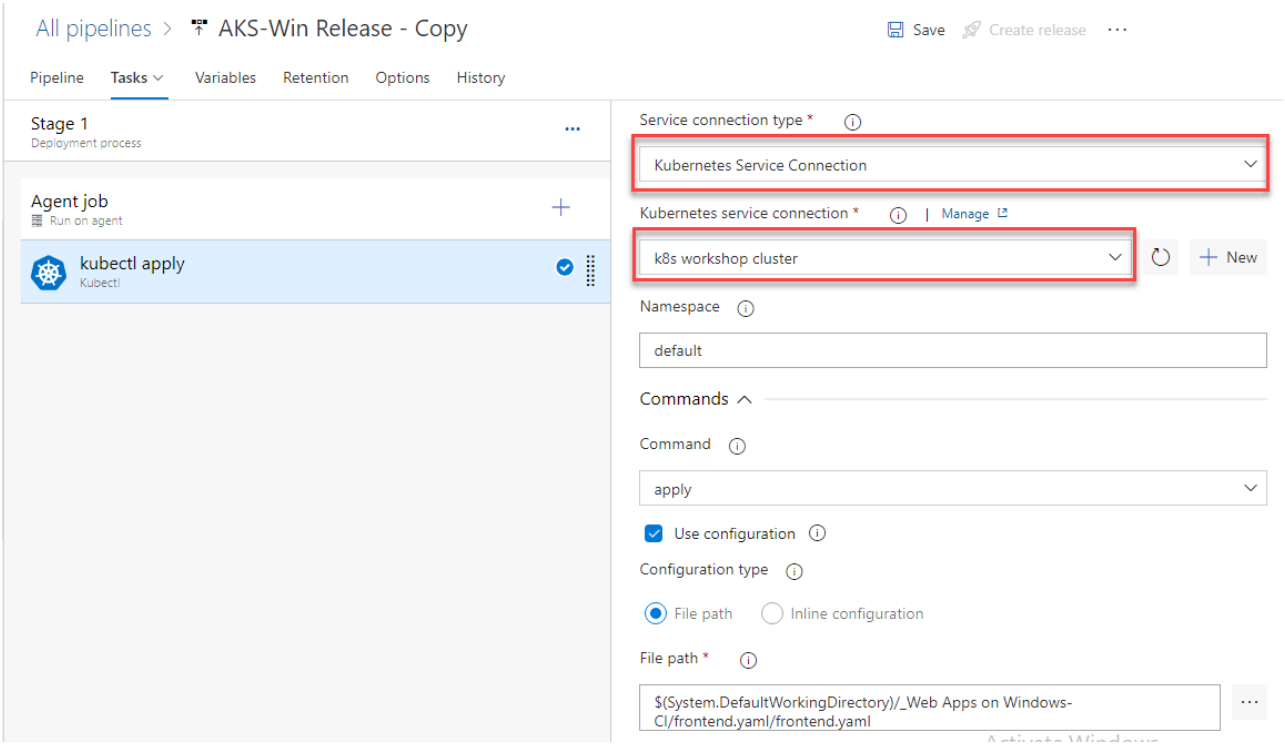
5. Update the deployment task by clicking on the tasks link under **Stage 1**



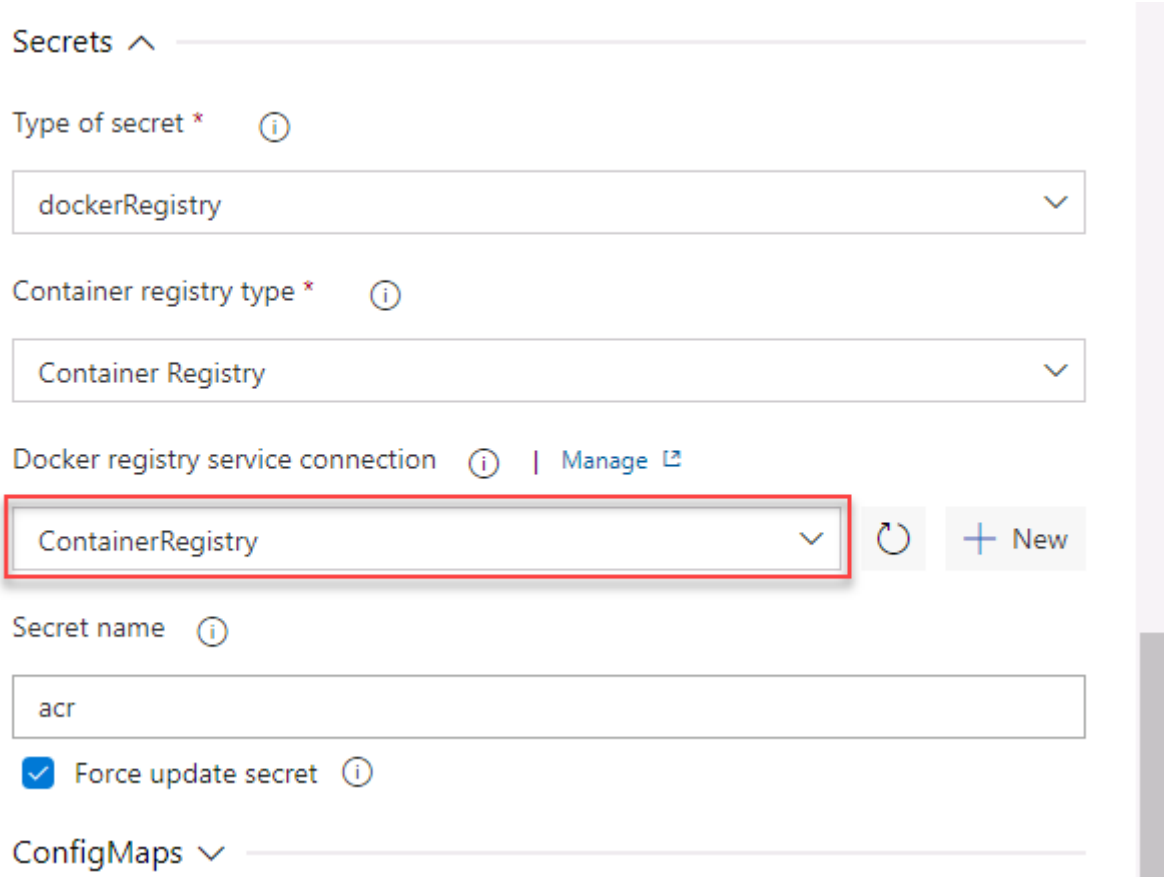
6. Click on **Agent Job** and set the Agent Pool and set the **Agent Pool** to Azure Pipelines and **Agent Specification** to Windows-2019. You must use a Windows Agent to build Windows images.



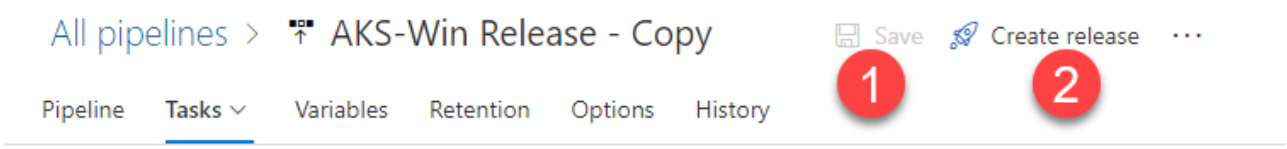
7. Next, click on the **kubectl apply** task and verify that the **Service Connection Type** and **Kubernetes Service Connection** are set correctly



8. Scoll down and expand the **Secrets** section then update the **Docker Registry service connection** with the previously created connection **ContainerRegistr**



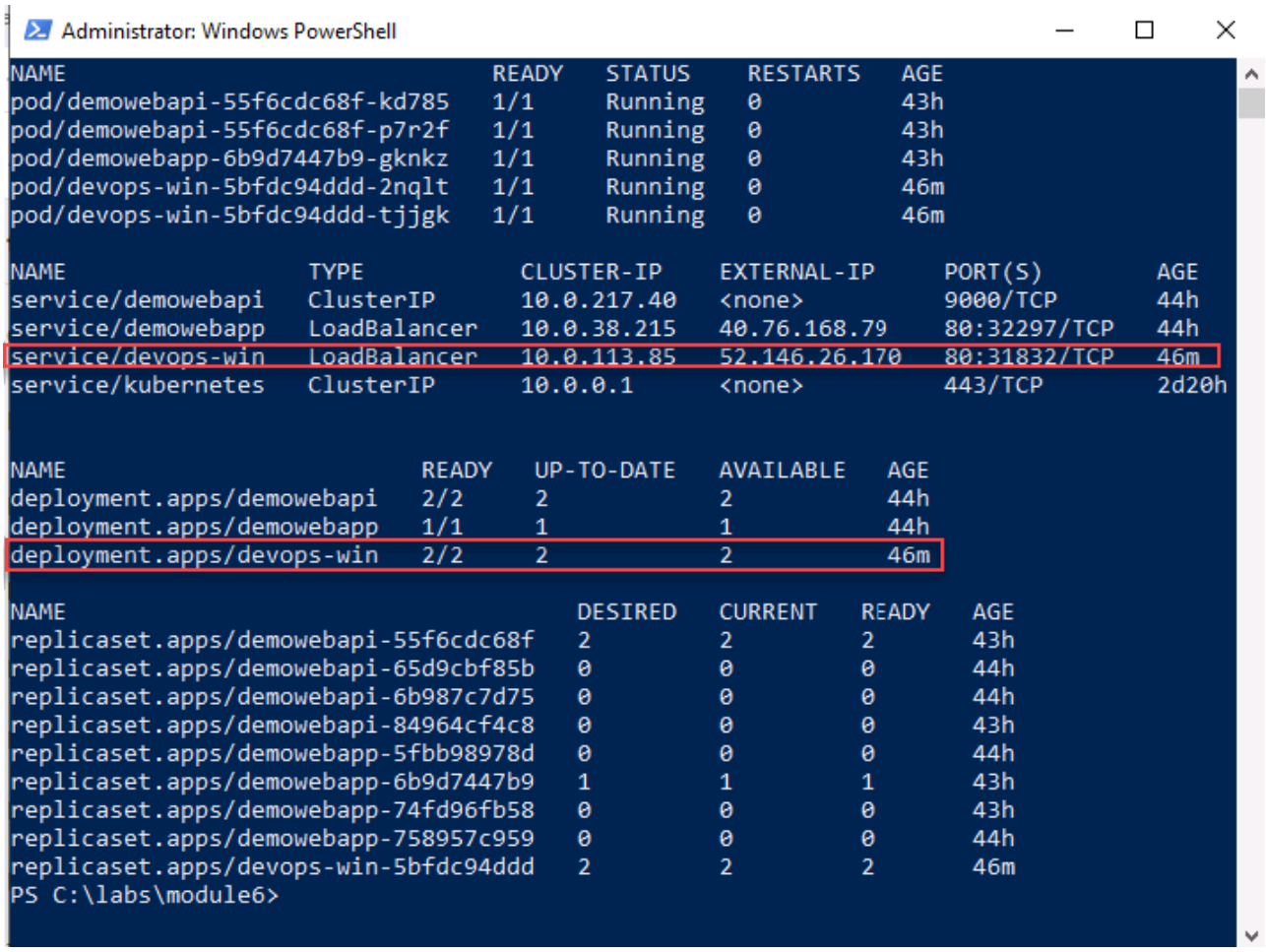
9. Finally, click **Save** Then **Create Release** then **Save** to deploy your Windows Container to AKS



10. Wait for the Release pipeline to successfully complete.

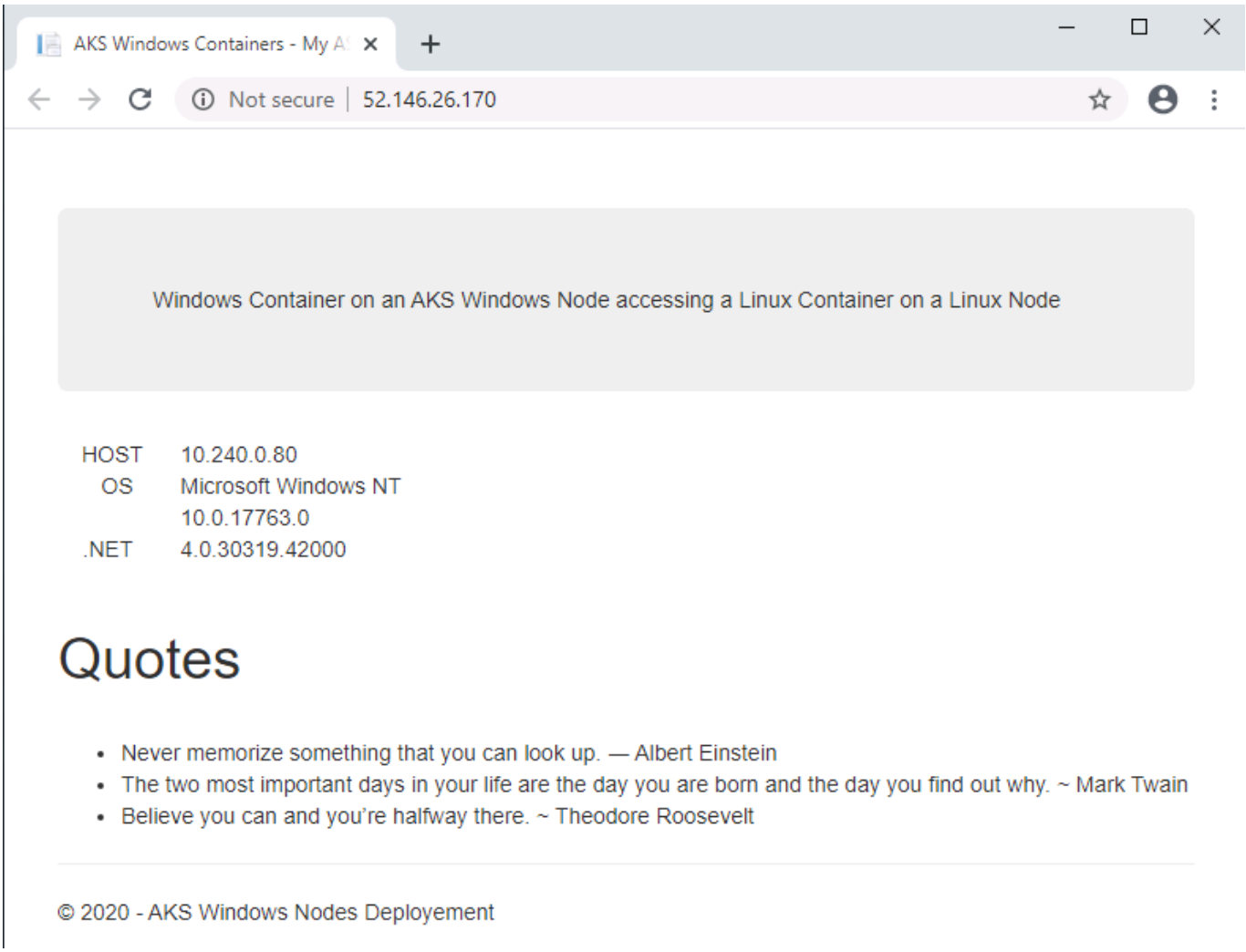
Verify Deployment

1. In Powershell, type `*kubectl get all*`
2. Wait for the **devops-win** deployment to show 2 pods available then browse to the EXTERNAL-IP address shown for the **devops-win** service



> Alert: It can take several minutes before the pods are ready

1. Browse to devops-win application `*start http://$( kubectl get service devops-win -o=jsonpath='{.status.loadBalancer.ingress[*].ip}')*`



Congratulations!

You have successfully completed this exercise.

Module 7 - Monitoring and Troubleshooting Containers

> Duration: 25 minutes

Module 7: Table of Contents

Exercise 1: Enable Azure Monitor for Containers in an Azure Kubernetes Service cluster

Azure Monitor for Containers is a feature designed to monitor the performance of container workloads deployed to either Azure Container Instances or managed Kubernetes clusters hosted on Azure Kubernetes Service (AKS).

Azure Monitor for containers gives you performance visibility by collecting memory and processor metrics from controllers, nodes, and containers that are available in Kubernetes through the Metrics API. Container logs are also collected.

After you enable monitoring from Kubernetes clusters, metrics and logs are automatically collected for you through a containerized version of the Log Analytics agent for Linux.

Metrics are written to the metrics store and log data is written to the logs store associated with your Log Analytics workspace.

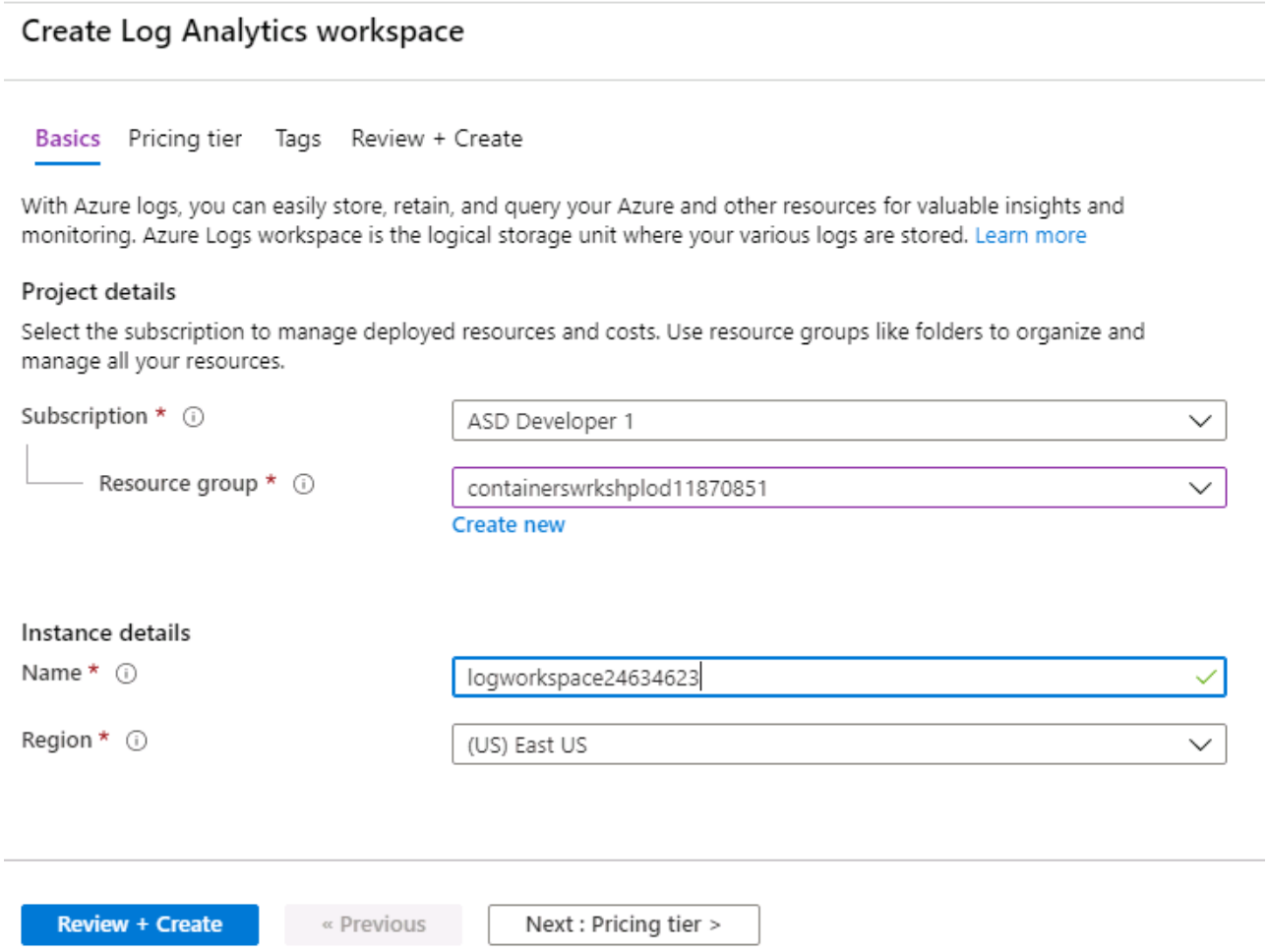
> Note: With the preview release of Windows Server support for AKS, a Linux node automatically deployed in the cluster as part of the standard deployment collects and forwards the data to Azure Monitor on behalf all Windows nodes in the cluster.***

Return to list of exercises - Return to list of modules

Create a Log Analytics Workspace

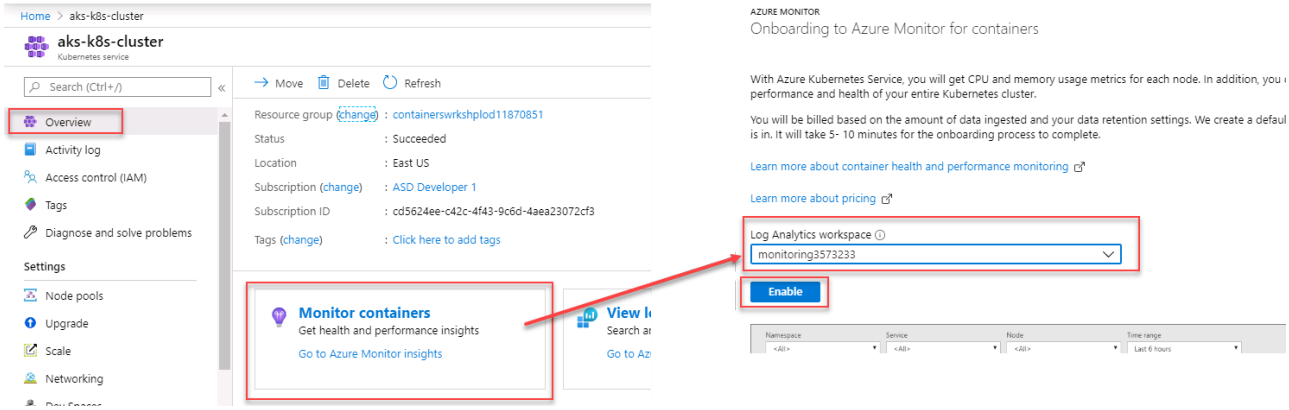
1. From the +++https://portal.azure.com+++ click on **Create a Resource** and search for **Log Ananytics Workspace**

2. Choose the Rosource Group used for the Labs and a unique Workspace Name then click on **Create**



3. Navigate to your AKS Cluster in the Azure Portal using the search bar

    1. On the cluster overiew page, select **Monitor Containers**
    2. Select the existing Log Analytics workspace created earlier and click **Enable**



4. Ensure you can access your AKS cluster through the Azure CLI by getting the credentials: `az aks get-credentials --resource-group <resource_group_name> --name aks-k8s-cluster`

    > Knowledge: use **kubectl config --help** to see how you can ensure your current context points to the correct AKS cluster. You can also use the displayed commands to delete contexts for AKS clusters that no longer exist, switch between contexts, etc.

5. Once you have verified your AKS cluster is configured as the current-context, verify the OMS agent was successfully deployed using the following command: `kubectl get ds omsagent --`

```
namespace=kube-system
```

6. Verify the OMS agent solution was also deployed successfully using the following command: `kubectl get deployment omsagent-rs -n=kube-system`

7. Use the `az aks show --resource-group <resource_group_name> --name aks-k8s-cluster --query addonProfiles.omsagent -o json` command to get details regarding the OMS Agent configuration

Congratulations!

You have successfully completed this exercise. Click **Next** to advance to the next exercise.

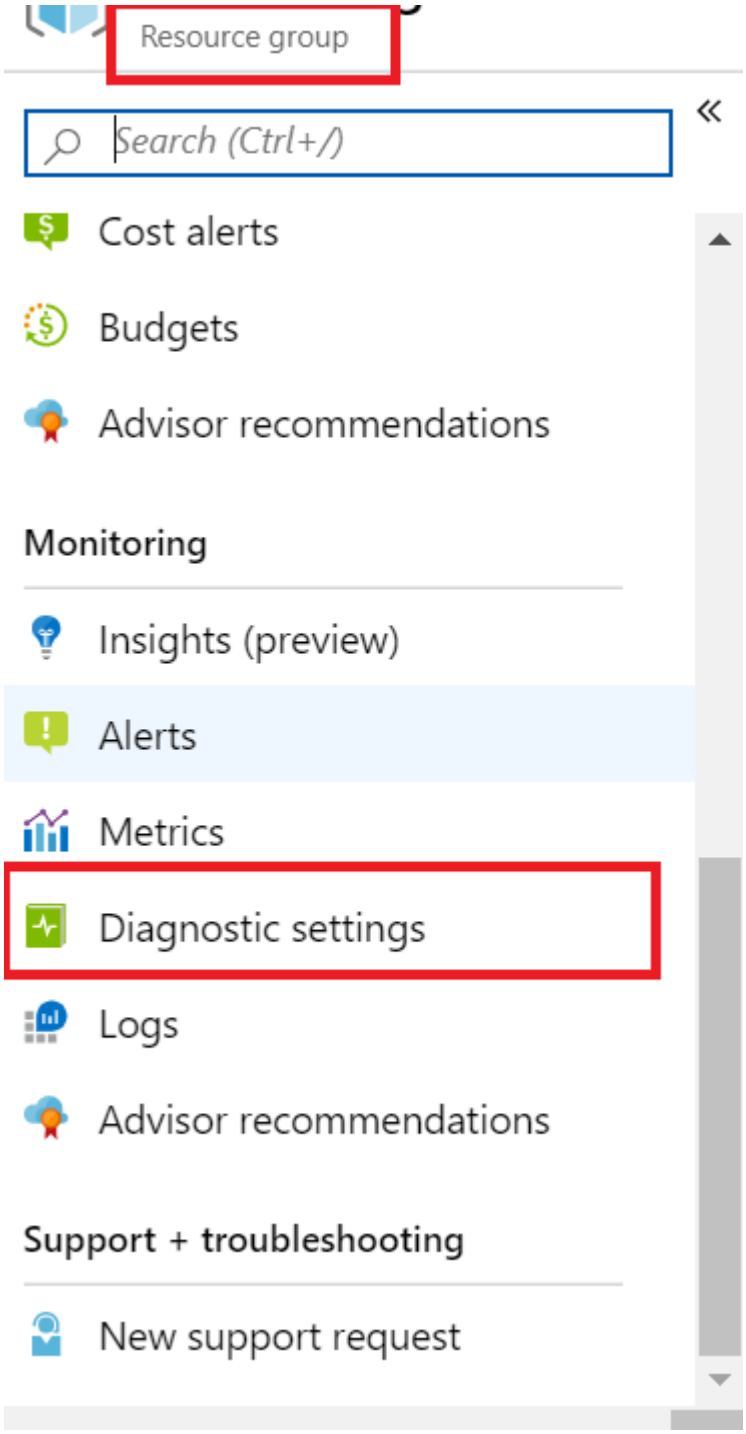
Exercise 2: Enable Master Node Logs in AKS

One of the first things you are going to want to do is enable the Master Node Logs to help with troubleshooting applications running in production. This will help you gain additional insights into potential issues.

Return to list of exercises - Return to list of modules

1. Navigate to the Azure Portal and select your Resource Group.

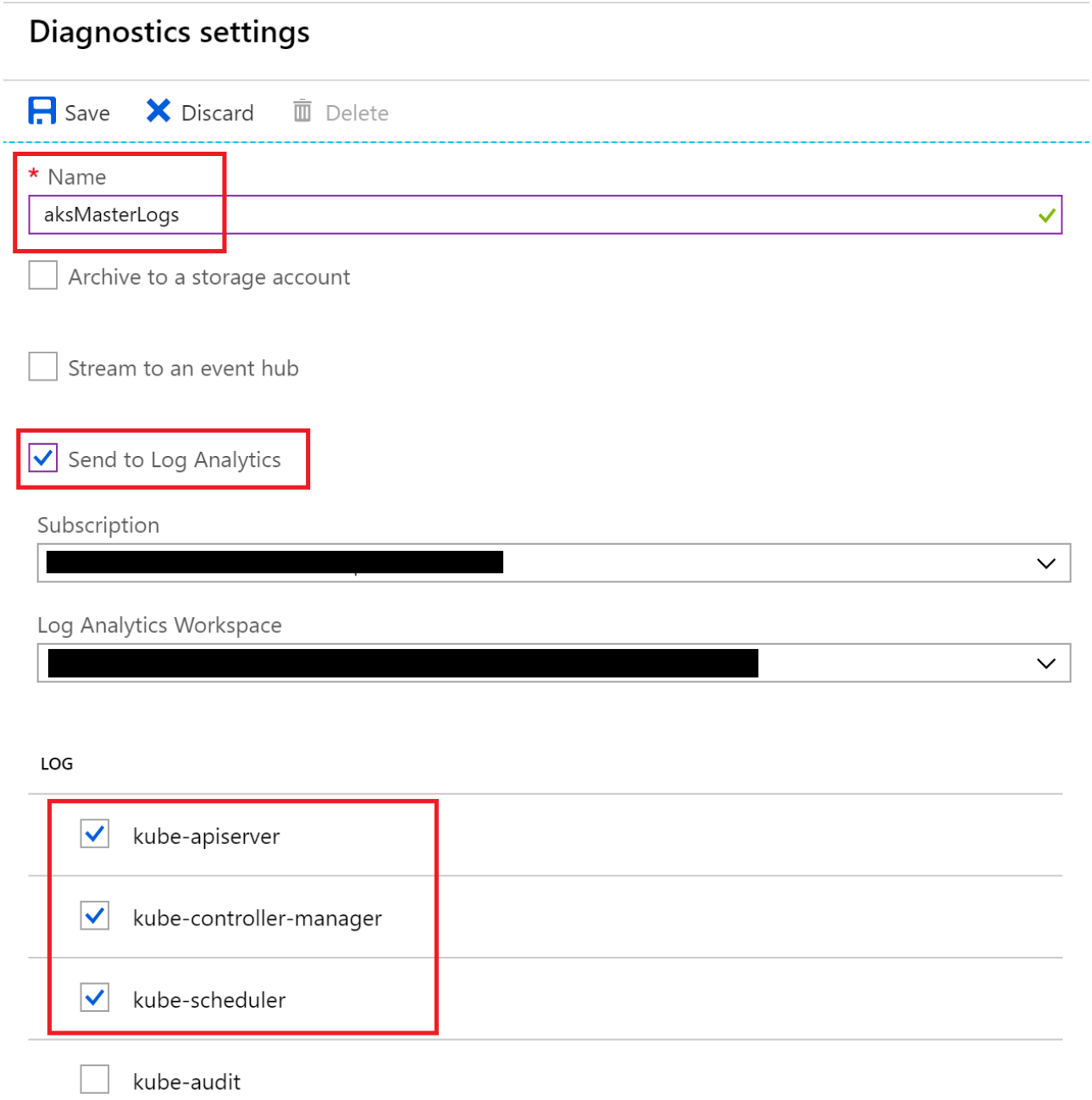
2. Choose the **Diagnostic settings** blade at the Resource Group level.



3. Select your AKS cluster resource

4. Click **Add diagnostic setting**, then provide a **Name**, select your **Log Analytics Workspace**, and select which master node logs you are interested in. When you have finished inputting the information shown in the image below, click **Save**.

> Note: It may take several minutes for the Master node logs to start syncing with Log Analytics.



5. To understand how you can use Azure Monitor for Containers to analyze your AKS cluster,leverage the insight provided in the official Azure Monitor documentation: https://docs.microsoft.com/en-us/azure/azure-monitor/insights/container-insights-analyze

Congratulations!

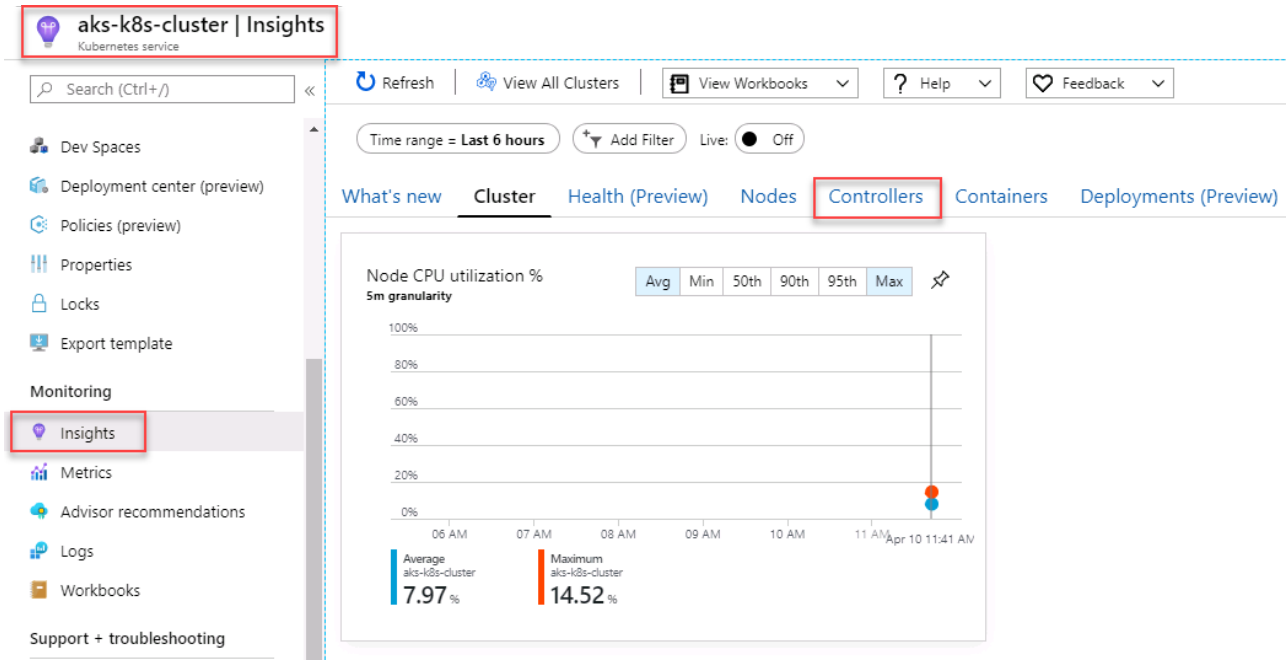
You have successfully completed this exercise. Click **Next** to advance to the next exercise.

Exercise 3: Get access to container logs

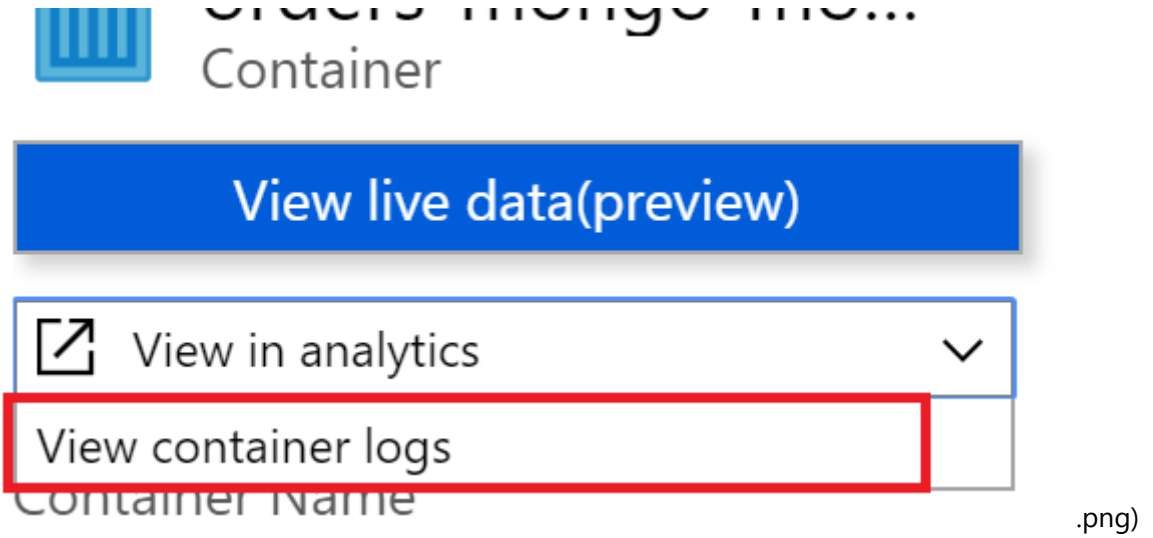
Now that we understand the functionality provided through Azure Monitor for Containers, lets drill in on how to access the container logs for troubleshooting.

Return to list of exercises - Return to list of modules

1. Navigate to the **Kubernetes CLuster** then the **Insights** section under Monitoring. Finally, select the **Containers** tab. Select one of the available containers running in the cluster, be it an instance of the **Web App** container or the **Web API** container.

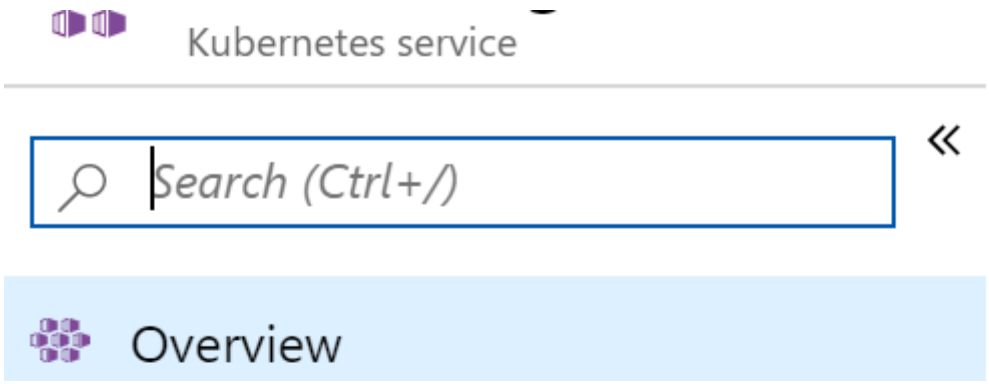


2. Once you have selected a running container,you will see a property pane appear on the right-hand side your screen. From the property pane, select **View in analytics - View container logs**. This output will mirror the output obtained by running `kubectl logs` on a particular pod.



.png)

> *Note: You can also access the logs for your cluster directly from the AKS cluster resource under **Logs**.*




Activity log


Access control (IAM)


Tags

Settings


Node pools (preview)


Upgrade


Scale


Dev Spaces


Deployment center (preview)


Policies (preview)

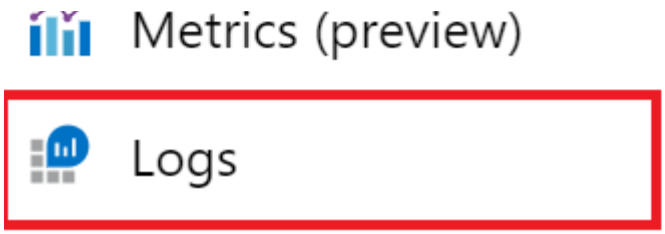

Properties


Locks


Export template

Monitoring


Insights



3. Run a sample query to list all of a container's lifecycle information.

```
ContainerInventory | project Computer, Name, Image, ImageTag, ContainerState,
CreatedTime, StartedTime, FinishedTime | render table
```

4. Run a sample query to view logs from the master node

```
AzureDiagnostics
```

5. View logs specifically from the kube-apiserver

```
AzureDiagnostics | where Category == "kube-apiserver" | project log_s
```

6. To dig deeper into log queries and alerting, visit the following documentation:

    - Analyze Data with Log Queries
    - Alerting

Congratulations!

You have successfully completed this lab. To mark the lab as complete click **End**

