Getting Started with .NET Core, Docker, and RabbitMQ — Part 2

Matthew Harper Following
Aug 9, 2019 · 8 min read

Picking up from Part1, we're going to be containerizing the application we've built. Containerization is an approach where an application and its dependencies are packaged together and and run in an isolated environment. A VM (or physical machine) runs the container host, instead of running our applications directly. The container host, in turn, runs our containers. Each container is isolated from the others, and can even run a different operating system than the VM.

In this project, we'll be using Docker, an open-source project for automating the deployment of applications as portable, self sufficient containers that can run almost anywhere. Before we start, let's walk through some of the benefits of containerization.

Consistency & Portability

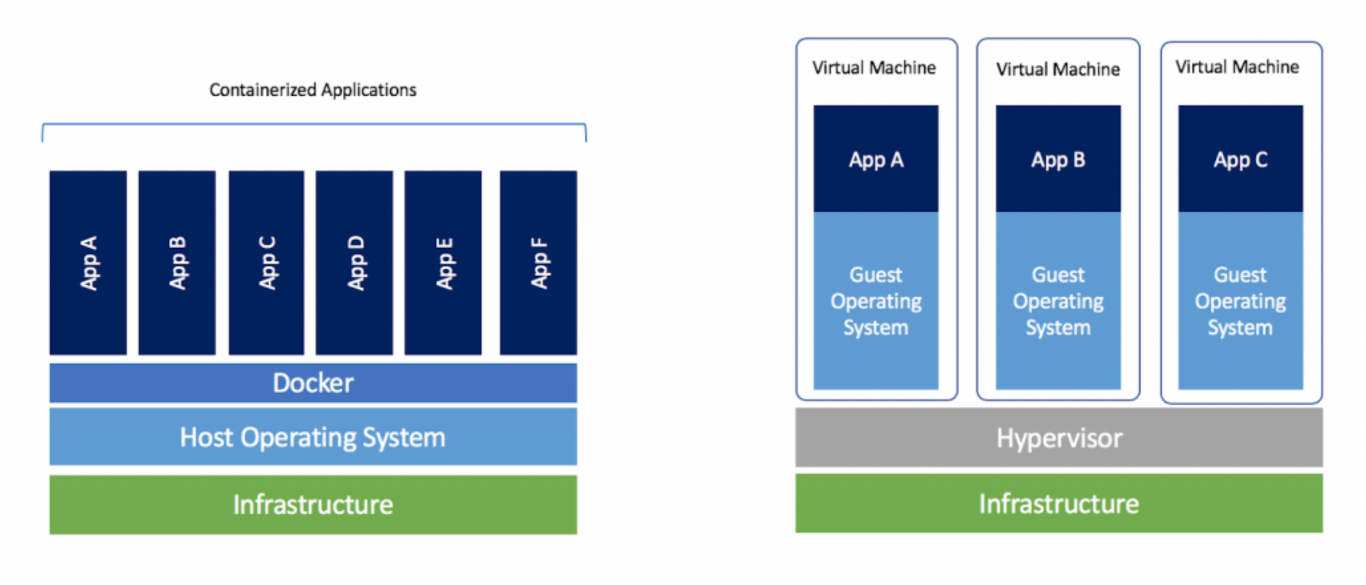
How do you replicate the software you've built on your local machine somewhere else, perhaps on another physical machine or on a virtual machine in the cloud? Are all the run time dependencies (including any specific version requirements) in place? When dependencies change, how do you apply that change across all relevant machines and environments? A frequent symptom of this problem is when you hear "it works fine on my machine" or "why does it work in QA but not in Production?"

Once a container is created, you can run it basically anywhere and it will behave the same way. Because the application and all of its dependencies (including the OS) are encapsulated within the container, there will be no difference in execution between your local machine, another physical machine, or a virtual machine in the cloud. This approach also helps to simplify deployments.

Isolation & Efficiency

Another common problem is VM utilization. To maximize efficiency and minimize cost, you may want to host multiple apps on a single VM. In that scenario, there is no logical boundary between the apps. Containers isolate each application, preventing dependency conflicts. Containers also allow resource limits to be set for each service, preventing a service on a shared VM from consuming all available RAM and starving other critical services. These silos provide more security because your applications aren't running within a shared operating system.

A container also requires fewer resources than a virtual machine (containers don't have any direct interface with hardware, for example). This enables them to launch faster, and allows your application scale quickly in response to heavy traffic.



https://blog.docker.com/2018/08/containers-replacing-virtual-machines/

. . .

We're going to modify our project to run the webAPI and console app within individual containers, and then use a tool called Docker-Compose to launch them together. Hopefully you've completed part 1 of the tutorial so we can pick up right where we left off. If you need the code for a starting point, look for the v1.0 tag in the Github repo. The only additional software you need installed is Docker Community Edition. I'm working on a Windows machine, so that's Docker Desktop for Windows. Follow the installation guide and then we can get started.

The first step in introducing Docker to our project is to add a Dockerfile. A Dockerfile is a text file that contains all of the commands that will be executed to create a Docker image. An image is basically a package with all the dependencies and information needed to create a container. Images are composed of layers; you typically start with an operating system (maybe your container will run linux or windows server core), then install dependencies (such as the .NET Core SDK), and then install your application.

Let's start with the publisher_api project. Add new file called Dockerfile (no file extension) to the publisher_api directory (publisher_api/Dockerfile). You can copy the content below, and then we'll get into what each line is doing.

```
 1  FROM microsoft/dotnet:2.2-sdk AS build
 2  WORKDIR /app
 3
 4  # Copy csproj and restore as distinct layers
 5  COPY *.csproj ./
 6  RUN dotnet restore
 7
 8  # Copy everything else and build
 9  COPY . ./
10  RUN dotnet publish -c Release -o out
11
12  # Build runtime image
13  FROM microsoft/dotnet:2.2-aspnetcore-runtime AS runtime
14
15  WORKDIR /app
16
17  COPY --from=build /app/out .
18
19  ENTRYPOINT ["dotnet", "publisher_api.dll"]
```

Dockerfile hosted with ♡ by GitHub view raw

The first line declares the base image we're building our new image from; in this case, it contains the .NET Core SDK and is optimized for local development and debugging. Line 2 declares our working directory within the container (future COPY and RUN commands will execute within this directory).

Lines 5 and 6 copy our project file into the WORKDIR, and then execute a dotnet restore to pull down required dependencies. Lines 9 and 10 copy the rest of the files in our project directory into the WORKDIR, and then actually build the application.

On line 13, it looks like we start over, building an image from a different base image. What we're actually doing here is taking advantage of a Docker feature called multistage builds. The short of it is that we want to build and run our application in a container, but building the app requires a lot more overhead than running it. So what happens here is that when we declare another FROM statement in our Dockerfile, we're starting the next stage of our build from a clean image. But we still have access to all of the artifacts from the previous stage. So the final image will only contain the results of the last stage of the build.

So getting back to line 13, we‘re creating a clean image from Microsoft's .NET core 'runtime' image (it's leaner than the image we used for the first stage of the build). Line 15 declares our working directory in this image, and line 17 copies the binaries from our build image into this new image.

Line 19 declares the ENTRYPOINT for the container, which basically allows the container to run like an executable; when the container launches, it will launch the process declared here.

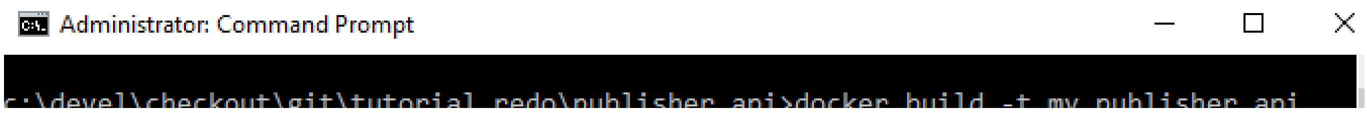
That's all we need to containerize our publisher_api application. What we've done is actually more complex than the most basic example we could create, because we are using a multistage build. Let's test it out!

Navigate to the publisher_api directory in a command prompt, and run the following command:

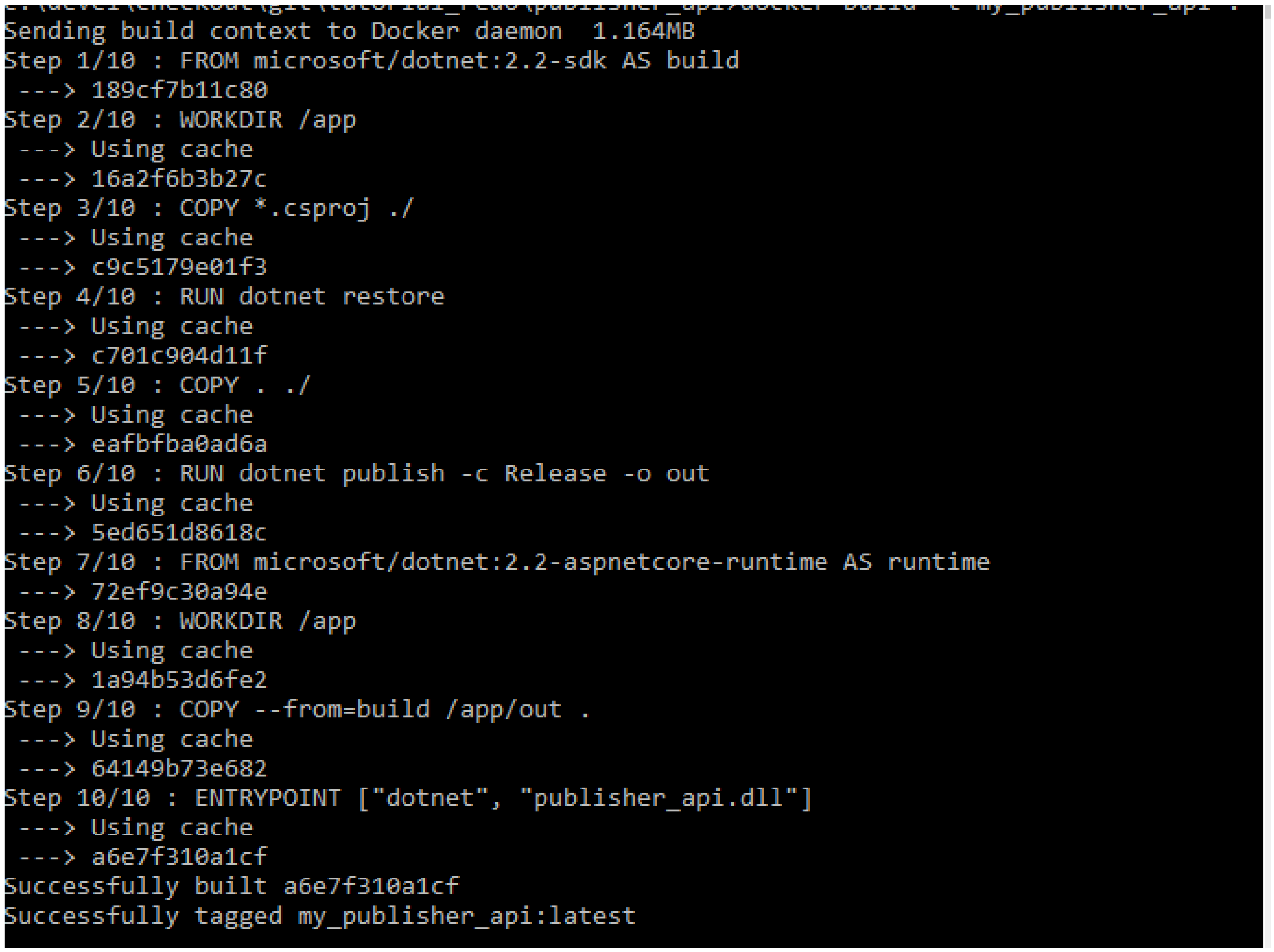
```
docker build -t my_publisher_api .
```

cmd hosted with by GitHub view raw

You should see output similar to this:

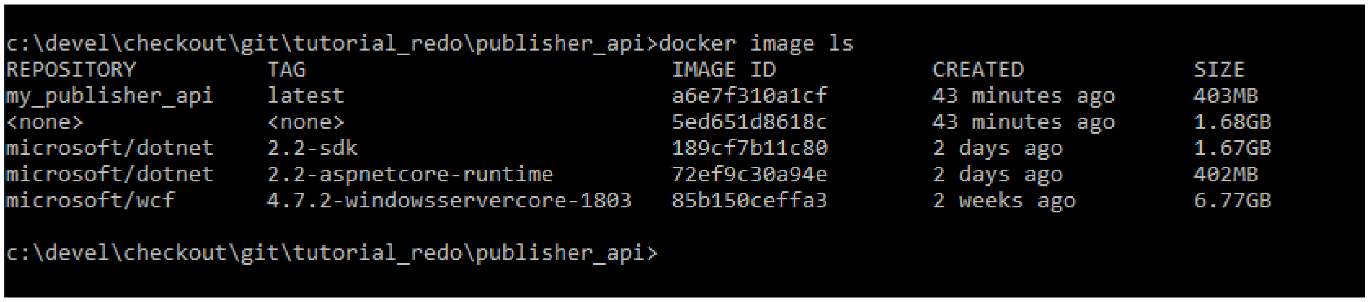


```
c:\devel\checkout\git\tutorial_redo\publisher_api>docker build -t my_publisher_api
```



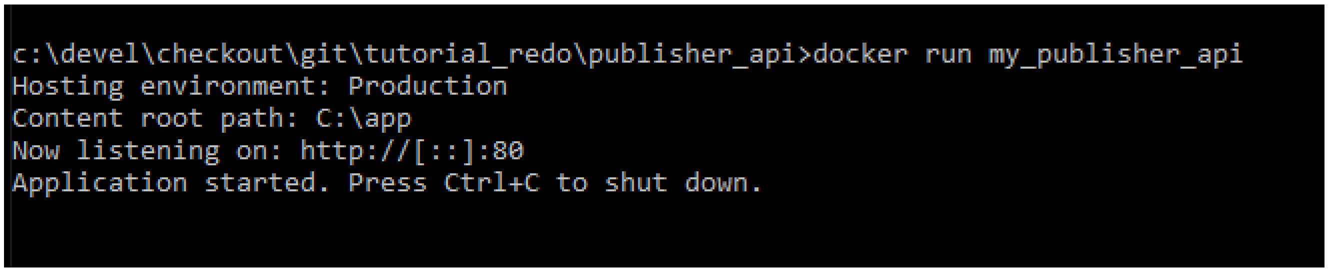
```
Sending build context to Docker daemon  1.164MB
Step 1/10 : FROM microsoft/dotnet:2.2-sdk AS build
 ---> 189cf7b11c80
Step 2/10 : WORKDIR /app
 ---> Using cache
 ---> 16a2f6b3b27c
Step 3/10 : COPY *.csproj ./
 ---> Using cache
 ---> c9c5179e01f3
Step 4/10 : RUN dotnet restore
 ---> Using cache
 ---> c701c904d11f
Step 5/10 : COPY . ./
 ---> Using cache
 ---> eafbfba0ad6a
Step 6/10 : RUN dotnet publish -c Release -o out
 ---> Using cache
 ---> 5ed651d8618c
Step 7/10 : FROM microsoft/dotnet:2.2-aspnetcore-runtime AS runtime
 ---> 72ef9c30a94e
Step 8/10 : WORKDIR /app
 ---> Using cache
 ---> 1a94b53d6fe2
Step 9/10 : COPY --from=build /app/out .
 ---> Using cache
 ---> 64149b73e682
Step 10/10 : ENTRYPOINT ["dotnet", "publisher_api.dll"]
 ---> Using cache
 ---> a6e7f310a1cf
Successfully built a6e7f310a1cf
Successfully tagged my_publisher_api:latest
```

You can use **docker image ls** to view your new image:



```
c:\devel\checkout\git\tutorial_redo\publisher_api>docker image ls
REPOSITORY          TAG                          IMAGE ID            CREATED             SIZE
my_publisher_api    latest                       a6e7f310a1cf        43 minutes ago      403MB
<none>              <none>                       5ed651d8618c        43 minutes ago      1.68GB
microsoft/dotnet    2.2-sdk                      189cf7b11c80        2 days ago          1.67GB
microsoft/dotnet    2.2-aspnetcore-runtime       72ef9c30a94e        2 days ago          402MB
microsoft/wcf       4.7.2-windowsservercore-1803 85b150ceffa3        2 weeks ago         6.77GB

c:\devel\checkout\git\tutorial_redo\publisher_api>
```

And **docker run my_publisher_api** to start the container:



```
c:\devel\checkout\git\tutorial_redo\publisher_api>docker run my_publisher_api
Hosting environment: Production
Content root path: C:\app
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
```

Next, we'll follow the same exact steps for the console app (adding a Dockerfile, building the image, and running the container.) The only difference in the new Dockerfile is the name of the DLL in the entrypoint:

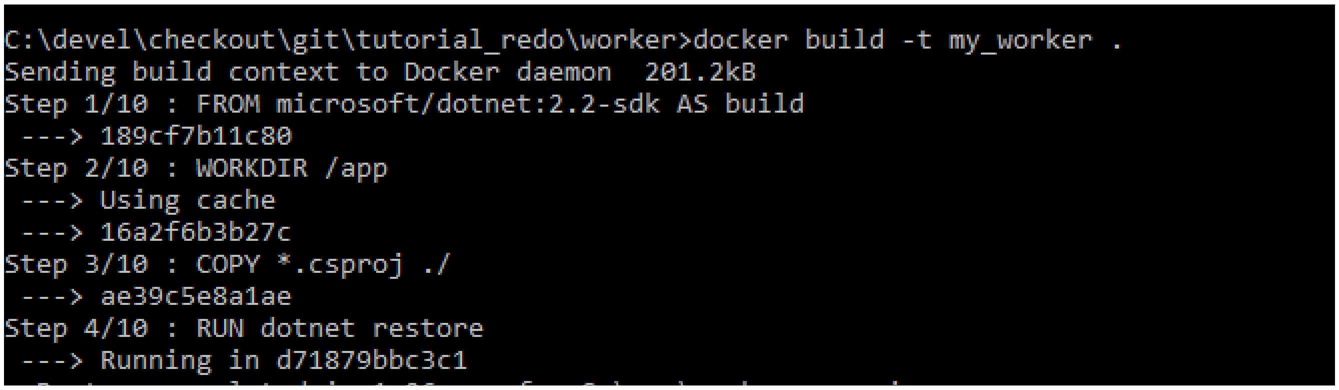
```
FROM microsoft/dotnet:2.2-sdk AS build
WORKDIR /app

# 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 microsoft/dotnet:2.2-aspnetcore-runtime AS runtime

WORKDIR /app

COPY --from=build /app/out .

ENTRYPOINT ["dotnet", "worker.dll"]
```

Dockerfile hosted with ♡ by GitHub view raw

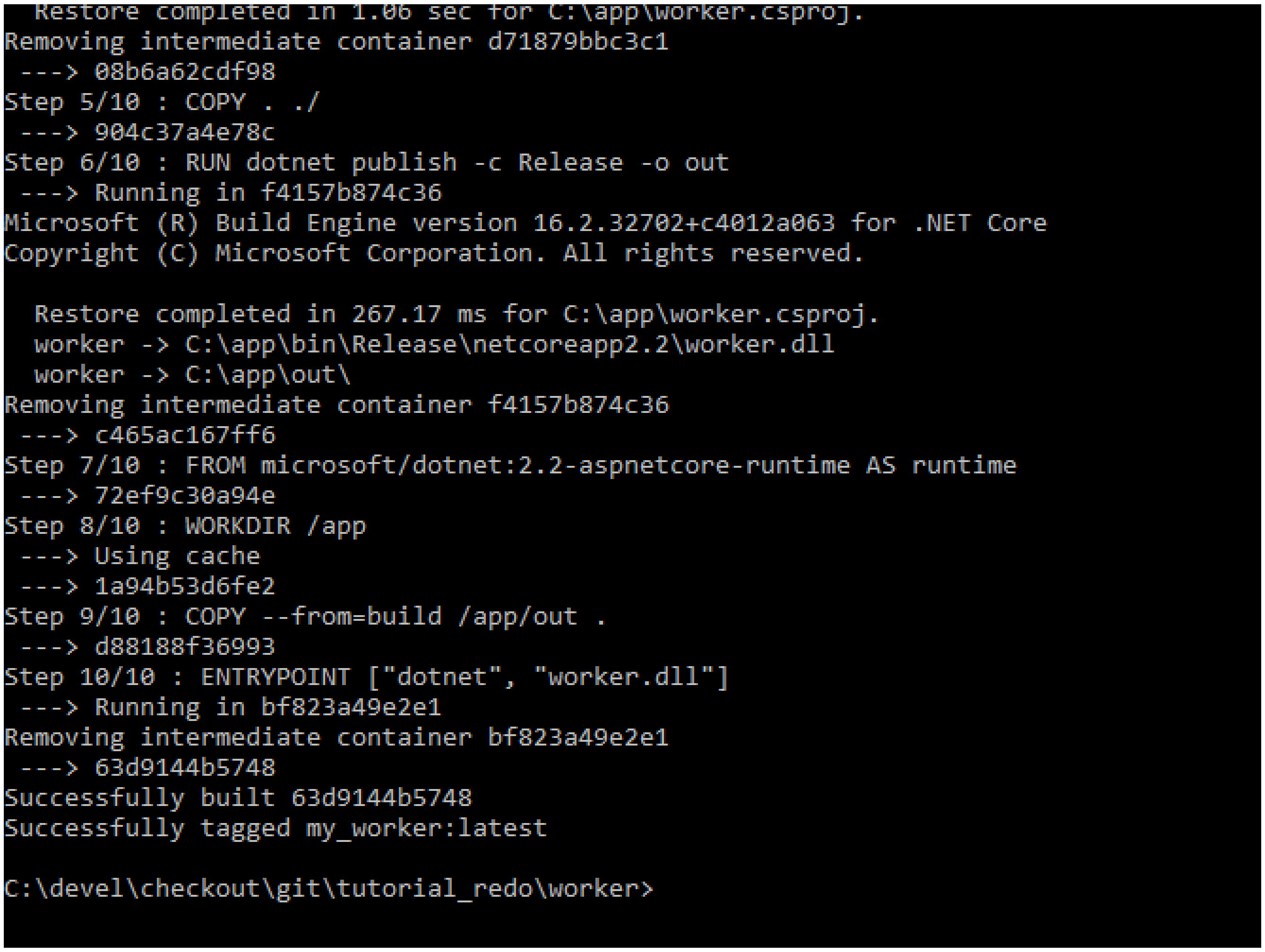
Navigate to the worker directory in a command prompt, and run the command to build the image:

```
docker build -t my_worker .
```

cmd hosted with ♡ by GitHub view raw

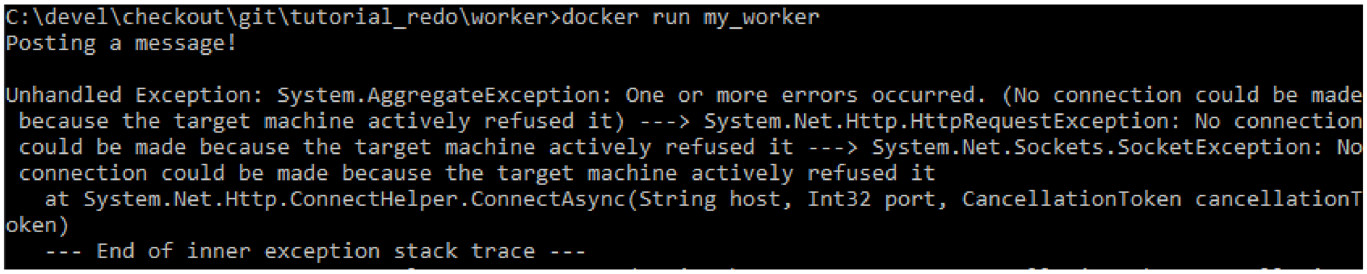


```
C:\devel\checkout\git\tutorial_redo\worker>docker build -t my_worker .
Sending build context to Docker daemon  201.2kB
Step 1/10 : FROM microsoft/dotnet:2.2-sdk AS build
 ---> 189cf7b11c80
Step 2/10 : WORKDIR /app
 ---> Using cache
 ---> 16a2f6b3b27c
Step 3/10 : COPY *.csproj ./
 ---> ae39c5e8a1ae
Step 4/10 : RUN dotnet restore
 ---> Running in d71879bbc3c1
```



```
  Restore completed in 1.06 sec for C:\app\worker.csproj.
Removing intermediate container d71879bbc3c1
 ---> 08b6a62cdf98
Step 5/10 : COPY . ./
 ---> 904c37a4e78c
Step 6/10 : RUN dotnet publish -c Release -o out
 ---> Running in f4157b874c36
Microsoft (R) Build Engine version 16.2.32702+c4012a063 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 267.17 ms for C:\app\worker.csproj.
  worker -> C:\app\bin\Release\netcoreapp2.2\worker.dll
  worker -> C:\app\out\
Removing intermediate container f4157b874c36
 ---> c465ac167ff6
Step 7/10 : FROM microsoft/dotnet:2.2-aspnetcore-runtime AS runtime
 ---> 72ef9c30a94e
Step 8/10 : WORKDIR /app
 ---> Using cache
 ---> 1a94b53d6fe2
Step 9/10 : COPY --from=build /app/out .
 ---> d88188f36993
Step 10/10 : ENTRYPOINT ["dotnet", "worker.dll"]
 ---> Running in bf823a49e2e1
Removing intermediate container bf823a49e2e1
 ---> 63d9144b5748
Successfully built 63d9144b5748
Successfully tagged my_worker:latest

C:\devel\checkout\git\tutorial_redo\worker>
```

And go ahead and test it with **docker run my_worker** to launch the container:



```
C:\devel\checkout\git\tutorial_redo\worker>docker run my_worker
Posting a message!

Unhandled Exception: System.AggregateException: One or more errors occurred. (No connection could be made
 because the target machine actively refused it) ---> System.Net.Http.HttpRequestException: No connection
 could be made because the target machine actively refused it ---> System.Net.Sockets.SocketException: No
 connection could be made because the target machine actively refused it
   at System.Net.Http.ConnectHelper.ConnectAsync(String host, Int32 port, CancellationToken cancellationT
oken)
   --- End of inner exception stack trace ---
```

Our container launched as — but there's an error. Our container ran as we expected, but since we're not running the publisher_api, our POST request fails. Luckily, there is a way we easily run both of our containers at the same time; using a tool is called Docker-compose.

Docker-compose is basically orchestration for containers. It allows you to define and run multi-container applications. Create a docker-compose.yml file in our root directory (at same level as our worker and publisher_api folders.) The contents of our implementation are below; copy them into your file and then we will get into the details.

```yaml
version: '3.4'

services:
  publisher_api:
    image: my_publisher_api:latest
    build:
      context: ./publisher_api
      dockerfile: Dockerfile

  worker:
    image: my_worker:latest
    depends_on:
      - "publisher_api"
    build:
      context: ./worker
      dockerfile: Dockerfile
```

docker-compose.yml hosted with ♡ by GitHub view raw

First we define the version of docker-compose that we'll use (3 is the latest major version.) Then we declare our services; a service is basically a single container, encapsulating an isolated piece of our application. We have two services - one for our webAPI and one for our console app.

The first service, which we've named publisher_api (the same as the project, although that's not required), is pretty straightforward. All we have to do is specify the build context, which is the directory where the Dockerfile for that project can be found.

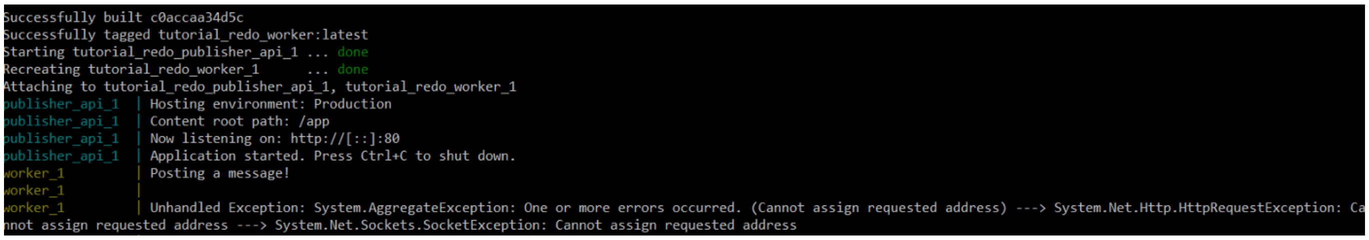
The other service is exactly the same, except we express the dependency between our services using *depends-on*. We don't want the worker to start up until the publisher_api is running.

Our docker is complete. However, if we run our app, we'll still see an error. Try it by executing the following command from the root directory of our project. The *build* flag specifies that we want to rebuild our containers to pick up the latest changes to our code or configuration.

```
docker-compose up --build
```

cmd hosted with ♡ by GitHub view raw

You should see output similar to the below image. Our containers are built, the publisher_api launches and start listening for HTTP requests, the worker launches and tries to post a message but fails with an error.



```
Successfully built c0accaa34d5c
Successfully tagged tutorial_redo_worker:latest
Starting tutorial_redo_publisher_api_1 ... done
Recreating tutorial_redo_worker_1      ... done
Attaching to tutorial_redo_publisher_api_1, tutorial_redo_worker_1
publisher_api_1  | Hosting environment: Production
publisher_api_1  | Content root path: /app
publisher_api_1  | Now listening on: http://[::]:80
publisher_api_1  | Application started. Press Ctrl+C to shut down.
worker_1         | Posting a message!
worker_1         |
worker_1         | Unhandled Exception: System.AggregateException: One or more errors occurred. (Cannot assign requested address) ---> System.Net.Http.HttpRequestException: Ca
nnot assign requested address ---> System.Net.Sockets.SocketException: Cannot assign requested address
```

Currently, the worker sends a POST to http://localhost:5001/api/Values, and this worked because we were running both the webAPI and the worker on our local machine. But now that they are in separate containers, we need a way for the worker to discover and communicate with the publisher_api.

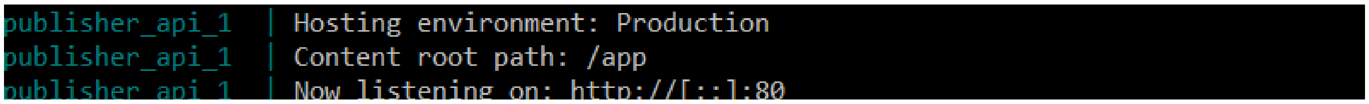
Since we're using docker-compose to launch our application, by default a single network is created for our application. Each container for a service joins the default network and is both *reachable* by other containers on that network, and *discoverable* by them at a hostname identical to the container name.

> Understanding docker-compose networking, we need to modify the URL the worker is trying to reach. Replace localhost with the desired container name (publisher_api) and change the port to 80 (5001 was where Visual Studio Code hosted the debugger).

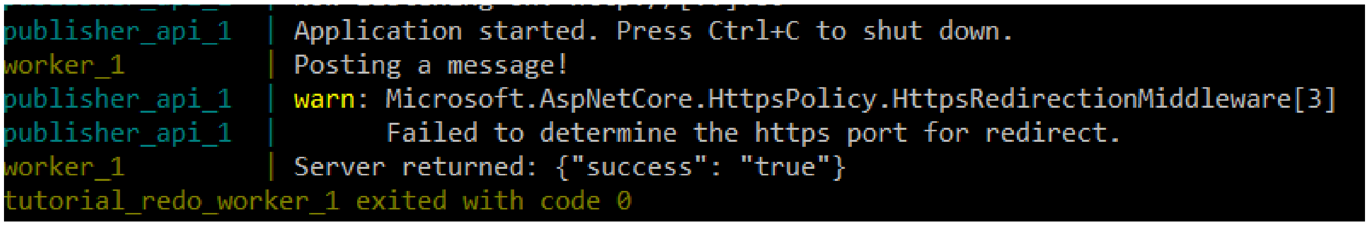
```
var result = await client.PostAsync("http://publisher_api:80/api/Values", content);
```

Program.cs hosted with ♡ by GitHub view raw

Launch the app again using the same command as before (make sure you use the --build flag so our code changes are picked up.) This time, the worker should successfully reach the publisher_api, and the response (success:true) should be logged to the console.



```
publisher_api_1  | Hosting environment: Production
publisher_api_1  | Content root path: /app
publisher_api_1  | Now listening on: http://[::]:80
```



```
publisher_api_1  | Application started. Press Ctrl+C to shut down.
worker_1         | Posting a message!
publisher_api_1  | warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
publisher_api_1  |       Failed to determine the https port for redirect.
worker_1         | Server returned: {"success": "true"}
tutorial_redo_worker_1 exited with code 0
```

. . .

To recap, we took our existing .NET core application, containerized each of the two components, and then used docker-compose to launch our multi-container application. Stay tuned for Part 3 of this series, where we'll use RabbitMQ to decouple our client and server and experiment a bit more with docker-compose.

Interested in joining our team at Trimble Maps? Click here!

Docker   Docker Compose   Web Development   Dotnet Core   Programming

About   Help   Legal