Tutorial: Containerize a .NET Core app

01/09/2020 • 11 minutes to read • +5

In this article

This tutorial teaches you how to build a Docker image that contains your .NET Core application. The image can be used to create containers for your local development environment, private cloud, or public cloud.

You'll learn to:

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

You'll understand the Docker container build and deploy tasks for a .NET Core 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.

Tip

If you're working with an existing ASP.NET Core application, see the Learn how to containerize an ASP.NET Core application tutorial.

Prerequisites

Install the following prerequisites:

- .NET Core 3.1 SDK

  If you have .NET Core installed, use the `dotnet --info` command to determine which SDK you're using.

- Docker Community Edition

- A temporary working folder for the *Dockerfile* and .NET Core example app. In this tutorial, the name *docker-working* is used as the working folder.

Create .NET Core app

You need a .NET Core app that the Docker container will run. 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*:

.NET Core CLI Copy

```
dotnet new console -o app -n myapp
```

Your folder tree will look like the following:

Copy

```
docker-working
│
└──app
    │   myapp.csproj
    │   Program.cs
    │
    └──obj
            myapp.csproj.nuget.cache
            myapp.csproj.nuget.dgspec.json
            myapp.csproj.nuget.g.props
            myapp.csproj.nuget.g.targets
            project.assets.json
```

The `dotnet new` command creates a new folder named *app* and generates a "Hello World" app. Enter the *app* folder and run the command `dotnet run`. You'll see the following output:

console Copy

```
> dotnet run
Hello World!
```

The default template creates an app that prints to the terminal and then exits. For this tutorial, you'll use an app that loops indefinitely. Open the *Program.cs* file in a text editor. It should currently look like the following code:

C# Copy

```
using System;

namespace myapp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

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

C# Copy

```
using System;

namespace myapp
{
    class Program
    {
        static void Main(string[] args)
        {
            var counter = 0;
            var max = args.Length != 0 ? Convert.ToInt32(args[0]) :
-1;
            while (max == -1 || counter < max)
            {
                counter++;
                Console.WriteLine($"Counter: {counter}");
                System.Threading.Tasks.Task.Delay(1000).Wait();
            }
        }
    }
}
```

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. You'll see the following output:

console Copy

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

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

> ⓘ Note
>
> Any parameters after `--` are not passed to the `dotnet run` command and instead are passed to your application.

Publish .NET Core app

Before you add your .NET Core app to the Docker image, publish it. You want to make sure that the container runs the published version of the app when it's started.

From the working folder, enter the *app* folder with the example source code and run the following command:

.NET Core CLI Copy

```
dotnet publish -c Release
```

This command compiles your app to the *publish* folder. The path to the *publish* folder from the working folder should be `.\app\bin\Release\netcoreapp3.1\publish\`

From the *app* folder, get a directory listing of the publish folder to verify that the *myapp.dll* file was created.

console Copy

```
> dir bin\Release\netcoreapp3.1\publish

    Directory:  C:\docker-working\app\bin\Release\netcoreapp3.1\pub-
lish

01/09/2020  11:41 AM    <DIR>          .
01/09/2020  11:41 AM    <DIR>          ..
01/09/2020  11:41 AM               407 myapp.deps.json
01/09/2020  12:15 PM             4,608 myapp.dll
01/09/2020  12:15 PM           169,984 myapp.exe
```

```
01/09/2020  12:15 PM               736 myapp.pdb
01/09/2020  11:41 AM               154 myapp.runtimeconfig.json
```

If you're using Linux or macOS, use the `ls` command to get a directory listing and verify that the *myapp.dll* file was created.

bash Copy

```bash
me@DESKTOP:/docker-working/app$ ls bin/Release/netcoreapp3.1/publish
myapp.deps.json  myapp.dll  myapp.pdb  myapp.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.

In your terminal, navigate up a directory to the working folder you created at the start. Create a file named *Dockerfile* in your working folder and open it in a text editor. Depending on the type of application you're going to containerize, you'll choose either the ASP.NET Core runtime or the .NET Core runtime. When in doubt, choose the ASP.NET Core runtime, which includes the .NET Core runtime. This tutorial will use the ASP.NET Core runtime image, but the application created in the previous sections is an .NET Core application.

- ASP.NET Core runtime

  Dockerfile Copy

  ```Dockerfile
  FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
  ```

- .NET Core runtime

  Dockerfile Copy

  ```Dockerfile
  FROM mcr.microsoft.com/dotnet/core/runtime:3.1
  ```

The `FROM` command tells Docker to pull down the image tagged **3.1** from the specified repository. 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 Core 3.1 SDK and the base image referred to in the *Dockerfile* is tagged with **3.1**.

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

Copy

```
docker-working
│   Dockerfile
│
└───app
    │   myapp.csproj
    │   Program.cs
    │
    ├───bin
    │   └───Release
    │       └───netcoreapp3.1
    │           └───publish
    │                   myapp.deps.json
    │                   myapp.exe
    │                   myapp.dll
    │                   myapp.pdb
    │                   myapp.runtimeconfig.json
    │
    └───obj
```

From your terminal, run the following command:

console Copy

```console
docker build -t myimage -f Dockerfile .
```

Docker will process each line in the *Dockerfile*. The `.` in the `docker build` command tells Docker to use the current folder to find a *Dockerfile*. This command builds the image and creates a local repository named **myimage** that points to that image. After this command finishes, run `docker images` to see a list of images installed:

console Copy

```console
> docker images
REPOSITORY                              TAG                 IMAGE ID
CREATED             SIZE
myimage                                 latest              38db0e-
b8f648        4 weeks ago         346MB
mcr.microsoft.com/dotnet/core/aspnet    3.1                 38db0e-
b8f648        4 weeks ago         346MB
```

Notice that the two images share the same **IMAGE ID** value. The value is the same between both images because the only command in the *Dockerfile* was to base the new image on an existing image. Let's add two commands to the *Dockerfile*. Each command creates a new image layer with the final command representing the image the **myimage** repository entry points to.

Dockerfile Copy

```
COPY app/bin/Release/netcoreapp3.1/publish/ app/

ENTRYPOINT ["dotnet", "app/myapp.dll"]
```

The `COPY` command tells Docker to copy the specified folder on your computer to a folder in the container. In this example, the *publish* folder is copied to a folder named *app* in the container.

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 will automatically stop.

From your terminal, run `docker build -t myimage -f Dockerfile .` and when that command finishes, run `docker images`.

console Copy

```
> docker build -t myimage -f Dockerfile .
Sending build context to Docker daemon  1.624MB
Step 1/3 : FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
 ---> 38db0eb8f648
Step 2/3 : COPY app/bin/Release/netcoreapp3.1/publish/ app/
 ---> 37873673e468
Step 3/3 : ENTRYPOINT ["dotnet", "app/myapp.dll"]
 ---> Running in d8deb7b3aa9e
Removing intermediate container d8deb7b3aa9e
 ---> 0d602ca35c1d
Successfully built 0d602ca35c1d
Successfully tagged myimage:latest

> docker images
REPOSITORY                              TAG                 IMAGE ID
CREATED             SIZE
myimage                                 latest              0d602-
ca35c1d        4 seconds ago       346MB
mcr.microsoft.com/dotnet/core/aspnet    3.1                 38db0e-
b8f648        4 weeks ago         346MB
```

Each command in the *Dockerfile* generated a layer and created an **IMAGE ID**. The final **IMAGE ID** (yours will be different) is **ddcc6646461b** and next you'll create a container based on this image.

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.

console Copy

```
> docker create myimage
ceda87b219a4e55e9ad5d833ee1a7ea4da21b5ea7ce5a7d08f3051152e784944
```

The `docker create` command from above will create a container based on the **myimage** image. The output of that command shows you the **CONTAINER ID** (yours will be different) of the created container. To see a list of *all* containers, use the `docker ps -a` command:

console Copy

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                   CRE-
ATED               STATUS         PORTS    NAMES
ceda87b219a4        myimage             "dotnet app/myapp.dll"   4
seconds ago         Created             gallant_lehmann
```

Manage the container

Each container is assigned a random name that you can use to refer to that container instance. For example, the container that was created automatically chose the name **gallant_lehmann** (yours will be different) and that name can be used to start the container. You override the automatic name with a specific one by using the `docker create --name` parameter.

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:

console Copy

```
> docker start gallant_lehmann
gallant_lehmann

> docker ps
CONTAINER ID        IMAGE               COMMAND                   CRE-
ATED               STATUS          PORTS    NAMES
ceda87b219a4        myimage             "dotnet app/myapp.dll"   7
minutes ago         Up 8 seconds         gallant_lehmann
```

Similarly, the `docker stop` command will stop the container. The following example uses the `docker stop` command to stop the container, and then uses the `docker ps` command to show that no containers are running:

console Copy

```console
> docker stop gallant_lehmann
gallant_lehmann

> docker ps
CONTAINER ID        IMAGE                   COMMAND                   CREATED
STATUS      PORTS    NAMES
```

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 may actually end the process in the container, which will stop the container. The `--sig-proxy=false` parameter ensures that `CTRL + C` won't stop the process in the container.

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

console Copy

```console
> docker start gallant_lehmann
gallant_lehmann

> docker attach --sig-proxy=false gallant_lehmann
Counter: 7
Counter: 8
Counter: 9
^C

> docker attach --sig-proxy=false gallant_lehmann
Counter: 17
Counter: 18
Counter: 19
^C
```

Delete a container

For the purposes of this article you don't want containers just hanging around doing nothing. Delete the container you previously created. If the container is running, stop it.

console Copy

```console
> docker stop gallant_lehmann
```

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.

console Copy

```console
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CRE-
ATED             STATUS      PORTS    NAMES
ceda87b219a4        myimage             "dotnet app/myapp.dll"   19
minutes ago      Exited              gallant_lehmann

> docker rm gallant_lehmann
gallant_lehmann

> docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS     PORTS    NAMES
```

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:

console Copy

```console
> docker run -it --rm myimage
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
^C
```

With `docker run -it`, the `CTRL + C` command will stop process that is 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. Verify that it doesn't exist:

console Copy

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CRE-
ATED             STATUS     PORTS    NAMES
```

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.

Windows

In this example, `ENTRYPOINT` is changed to `cmd.exe`. CTRL+C is pressed to end the process and stop the container.

console Copy

```
> docker run -it --rm --entrypoint "cmd.exe" myimage

Microsoft Windows [Version 10.0.17763.379]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\>dir
 Volume in drive C has no label.
 Volume Serial Number is 3005-1E84

 Directory of C:\

04/09/2019  08:46 AM    <DIR>          app
03/07/2019  10:25 AM             5,510 License.txt
04/02/2019  01:35 PM    <DIR>          Program Files
04/09/2019  01:06 PM    <DIR>          Users
04/02/2019  01:35 PM    <DIR>          Windows
               1 File(s)          5,510 bytes
               4 Dir(s)  21,246,517,248 bytes free

C:\>^C
```

Linux

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

bash Copy

```
root@user:~# docker run -it --rm --entrypoint "bash" myimage
root@8515e897c893:/# ls app
myapp.deps.json  myapp.dll  myapp.pdb  myapp.runtimeconfig.json
root@8515e897c893:/# exit
exit
```

Essential commands

Docker has many different commands that cover what you want to do with your container and images. These Docker commands are essential to managing your containers:

- docker build
- docker run
- docker ps
- docker stop
- docker rm
- docker rmi
- docker image

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

   console Copy

   ```
   > docker ps -a
   ```

2. Stop containers that are running. The `CONTAINER_NAME` represents the name automatically assigned to the container.

   console Copy

   ```
   > docker stop CONTAINER_NAME
   ```

3. Delete the container

   console Copy

   ```
   > docker rm CONTAINER_NAME
   ```

Next, delete any images that you no longer want on your machine. Delete the image created by your *Dockerfile* and then delete the .NET Core image the *Dockerfile* was based on. You can use the **IMAGE ID** or the **REPOSITORY:TAG** formatted string.

console Copy

```console
docker rmi myimage:latest
docker rmi mcr.microsoft.com/dotnet/core/aspnet:3.1
```

Use the `docker images` command to see a list of images installed.

> ⓘ **Note**
>
> Image files can be large. Typically, you would remove temporary containers you created while testing and developing your app. You usually keep the base images with the runtime installed if you plan on building other images based on that runtime.

Next steps

- Learn how to containerize an ASP.NET Core application.
- Try the ASP.NET Core Microservice Tutorial.
- Review the Azure services that support containers.
- Read about Dockerfile commands.
- Explore the Container Tools for Visual Studio

Is this page helpful?

Yes No