Tutorial: Containerize a .NET app

Article • 03/21/2024

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.

> ⓘ **Note**
>
> This tutorial **is not** for ASP.NET Core apps. If you're using ASP.NET Core, see the **Learn how to containerize an ASP.NET Core application** tutorial.

Prerequisites

Install the following prerequisites:

- .NET 8+ SDK .
  If you have .NET 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 example app. In this tutorial, the name *docker-working* is used as the working folder.

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*:

.NET CLI

```
dotnet new console -o App -n DotNet.Docker
```

Your folder tree looks similar to the following directory structure:

Directory

```
📁 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:

.NET CLI

```
cd App
dotnet run
Hello World!
```

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.

> **Tip**
>
> If you're using Visual Studio Code, from the previous terminal session type the following command:

Console

```
code .
```

This will open the *App* folder that contains the project in Visual Studio Code.

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

C#

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

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

C#

```
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:

.NET CLI

```
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.

ⓘ Important

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

Publish .NET app

Before adding the .NET app to the Docker image, first it must be published. It's best to have the container run the published version of the app. To publish the app, run the following command:

.NET CLI

```
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\net8.0\publish\`.

Windows

From the *App* folder, get a directory listing of the publish folder to verify that the *DotNet.Docker.dll* file was created.

PowerShell

```
dir .\bin\Release\net8.0\publish\

    Directory: C:\Users\default\App\bin\Release\net8.0\publish

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           9/22/2023  9:17 AM            431 DotNet.Docker.deps.json
-a---           9/22/2023  9:17 AM           6144 DotNet.Docker.dll
-a---           9/22/2023  9:17 AM         157696 DotNet.Docker.exe
-a---           9/22/2023  9:17 AM          11688 DotNet.Docker.pdb
-a---           9/22/2023  9:17 AM            353
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.

docker

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env
WORKDIR /App

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

# Build runtime image
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /App
COPY --from=build-env /App/out .
ENTRYPOINT ["dotnet", "DotNet.Docker.dll"]
```

> **Note**
>
> The ASP.NET Core runtime image is used intentionally here, although the `mcr.microsoft.com/dotnet/runtime:8.0` image could have been used.

> **Tip**
>
> This *Dockerfile* uses multi-stage builds, which optimizes the final size of the image by layering the build and leaving only required artifacts. For more information, see **Docker Docs: multi-stage builds** .

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**.

> **ⓘ Important**
>
> When using Windows-based container images, you need to specify the image tag beyond simply `8.0`, for example, `mcr.microsoft.com/dotnet/aspnet:8.0-nanoserver-1809` instead of `mcr.microsoft.com/dotnet/aspnet:8.0`. Select an image name based on whether you're using Nano Server or Windows Server Core and which version of that OS. You can find a full list of all supported tags on .NET's Docker Hub page .

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 omitted to save space in the article:

Directory

```
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
    └─...
```

The `ENTRYPOINT` instruction sets `dotnet` as the host for the `DotNet.Docker.dll`. However, it's possible to instead define the `ENTRYPOINT` as the app executable itself, relying on the OS as the app host:

Dockerfile

```
ENTRYPOINT ["./DotNet.Docker"]
```

This causes the app to be executed directly, without `dotnet`, and instead relies on the app host and the underlying OS. For more information on deploying cross-platform binaries, see Produce a cross-platform binary.

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

Console

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

Docker will process 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:

Console

```
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. The `latest` tag is the tag that is used to identify the image. The `2f15637dc1f6` is the image ID. The `10 minutes ago` is the time the image was created. The `217MB` is the size of the image. 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.

Dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /App
COPY --from=build-env /App/out .
ENTRYPOINT ["dotnet", "DotNet.Docker.dll"]
```

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-env` layer were 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 will automatically stop.

> **Tip**
>
> Before .NET 8, containers configured to run as read-only may fail with `Failed to create CoreCLR, HRESULT: 0x8007000E`. To address this issue, specify a `DOTNET_EnableDiagnostics` environment variable as `0` (just before the `ENTRYPOINT` step):
>
> Dockerfile
>
> ```
> ENV DOTNET_EnableDiagnostics=0
> ```
>
> For more information on various .NET environment variables, see **.NET environment variables**.

> **Note**
>
> .NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

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

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

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

Console

```
d0be06126f7db6dd1cee369d911262a353c9b7fb4829a0c11b4b2eb7b2d429cf
```

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

Console

```
docker ps -a
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:

Console

```
docker start core-counter
core-counter

docker ps
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, and then uses the `docker ps` command to show that no containers are running:

Console

```
docker stop core-counter
core-counter

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 ends the process in the container unless otherwise specified, which would 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

```
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
```

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.

Console

```
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.

Console

```
docker ps -a
CONTAINER ID    IMAGE            COMMAND                  CREATED
STATUS                         PORTS    NAMES
2f6424a7ddce    counter-image    "dotnet DotNet.Dock…"    7 minutes ago
Exited (143) 20 seconds ago            core-counter

docker rm core-counter
core-counter

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

```
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.

Console

```
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. Verify that it doesn't exist:

Console

```
docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    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

```
docker run -it --rm --entrypoint "cmd.exe" counter-image

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
```

Essential commands

Docker has many different commands that create, manage, and interact with containers 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

   ```
   docker ps -a
   ```

2. Stop containers that are running by their name.

   Console

   ```
   docker stop core-counter
   ```

3. Delete the container

Console

```
docker rm core-counter
```

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

Console

```
docker rmi counter-image:latest
docker rmi mcr.microsoft.com/dotnet/aspnet:8.0
```

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

> **Tip**
>
> 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

- Containerize a .NET app with dotnet publish
- .NET container images
- Containerize an ASP.NET Core application
- Azure services that support containers
- Dockerfile commands
- Container Tools for Visual Studio

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

information, see our contributor guide.

Provide product feedback