Lab: Create a Docker container for a simple .NET Core app and run it locally

Activity: Containerize a .NET app (Console)

In this tutorial, you learn how to containerize a .NET application with Docker. Containers have many features and benefits, such as being an immutable infrastructure, providing a portable architecture, and enabling scalability. The image can be used to create containers for your local development environment, private cloud, or public cloud.

In this tutorial, you:

> - *Create and publish a simple .NET app*
> - *Create and configure a Dockerfile for .NET*
> - *Build a Docker image*
> - *Create and run a Docker container*

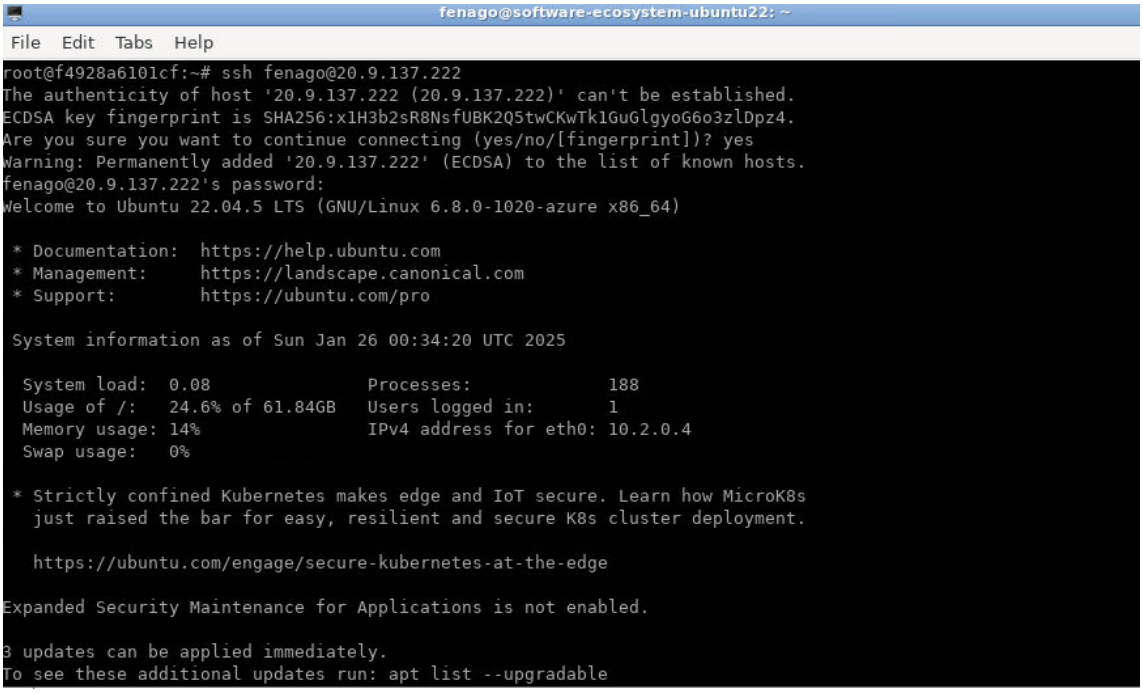
You explore the Docker container build and deploy tasks for a .NET application. The *Docker platform* uses the *Docker engine* to quickly build and package apps as *Docker images*. These images are written in the *Dockerfile* format to be deployed and run in a layered container.

IMPORTANT: Open Terminal and Connect via SSH

1. Open a terminal on your local machine.
2. Use the following command to connect to the remote machine via SSH:

   ```
   ssh username@remote_host
   ```

   Replace `username` with your SSH username and `remote_host` with the hostname of your lab environment. **Note:** Instructor will provide username & password.
3. If prompted, enter your SSH password to establish the connection.



Note: You can connect using SSH with remote lab machine using your local machine.

Prerequisites

- Use the `dotnet --info` command to determine which SDK you're using.

Create .NET app

You need a .NET app that the Docker container runs. Open your terminal, create a working folder if you haven't already, and enter it. In the working folder, run the following command to create a new project in a subdirectory named *App*:

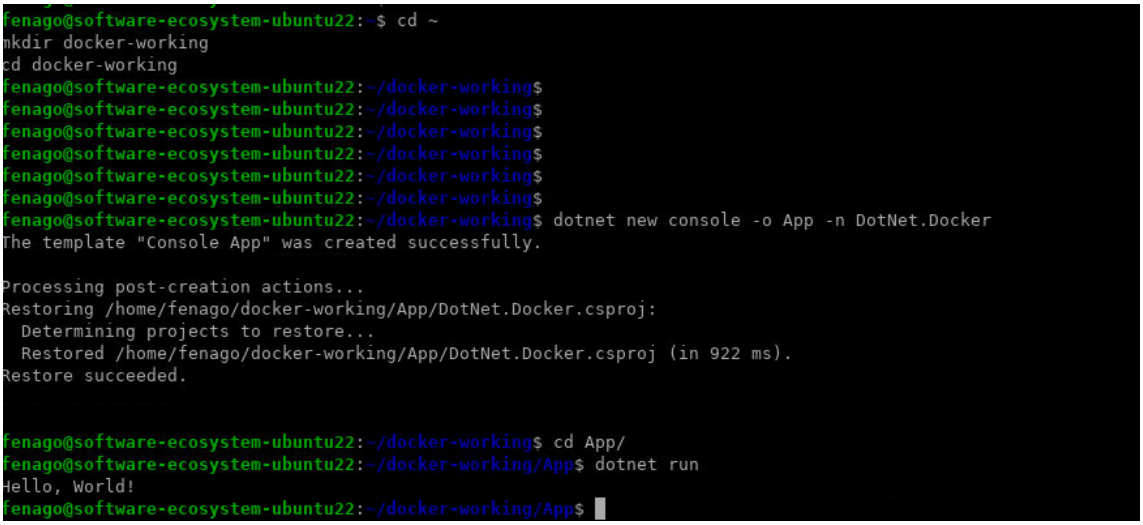
```
cd ~
mkdir docker-working
cd docker-working
dotnet new console -o App -n DotNet.Docker
```

Your folder tree looks similar to the following directory structure:

```
docker-working
    └──App
        ├──DotNet.Docker.csproj
        ├──Program.cs
        └──obj
            ├── DotNet.Docker.csproj.nuget.dgspec.json
            ├── DotNet.Docker.csproj.nuget.g.props
            ├── DotNet.Docker.csproj.nuget.g.targets
            ├── project.assets.json
            └── project.nuget.cache
```

The `dotnet new` command creates a new folder named *App* and generates a "Hello World" console application. Now, you change directories and navigate into the *App* folder from your terminal session. Use the `dotnet run` command to start the app. The application runs, and prints `Hello World!` below the command:

```
cd App
dotnet run
```



The default template creates an app that prints to the terminal and then immediately terminates. For this tutorial, you use an app that loops indefinitely. Open the *Program.cs* file in a text editor.

The *Program.cs* should look like the following C# code:

```
Console.WriteLine("Hello World!");
```

Replace the file with the following code that counts numbers every second:

```
var counter = 0;
var max = args.Length is not 0 ? Convert.ToInt32(args[0]) : -1;

while (max is -1 || counter < max)
{
    Console.WriteLine($"Counter: {++counter}");

    await Task.Delay(TimeSpan.FromMilliseconds(1_000));
}
```

Save the file and test the program again with `dotnet run`. Remember that this app runs indefinitely. Use the cancel command `Ctrl+C` to stop it. Consider the following example output:

```
dotnet run
Counter: 1
Counter: 2
Counter: 3
Counter: 4
^C
```

If you pass a number on the command line to the app, it limits the count to that amount and then exits. Try it with `dotnet run -- 5` to count to five.

IMPORTANT: Any parameters after `--` aren't passed to the `dotnet run` command and instead are passed to your application.

Publish .NET app

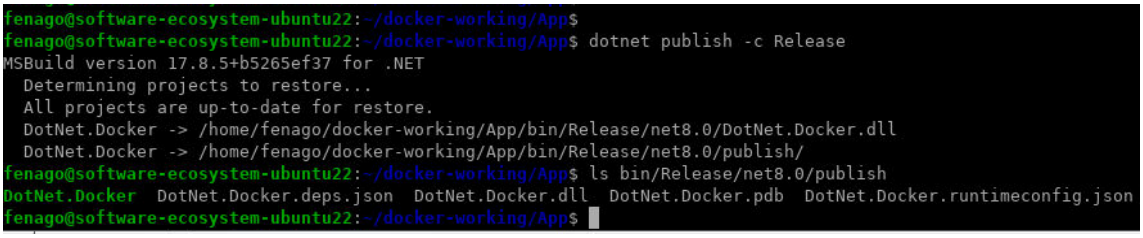
In order for the app to be suitable for an image creation it has to compile. The `dotnet publish` command is most apt for this, as it builds and publishes the app.

```
dotnet publish -c Release
```

The `dotnet publish` command compiles your app to the *publish* folder. The path to the *publish* folder from the working folder should be *./App/bin/Release/<TFM>/publish/*:

Use the `ls` command to get a directory listing and verify that the *DotNet.Docker.dll* file was created.

```
me@DESKTOP:/docker-working/app$ ls bin/Release/net8.0/publish
DotNet.Docker.deps.json  DotNet.Docker.dll  DotNet.Docker.exe  DotNet.Docker.pdb
DotNet.Docker.runtimeconfig.json
```



Create the Dockerfile

The *Dockerfile* file is used by the `docker build` command to create a container image. This file is a text file named *Dockerfile* that doesn't have an extension.

Create a file named *Dockerfile* in the directory containing the *.csproj* and open it in a text editor. This tutorial uses the ASP.NET Core runtime image (which contains the .NET runtime image) and corresponds with the .NET console application.

```
FROM
mcr.microsoft.com/dotnet/sdk:8.0@sha256:35792ea4ad1db051981f62b313f1be3b46b1f45cadbaa3c2
 AS build
WORKDIR /App

# Copy everything
COPY . ./
# Restore as distinct layers
RUN dotnet restore
# Build and publish a release
RUN dotnet publish -o out

# Build runtime image
FROM
mcr.microsoft.com/dotnet/aspnet:8.0@sha256:6c4df091e4e531bb93bdbfe7e7f0998e7ced344f5442

WORKDIR /App
```

```
COPY --from=build /App/out .
ENTRYPOINT ["dotnet", "DotNet.Docker.dll"]
```

> *TIP: This Dockerfile uses multi-stage builds, which optimize the final size of the image by layering the build and leaving only required artifacts.*

The `FROM` keyword requires a fully qualified Docker container image name. The Microsoft Container Registry (MCR, mcr.microsoft.com) is a syndicate of Docker Hub, which hosts publicly accessible containers. The `dotnet` segment is the container repository, whereas the `sdk` or `aspnet` segment is the container image name. The image is tagged with `9.0`, which is used for versioning. Thus, `mcr.microsoft.com/dotnet/aspnet:9.0` is the .NET 9.0 runtime. Make sure that you pull the runtime version that matches the runtime targeted by your SDK. For example, the app created in the previous section used the .NET 9.0 SDK, and the base image referred to in the *Dockerfile* is tagged with **9.0**.

Save the *Dockerfile* file. The directory structure of the working folder should look like the following. Some of the deeper-level files and folders are omitted to save space in the article:

```
docker-working
    └── App
        ├── Dockerfile
        ├── DotNet.Docker.csproj
        ├── Program.cs
        ├── bin
        │   └── Release
        │       └── net9.0
        │           ├── publish
        │           │   ├── DotNet.Docker.deps.json
        │           │   ├── DotNet.Docker.dll
        │           │   ├── DotNet.Docker.exe
        │           │   ├── DotNet.Docker.pdb
        │           │   └── DotNet.Docker.runtimeconfig.json
        │           ├── DotNet.Docker.deps.json
        │           ├── DotNet.Docker.dll
        │           ├── DotNet.Docker.exe
        │           ├── DotNet.Docker.pdb
        │           └── DotNet.Docker.runtimeconfig.json
        └── obj
            └── ...
```

The `FROM` keyword requires a fully qualified Docker container image name. The Microsoft Container Registry (MCR, mcr.microsoft.com) is a syndicate of Docker Hub, which hosts publicly accessible containers. The `dotnet` segment is the container repository, whereas the `sdk` or `aspnet` segment is the container image name. The image is tagged with `8.0`, which is used for versioning. Thus, `mcr.microsoft.com/dotnet/aspnet:8.0` is the .NET 8.0 runtime. Make sure that you pull the runtime version that matches the runtime targeted by your SDK. For example, the app created in the previous section used the .NET 8.0 SDK, and the base image referred to in the *Dockerfile* is tagged with **8.0**.

Save the *Dockerfile* file. The directory structure of the working folder should look like the following. Some of the deeper-level files and folders are omitted to save space in the article:

```
docker-working
    └── App
        ├── Dockerfile
```

```
        ├── DotNet.Docker.csproj
        ├── Program.cs
        ├──🗁 bin
        │   └──🗁 Release
        │       └──🗁 net8.0
        │           └──🗁 publish
        │               ├── DotNet.Docker.deps.json
        │               ├── DotNet.Docker.exe
        │               ├── DotNet.Docker.dll
        │               ├── DotNet.Docker.pdb
        │               └── DotNet.Docker.runtimeconfig.json
        └──🗀 obj
            └──...
```

To build the container, from your terminal, run the following command:

```
docker build -t counter-image -f Dockerfile .
```

Docker processes each line in the *Dockerfile*. The `.` in the `docker build` command sets the build context of the image. The `-f` switch is the path to the *Dockerfile*. This command builds the image and creates a local repository named **counter-image** that points to that image. After this command finishes, run `docker images` to see a list of images installed:

```
docker images
REPOSITORY       TAG       IMAGE ID       CREATED          SIZE
counter-image    latest    2f15637dc1f6   10 minutes ago   217MB
```

The `counter-image` repository is the name of the image. Additionally, the image tag, image identifier, size and when it was created are all part of the output. The final steps of the *Dockerfile* are to create a container from the image and run the app, copy the published app to the container, and define the entry point.

The `FROM` command specifies the base image and tag to use. The `WORKDIR` command changes the **current directory** inside of the container to *App*.

The `COPY` command tells Docker to copy the specified source directory to a destination folder. In this example, the *publish* contents in the `build` layer are output into the folder named *App/out*, so it's the source to copy from. All of the published contents in the *App/out* directory are copied into current working directory (*App*).

The next command, `ENTRYPOINT`, tells Docker to configure the container to run as an executable. When the container starts, the `ENTRYPOINT` command runs. When this command ends, the container automatically stops.

Create a container

Now that you have an image that contains your app, you can create a container. You can create a container in two ways. First, create a new container that is stopped.

```
docker create --name core-counter counter-image
```

This `docker create` command creates a container based on the **counter-image** image. The output of the `docker create` command shows you the **CONTAINER ID** of the container (your identifier will be different):

```
d0be06126f7db6dd1cee369d911262a353c9b7fb4829a0c11b4b2eb7b2d429cf
```

To see a list of *all* containers, use the `docker ps -a` command:

```
docker ps -a --filter "name=core-counter"

CONTAINER ID   IMAGE           COMMAND                  CREATED          STATUS
PORTS     NAMES
d0be06126f7d   counter-image   "dotnet DotNet.Docke…"   12 seconds ago   Created
core-counter
```

Manage the container

The container was created with a specific name `core-counter`. This name is used to manage the container. The following example uses the `docker start` command to start the container, and then uses the `docker ps` command to only show containers that are running:

```
docker start core-counter


docker ps --filter "name=core-counter"
CONTAINER ID   IMAGE           COMMAND                  CREATED          STATUS
PORTS     NAMES
cf01364df453   counter-image   "dotnet DotNet.Docke…"   53 seconds ago   Up 10 seconds
core-counter
```

Similarly, the `docker stop` command stops the container. The following example uses the `docker stop` command to stop the container:

```
docker stop core-counter
```

Connect to a container

After a container is running, you can connect to it to see the output. Use the `docker start` and `docker attach` commands to start the container and peek at the output stream. In this example, the `Ctrl+C` keystroke is used to detach from the running container. This keystroke ends the process in the container unless otherwise specified, which would stop the container. The `--sig-proxy=false` parameter ensures that `Ctrl+C` doesn't stop the process in the container.

After you detach from the container, reattach to verify that it's still running and counting.

```
docker start core-counter
core-counter

docker attach --sig-proxy=false core-counter
Counter: 7
Counter: 8
Counter: 9
^C

docker attach --sig-proxy=false core-counter
Counter: 17
Counter: 18
Counter: 19
^C
```

```
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ docker start core-counter
core-counter
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ docker attach --sig-proxy=false core-counter
Counter: 26
Counter: 27
Counter: 28
Counter: 29
Counter: 30
Counter: 31
^CCounter: 32
Counter: 33
Counter: 34
Counter: 35
^CCounter: 36
Counter: 37
Counter: 38
Counter: 39
^C
got 3 SIGTERM/SIGINTs, forcefully exiting
```

Delete a container

For this article, you don't want containers hanging around that don't do anything. Delete the container you previously created. If the container is running, stop it.

```
docker stop core-counter
```

The following example lists all containers. It then uses the `docker rm` command to delete the container and then checks a second time for any running containers.

```
docker ps -a --filter "name=core-counter"

docker rm core-counter

docker ps -a --filter "name=core-counter"
```

```
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ ^C
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ docker stop core-counter
core-counter
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ docker ps -a --filter "name=core-counter"
CONTAINER ID   IMAGE           COMMAND                  CREATED         STATUS                       PORTS     NAMES
aba74a05a04b   counter-image   "dotnet DotNet.Docke…"   7 minutes ago   Exited (143) 10 seconds ago            core-counter
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ docker rm core-counter
core-counter
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ docker ps -a --filter "name=core-counter"
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
fenago@software-ecosystem-ubuntu22:~/docker-working/App$
```

Single run

Docker provides the `docker run` command to create and run the container as a single command. This command eliminates the need to run `docker create` and then `docker start`. You can also set this command to automatically delete the container when the container stops. For example, use `docker run -it --rm` to do two things, first, automatically use the current terminal to connect to the container, and then when the container finishes, remove it:

```
docker run -it --rm counter-image
Counter: 1
```

```
Counter: 2
Counter: 3
Counter: 4
Counter: 5
^C
```

The container also passes parameters into the execution of the .NET app. To instruct the .NET app to count only to three, pass in 3.

```
docker run -it --rm counter-image 3
Counter: 1
Counter: 2
Counter: 3
```

With `docker run -it`, the `Ctrl+C` command stops the process that's running in the container, which in turn, stops the container. Since the `--rm` parameter was provided, the container is automatically deleted when the process is stopped.

Change the ENTRYPOINT

The `docker run` command also lets you modify the `ENTRYPOINT` command from the *Dockerfile* and run something else, but only for that container. For example, use the following command to run `bash` or `cmd.exe`. Edit the command as necessary.

In this step, `ENTRYPOINT` is changed to `bash`. The `exit` command is run which ends the process and stop the container.

```
docker run -it --rm --entrypoint "bash" counter-image

ls

dotnet DotNet.Docker.dll 7

exit
```

```
fenago@software-ecosystem-ubuntu22:~/docker-working/App$ docker run -it --rm --entrypoint "bash" counter-image
root@2595f2be90d7:/App# ls
DotNet.Docker  DotNet.Docker.deps.json  DotNet.Docker.dll  DotNet.Docker.pdb  DotNet.Docker.runtimeconfig.json
root@2595f2be90d7:/App# dotnet DotNet.Docker.dll 7
Counter: 1



Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
root@2595f2be90d7:/App#
root@2595f2be90d7:/App#
root@2595f2be90d7:/App#
root@2595f2be90d7:/App#
root@2595f2be90d7:/App# exit
exit
```

Clean up resources

During this tutorial, you created containers and images. If you want, delete these resources. Use the following commands to

1. List all containers

```
docker ps -a --filter "name=core-counter"
```

Note: You can skip deleting the container step if no container exits.

2. Stop containers that are running by their name.

```
docker stop core-counter
```

3. Delete the container

```
docker rm core-counter
```

Activity: Containerize a .NET app (Web)

Step 1: Verify the .NET SDK

```
dotnet --version
```

Step 2: Create a Simple .NET Core Application

1. **Create a new directory for your project:**

```
cd ~
mkdir MyDotNetApp && cd MyDotNetApp
```

2. **Initialize a new .NET project:**

```
dotnet new web -n MyDotNetApp
```

3. **Navigate to the project directory:**

```
cd MyDotNetApp
```

4. **Update the `Program.cs` file to listen on all network interfaces:** Edit the `Program.cs` file:

```
nano Program.cs
```

Replace the contents with:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello, Dockerized .NET App!");

app.Run("http://0.0.0.0:5000");
```

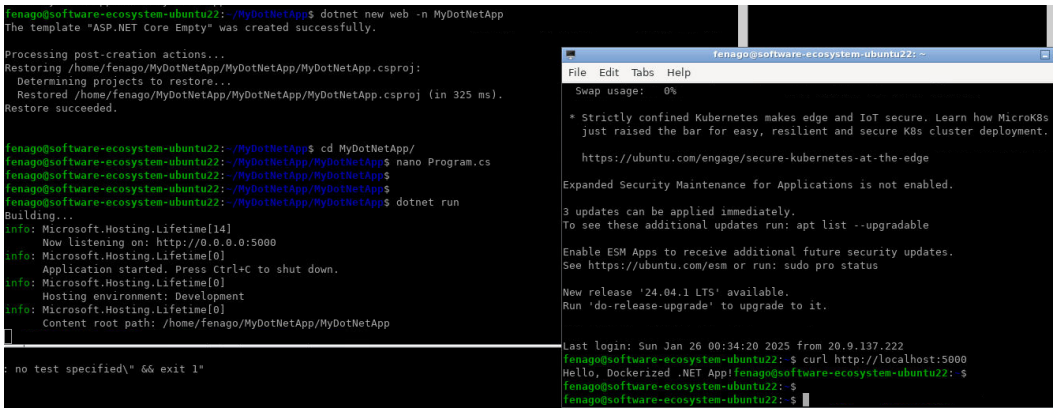
```
fenago@software-ecosystem-ubuntu22:~/MyDotNetApp$ dotnet new web -n MyDotNetApp
The template "ASP.NET Core Empty" was created successfully.

Processing post-creation actions...
Restoring /home/fenago/MyDotNetApp/MyDotNetApp/MyDotNetApp.csproj:
  Determining projects to restore...
  Restored /home/fenago/MyDotNetApp/MyDotNetApp/MyDotNetApp.csproj (in 325 ms).
Restore succeeded.


fenago@software-ecosystem-ubuntu22:~/MyDotNetApp$ cd MyDotNetApp/
fenago@software-ecosystem-ubuntu22:~/MyDotNetApp/MyDotNetApp$ nano Program.cs
fenago@software-ecosystem-ubuntu22:~/MyDotNetApp/MyDotNetApp$
```

5. **Run the application to verify it works:**

```
dotnet run
```

IMPORTANT: Open new terminal and connect with SSH again with remote lab machine and use `curl` to verify:

```
curl http://localhost:5000
```



NOTE: Make sure to exit the dotnet program before proceeding to next step.

Step 3: Create a Dockerfile

1. **Create a `Dockerfile` in the project directory:**

```
nano Dockerfile
```

2. **Add the following content to the `Dockerfile` :**

```
# Use the official .NET runtime image
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

# Use the official SDK image for building the app
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /app
```

```
# Copy project files and restore dependencies
COPY . ./
RUN dotnet restore

# Build the app
RUN dotnet publish -c Release -o /app/publish

# Final stage: Use runtime image and copy built files
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY --from=build /app/publish .

# Expose port 5000
EXPOSE 5000

# Specify the entry point for the container
ENTRYPOINT ["dotnet", "MyDotNetApp.dll"]
```

3. **Save and exit the file.**

Step 4: Build and Run the Docker Container

1. **Build the Docker image:**

```
docker build -t my-dotnet-app .
```

2. **Verify the Docker image is created:**

```
docker images
```

3. **Run the Docker container with port mapping:**

```
docker run --name my-dotnet-container -d -p 5000:5000 my-dotnet-app
```

4. **Verify the app is running using `curl`:**

```
curl http://localhost:5000
```

You should see the response:

```
Hello, Dockerized .NET App!
```

5. **Access the app from another machine on the same network:** Replace `localhost` with the server's IP address:

```
curl http://<server-ip>:5000
```

Step 5: Clean Up

1. **Stop and remove the container:**

```
docker stop my-dotnet-container
docker rm my-dotnet-container
```

Task: Push the Image to Docker Hub (Optional)

1. **Sign up for a Docker Hub account:**
   - Go to Docker Hub and create a free account if you don't already have one.
2. **Log in to Docker Hub from your terminal:**

   ```
   docker login
   ```

3. **Tag your Docker image with your Docker Hub username:** Replace `<your-username>` with your Docker Hub username:

   ```
   docker tag my-dotnet-app <your-username>/my-dotnet-app
   ```

4. **Push the image to Docker Hub:**

   ```
   docker push <your-username>/my-dotnet-app
   ```

5. **Verify the image is available on Docker Hub:**
   - Log in to your Docker Hub account in your browser.
   - Navigate to the `Repositories` section and confirm that the `my-dotnet-app` image is listed.

You have successfully created, pushed, and run a Docker container for a .NET Core application on Ubuntu. 🎉