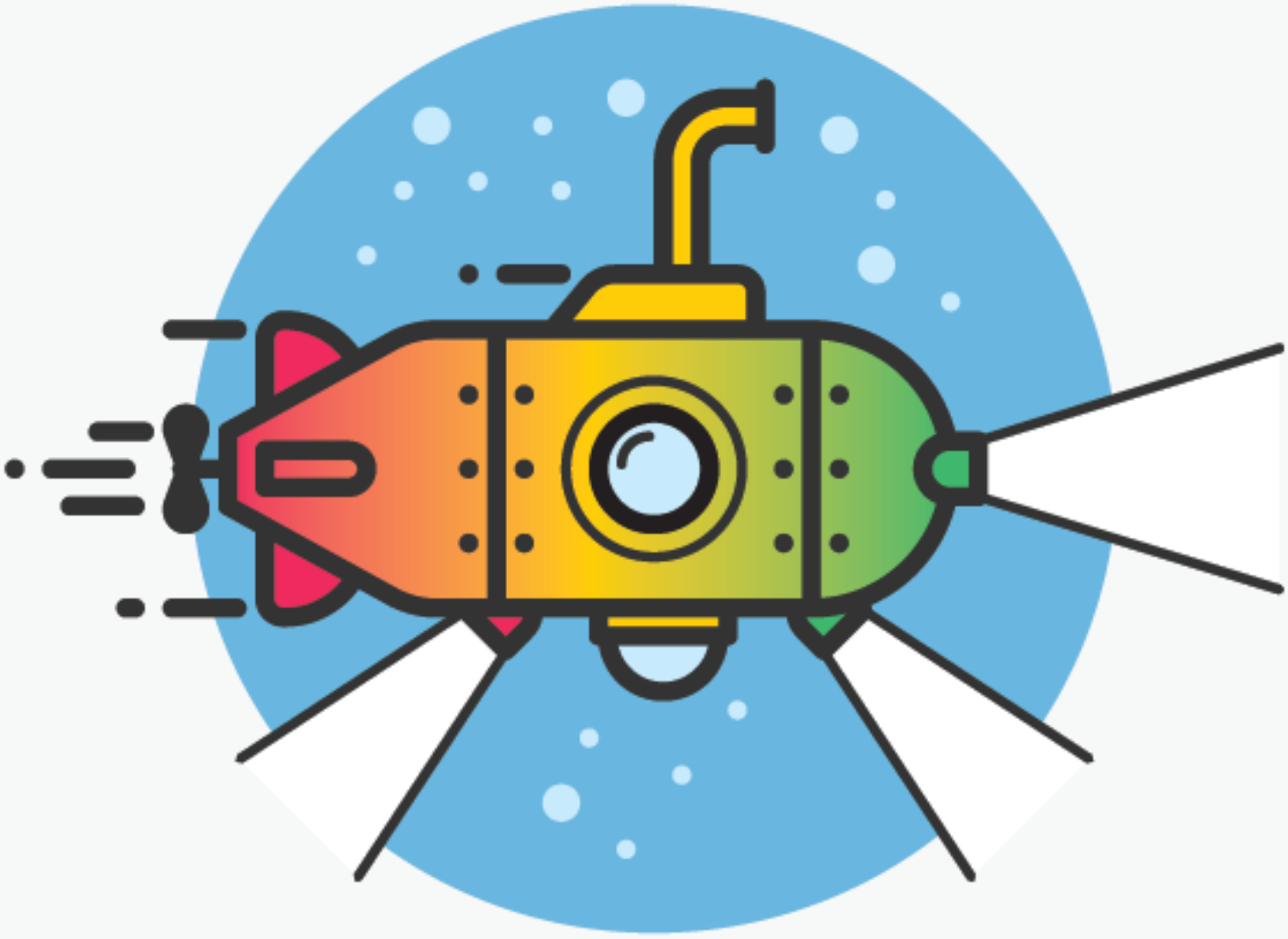
LIMITED
BUNDLE EDITION

Marinko Spasojevic
Vladimir Pecanac

DOCKERIZING ASP.NET CORE APPLICATION

Changing the Way You Plan, Build and Deliver Applications



Made with ❤ by: CodeMaze

1 Introduction

Docker is one of the greatest innovations that happened in the last few years. It has opened new horizons in software development and it spun off many innovative solutions and projects. Docker images and containers are rapidly becoming **THE way** to do software development.

Everything is dockerized. You can find an **image for almost everything**. If you can't, well, make your own and push it. Now, there is an image for that too.

We won't go too deep into the Docker itself since it already has fantastic documentation, and there are plenty of in-depth Docker guides throughout the internet. There are some concepts you'll need to understand to be able to follow along since it gets progressively harder.

1.1 So, What's the Point?

Our main goal will be to **show off the tremendous power of Docker and ASP.NET Core** when combined.

We love the hands-on approach, so we will use concrete examples to demonstrate the concepts and explain the most important aspects of these technologies.

We will use the example project from the book, so you have nothing to fear. Once we finish, we'll have a nice, dockerized application, ready to be deployed to any server.

1.2 The Technology Stack

The technologies you'll need to follow along with the complete series:

- Windows 10 or any newer Linux OS (We're going to use Win10 with Linux containers through the series)
- Docker Desktop for Windows

- Visual Studio 2022 or any other IDE of your choice
- Powershell (recommended)

We recommend using this stack since that is what we've used and we can vouch for it to work for you too.

Since Docker is a complex set of technologies bound together, sometimes you can stumble upon some OS-specific quirks that can make you lose a few hours of your life.

We've encountered and solved a few of those so you don't have to.

1.2 Standing Out From The Crowd

Besides being a phenomenal technology and the one that can potentially improve your career by a large margin. Learning Docker moves you one step closer to a DevOps Engineer title and not only helps you personally, but your employees will be thrilled that you learned it on your own.

According to Glassdoor, the DevOps Engineer job has been hitting the top of the tech job lists for the three consecutive years right after Data Scientist.

Here are the reports for 2017., 2018., 2019., 2020 and yes, even 2021. respectively:

- https://www.glassdoor.com/List/Best-Jobs-in-America-2017-LST_KQ0,25.htm
- https://www.glassdoor.com/List/Best-Jobs-in-America-2018-LST_KQ0,25.htm
- https://www.glassdoor.com/List/Best-Jobs-in-America-2019-LST_KQ0,25.htm
- https://www.glassdoor.com/List/Best-Jobs-in-America-2020-LST_KQ0,25.htm

- https://www.glassdoor.com/List/Best-Jobs-in-America-LST_KQ0,20.htm

The list has taken **job satisfaction** and **median base salary** into account. So, not only will you get your six-figure job, but you will most probably enjoy your position as well.

As you can see, businesses look for the best ways to scale up, and the DevOps engineer is the right choice for the best companies out there like Google, Amazon, Apple and so on.

So what are we waiting for, let’s take the first step towards a six-figure dream job!

2 Preparing the Application

Before we delve deeper into the dockerization of our Web API project, we need to make some changes.

For starters, we need to modify our launchSettings.json file to be able to work with Docker later on.

2.1 Modifying the launchSetting.json File

The **launchSettings.json** file is a file that is used by IDEs only. We can use it while developing our application and we should change it to be able to expose the application port from the docker container to the local environment.

If this seems confusing, just go with it now, it will be much clearer later on. The important thing to know is that we can't use the localhost inside the docker container since the localhost is different in the container and outside of it.

Instead of localhost, we will use the IP address 0.0.0.0. We're also going to remove https address for now because it needs some additional work with certificates. We'll add it back later. Let's modify our file:

```
{
      "profiles": {
            "CompanyEmployees": {
                  "commandName": "Project",
                  "dotnetRunMessages": true,
                  "launchBrowser": false,
                  "launchUrl": "weatherforecast",
                  "applicationUrl": "http://0.0.0.0:5000",
                  "environmentVariables": {
                        "ASPNETCORE_ENVIRONMENT": "Development"
                  }
            }
      }
}
```

Okay, that will do it for now.

2.2 Switching to an In-Memory Database

To be able to dockerize our application and make it \work without relying on external resources (database) we are going to make some changes to the configuration. We are going to re-introduce MSSQL later in the series, but for now, we are going to rely on an in-memory database to store our data.

So, let's modify our **ServiceExtensions.cs** class:

```
public static void ConfigureSqlContext(this IServiceCollection services,
IConfiguration configuration) =>
services.AddDbContext<RepositoryContext>(o =>

o.UseInMemoryDatabase("CompanyEmployees"));
```

The **UseInMemoryDatabase** method is part of the Microsoft.EntityFrameworkCore.InMemory NuGet package, so make sure to import it to the **CompanyEmployees** project!

We successfully removed the reference to the MSSQL database and added an in-memory database in a single line of code.

Don't worry about the hanging connection string in the **appsettings.json** file, we don't need to remove it since we are going to use it in the next part.

2.3 Adding a Test Project and a Unit Test

For testing purposes, we're going to use the xUnit tool and Moq for mocking our data.

We haven't added tests to the original project on purpose, but since we want to demonstrate how to run tests with Docker, we will create the most basic one as an example.

The easiest way to add a testing project in ASP.NET Core is with the dotnet CLI. Navigate to the root of the solution and type:

```
dotnet new xunit -o Tests
```

This will create the project using the xUnit template and restore the needed packages.

After that we can add the project to the solution:

```
dotnet sln CompanyEmployees.sln add .\Tests\Tests.csproj
```

And because we need to use the Moq library, we are going to navigate to Tests project and add it too:

```
cd Tests

dotnet add package Moq

dotnet restore
```

And finally, reference the projects we need to write unit tests:

```
dotnet add reference ..\Contracts\Contracts.csproj
```

That's it for the setup part.

Now we can easily add our first unit test to the project.

First, rename the default **UnitTests1.cs** file to something more appropriate like **CompanyRepositoryTests.cs** in which we can test our company repository logic.

Now, let's add our unit test:

```
[Fact]
public void
GetAllCompaniesAsync_ReturnsListOfCompanies_WithASingleCompany()
{
      // Arrange
      var mockRepo = new Mock<ICompanyRepository>();
      mockRepo.Setup(repo => (repo.GetAllCompaniesAsync(false)))
            .Returns(Task.FromResult(GetCompanies()));

      // Act
      var result = mockRepo.Object.GetAllCompaniesAsync(false)
            .GetAwaiter()
```

```
            .GetResult()
            .ToList();

      // Assert
      Assert.IsType<List<Company>>(result);
      Assert.Single(result);
}

public IEnumerable<Company> GetCompanies()
{
      return new List<Company>
      {
            new Company
            {
                  Id = Guid.NewGuid(),
                  Name = "Test Company",
                  Country = "United States",
                  Address = "908  Woodrow Way"
            }
      };
}
```

This is a simple demo test to check if the **GetAllCompaniesAsync** method returns a list of **Company** objects and if it returns exactly one company as we defined in the **GetCompanies** helper method.

Now if you run **dotnet test** you should get something like this:

```
Microsoft (R) Test Execution Command Line Tool Version 16.8.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     1, Skipped:     0, Total:     1, Duration: 71 ms
```

Excellent!

We have completed the preparation part of the project. We have a working ASP.NET Core application, together with interactive documentation and some unit tests.

Everything is set for the next step, which is why we did all this preparation. Now it’s time to dockerize our ASP.NET Core application.

To check if everything works alright, run **dotnet build** and **dotnet test**. This is how we do it usually during the development phase. This time around we'll use docker images and containers to do the same thing and bring our software development skills to another level.

3 The Docker CLI Through Examples

Now that we've prepared our ASP.NET Core application, we are going to learn how to install and use Docker on Windows 10, the reasons behind using Docker, and some useful Docker CLI commands. **Understanding how the Docker CLI works is crucial**, and you'll most definitely have a hard time proceeding further without getting a grasp of the basic commands Docker offers through its powerful CLI.

3.1 Short Intro to Docker on Windows

Docker Desktop for Windows, as well as some basic instructions, can be found in the docker store. The installation is pretty straightforward and you can easily test if it worked with a quick **docker version** command (recommended 20.10.2+ but older should work fine too). We recommend using **PowerShell** for all the docker commands we use since it's more verbose than plain old Command Prompt.

Docker relies on Windows 10 Hyper-V to instantiate the virtual machines which we need as a base for our containers. If you haven't enabled it, you'll need to do it to proceed. In most cases it is enabled by default, but sometimes you need to enter the BIOS to enable it.

But before that, let's go through some **very basic concepts** regarding Docker and ASP.NET Core.

As you know, **Docker images consist of a bunch of layers** which attribute to the final environment inside that image. There are a lot of **predefined images** we can pull from the Docker Hub, but we can also make our own images by adding our layers on top of existing ones to create an image that suits our needs.

On Windows 10, Docker lets you choose between Linux and Windows containers, which is a very nice feature. In this series, we are going to stick with the Linux containers, but the principles apply both to Windows or Linux containers. The syntax and underlying architecture (eg. paths) may vary a bit.

3.2 Why Docker?

Before we dive completely into Dockerfiles, compositions and advanced features of Docker, let's play around a bit with the Docker CLI.

Understanding how the Docker CLI works will help you later when we move on to Dockerfiles.

So, we have an application to play with now, it builds, the tests are passing and we can run it on a local machine.

So, how do we start working with Docker?

When we started learning Docker, **the main problem we had was to grasp the big picture**. It looked like just some complicated tool that produces an overhead in our projects. We've been already familiar with continuous integration and delivery, and Docker didn't seem to fit into our CI/CD pipelines.

So we'll try to explain why we use Docker and how to use the Docker CLI with our ASP.NET Core app.

3.3 Scenario One: Using Docker to Keep Local Environments Clean

Let's say you have a Java-oriented friend and you want to explain to him why .NET Core kicks ass. We'll call him Mike. Mike is quite a stubborn lad and doesn't want to install all that Microsoft mumbo-jumbo on his machine, which is perfectly understandable.

.NET SDK and Visual Studio can take time and disk space to install and Mike doesn't want that. But, Mike has heard how awesome Docker is and he even installed it at one time on his machine some time ago.

So how do we help Mike start with .NET Core as soon as possible?

First, give him some ASP.NET Core application. As it happens we have one at our hands ☺

Next, **navigate to the root** of the project and build the project by typing (works only in PowerShell):

```
docker run --rm -it -w /home/app/CompanyEmployees/ -v ${PWD}:/home/app
mcr.microsoft.com/dotnet/sdk:6.0 dotnet build
```

Followed by:

```
docker run --rm -it -w /home/app/CompanyEmployees/ -p 8080:5000 -v
${PWD}:/home/app mcr.microsoft.com/dotnet/sdk:6.0 dotnet run
```

And just like that we have built and run our application on Mike's machine even though he doesn't have any Microsoft dependencies installed. (If this command gives you trouble it's probably because you've pasted it. Completely normal. Try typing it instead.)

Now you can run the application on your local machine by typing **http://localhost:8080/swagger/** in your browser and you'll get the familiar Swagger docs page. You will get prompted by the browser about the missing certificate, but we'll deal with that later. For now, you can proceed to the page.

Isn't that just awesome?

But those commands look really complicated. Let's break them down.

3.3.1 A Bit of Clarification

These commands look scary when you see them for the first time but they are not that complicated once you get familiar with the syntax.

The actual command is **docker run**, followed by a few options, and then the image name, which is the name of the one we've built in the last step.

Let's explain what each option stands for:

- **--rm**: Automatically removes the container once it stops. If you exclude this command you'll see the Docker container is still attached by typing **docker ps -a**
- **-it**: This command is comprised of two parts, -i which means **interactive mode**, and -t which enables the **pseudo-terminal.** These commands can be used separately to get the same result, but they are usually used together. The -it command gives us the means to see and interact with our application through the terminal.
- **-w /home/app/CompanyEmployees/**: Sets the current working directory for the container when it gets instantiated. In our case that will be **/home/app/CompanyEmployees/** where the main app resides, so we can run our commands from there
- **-p**: Maps the external port (left) to the internal port (right) of the Docker container. In our case, this means that we can access our application through the **http://localhost:8080** on our local machine, although it runs on a port 5000 inside the Docker container
- **-v ${PWD}:/home/app**: This one is a bit tricky but once you get it, it can do wonders. What this actually means is that it maps our current directory to the **/home/app** directory inside the container. This enables us to change the code on our local system

and build it in the container. You can learn more about Docker volumes here.

The options are followed by the image name, and finally the command we want to run inside the instantiated container.

So, **what does this mean for Mike** the Java programmer?

This means Mike can safely change the code on his machine using any IDE, and then run the container to see the results of his work without having had to install any of the dependencies on his machine.

You can now take a break and watch Mike and enjoy his conversion to C#.

3.4 Scenario Two: Using Docker to Test Your App in a Clean Environment

Mike is now a full-fledged C# programmer. He tried it and he loved it so much that he decided to install the .NET SDK on his own machine.

Since Mike is a web developer in heart, he wants to make a small ASP.NET Core application to learn how web programming works in .NET.

But since Mike is a good developer and knows about the "It works on my machine" phenomenon, he wants to run the app in a clean environment to test it out and make sure the app works every time.

What can Mike do in this situation?

He can, of course, build his application on some of the many available CI/CD tools, and then deploy the application on the staging machine. Or he can publish directly using the Visual Studio publish tools. And he needs to do that both for Windows and Linux.

But there might be a faster and easier way he can do that: **spin up a local docker container optimized for the ASP.NET Core runtime.**

How can Mike do that?

We learned some commands in the previous example, so let's use them here too. We won't rely on **mcr.microsoft.com/dotnet/sdk:6.0** image since Mike builds his application locally.

What he wants to use is the **mcr.microsoft.com/dotnet/aspnet:6.0** image which is optimized for runtime.

But first, he needs to publish his app and make a small Dockerfile in the CompanyEmployees folder that will instruct Docker to copy the contents of the publish folder to the container.

So, Mike should navigate to **/CompanyEmployees** and then type **dotnet clean** and then **dotnet publish**. This will publish the application into the **./bin/Debug/net6.0/publish** directory.

After that, he needs to add the Dockerfile to the project root:

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /home/app
COPY bin/Debug/net6.0/publish .
ENTRYPOINT ["dotnet", "CompanyEmployees.dll"]
```

Build his own image using that Dockerfile (make sure not to forget to put the dot at the end of the command): **docker build -t companyemployees .**

And then simply run the image with: **docker run -p 8080:80 companyemployees**

That's it, Mike has run the application on a clean environment and now he can sleep peacefully because he knows that his app can run on more than just his machine.

3.5 Scenario Three: Persisting the Changes and Cross-Platform Development

Mike has been working now for a few weeks on his ASP.NET Core app. But he has already planned a nice trip with his family and he won't be able to continue working on his desktop machine. But he brings his Mac wherever he goes.

Up until now, Mike has been working locally and using the local Docker environment.

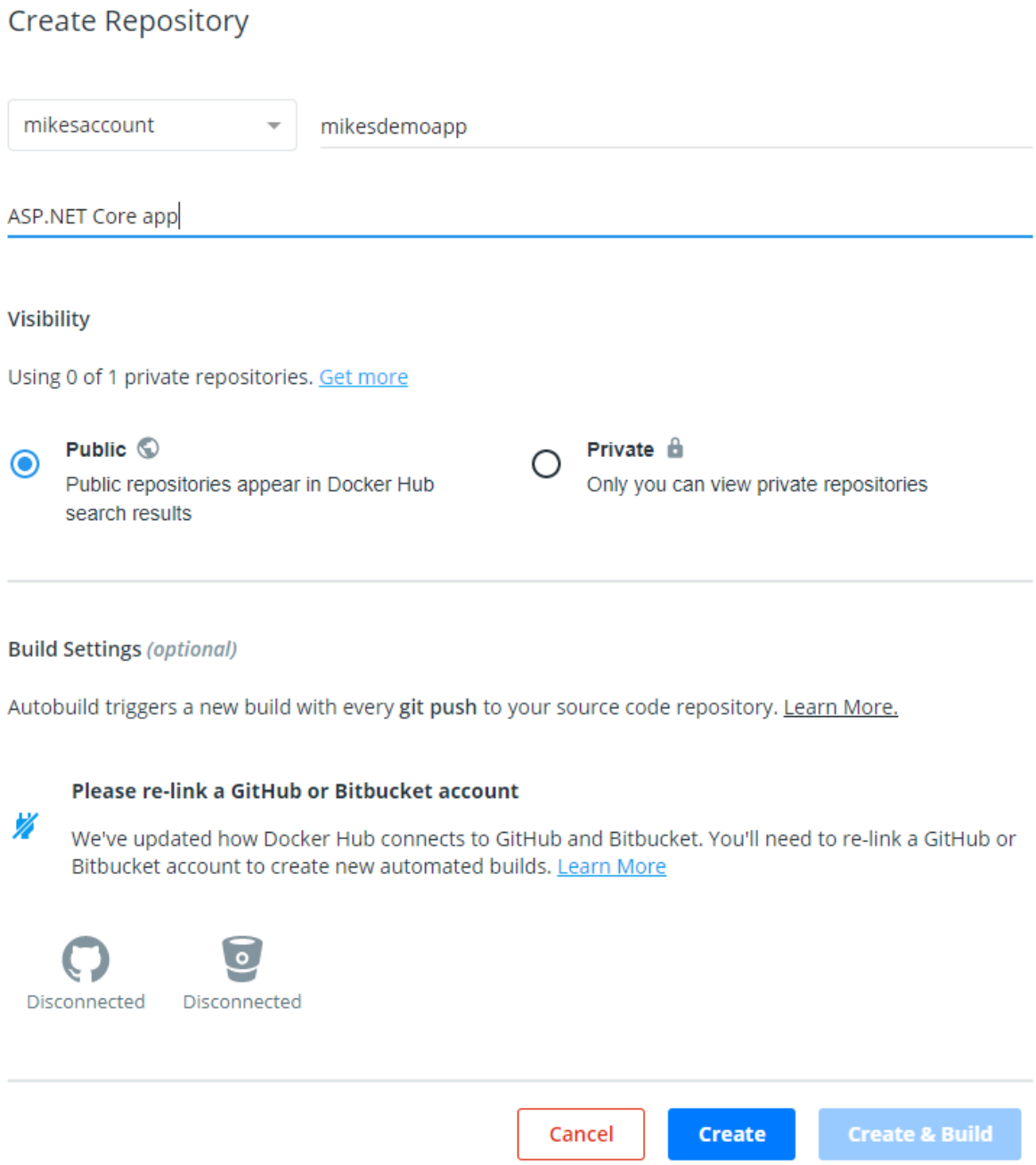
So, to continue working in the evening, after his long walks through the woods, he needs a mechanism to persist his work and bring it with him.

So how can Mike do this?

3.6 Creating the Image and Pushing It to Docker Hub

First, he needs to create an account on Docker Hub. It's an easy process and in a few minutes, Mike has a Docker Hub account.

Then he can create his own repository and name it MikesDemoApp:



To be able to work on the project, he needs to use **mcr.microsoft.com/dotnet/sdk:6.0** image, optimized to build and run ASP.NET Core applications.

So, he needs to modify the Dockerfile a bit:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0
WORKDIR /home/app
```

```
COPY . .
```

And build his own image and tag it **mikesaccount/mikesdemoapp**:

```
docker build -t mikesaccount/mikesdemoapp .
```

And then he can log in to his Docker Hub account and push the image through the Docker CLI:

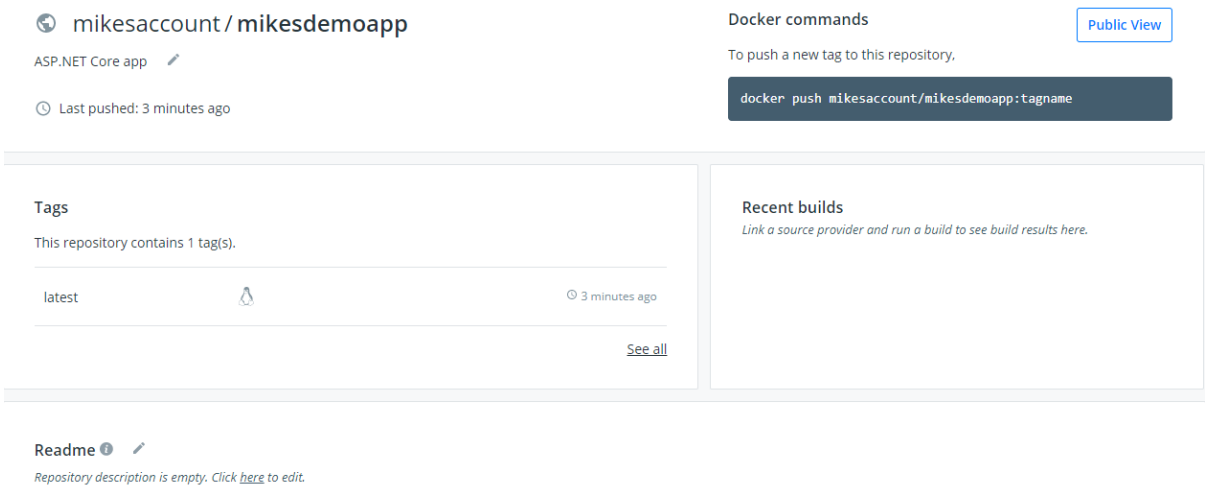
```
docker login -u mikesaccount -p SomeCoolPassword

docker push mikesaccount/mikesdemoapp
```

The result in the console should look something like this:

```
The push refers to repository [docker.io/mikesaccount/mikesdemoapp]
281394cc7361: Layer already exists
9e9c7cce0649: Layer already exists
530725a7d336: Pushed
b196c7c99e82: Layer already exists
f9d40f369825: Pushed
9c2a08e1e47a: Pushed
e1df5dc88d2c: Pushed
latest: digest: sha256:8024041df987c7a3051e68224a2c7903e688dcceb066fa27e8e2db6b9ffafd0e size: 1793
```

And now Mike can navigate to the "Tags" tab on the Docker Hub repository to see his image:



Now Mike can safely pull the image to his Mac and work some more on his app before he goes to sleep.

```
docker pull mikesaccount/mikesdemoapp
```

These scenarios are just a few of the unlimited number of possible Docker usages. You have seen how powerful the Docker CLI can be but we've just scratched the surface.

Docker's full potential can only be seen when you deploy multiple microservices and connect them.

You can dockerize everything. Applications, databases, monitoring tools, CI tools… Name it and there is an image of it.

Now that we are familiar with the Docker CLI, let's see what we can do with Dockerfiles.

4 Dockerizing ASP.NET Core App

Now we are going to focus on dockerizing our ASP.NET Core application with Dockerfiles, and understanding how Dockerfile syntax works. We are also going to spend a lot of effort into optimizing our images to achieve the best results.

4.1 Creating a Dockerfile

The first step we need to do is to navigate to the root folder of our solution and create a new Dockerfile. Dockerfiles are the declarative inputs that we can use to tell Docker what to do with our application.

Let's add some actions to our Dockerfile:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0
WORKDIR /home/app
COPY . .
RUN dotnet restore
RUN dotnet publish ./CompanyEmployees/CompanyEmployees.csproj -o /publish/
WORKDIR /publish
ENV ASPNETCORE_URLS=http://+:5000
ENTRYPOINT ["dotnet", "CompanyEmployees.dll"]
```

4.1.1 A Bit of Explanation

So what have we done exactly?

- **FROM mcr.microsoft.com/dotnet/sdk:6.0**: Every Dockerfile starts with the FROM command, which initializes a new build stage and sets the base image that we are going to build upon. In this case, that image is mcr.microsoft.com/dotnet/sdk:6.0 image because we want to build our ASP.NET Core application with SDK version 6.0. Microsoft has a plethora of other useful images on their Docker Hub profile, so go check them out.

- **WORKDIR /home/app**: The WORKDIR command simply sets the current working directory inside our image. In this case, that is the /home/app folder.
- **COPY . .**: The COPY command is pretty straightforward too. In this case, it copies all the files from the local system to the current working directory of the image. Since we don't need to copy all the files to build the project, we're going to use a .dockerignore file to which the COPY command will look up when it starts copying the files.
- **RUN dotnet restore**: The RUN command runs any command in a new layer and commits it to the base image. Concretely, in this step, we are restoring the packages for our solution, as we would if we run it locally, but this time it's happening inside the image.
- **RUN dotnet publish ./CompanyEmployees/CompanyEmployees.csproj -o /publish/**: After we restore the packages, the next step is to publish our application. Since the CompanyEmployees project is our main app, we are going to publish it to the **/publish** folder inside our image.
- **WORKDIR /publish:** We are switching our current directory to **/publish**
- **ENV ASPNETCORE_URLS="http://+:5000"**: Since launchSettings.json is a file that's just used by our local IDE, it won't make it to the publish folder. This means we need to set the application URL manually. Once again, we can't use localhost or we won't be able to bind the port from the docker container to the local environment.
- **ENTRYPOINT ["dotnet", "CompanyEmployees.dll"]**: The ENTRYPOINT command allows us to configure the container to run

as an executable. In this case, after the project has been published and we run the image, a container will spin up by firing **dotnet CompanyEmployees.dll** command which will start our application.

4.2 Creating a .dockerignore File

Since we are copying our files to the docker image on every build, we should create a **.dockerignore** file and select which files and folders we don't want to copy every time. The advantages of using a .dockerignore include faster image build, improving cache performance, and avoiding potential conflicts when building an application.

For example, we don't want to copy our Dockerfile.

Why is that?

Because besides not being important to the build process, that would mean our **COPY** step will trigger every time we change the Dockerfile, and that's not something we want. So you want to put all the files that you don't want to trigger a build into a **.dockerignore** file.

For starters, we are going put these files and folders in our **.dockerignore** file:

```
**/bin/
**/obj/
**/global.json
**/Dockerfile*
**/.dockerignore*
**/*.user
**/.vs/
```

4.3 Building the image

Once we configured our Dockerfile and .dockerignore, we can start the build and tag our image:

The first time you run the docker build, it will take a while, since Docker is fetching the base image. Give it some time. After the first build, every following build will use that image from the local machine.

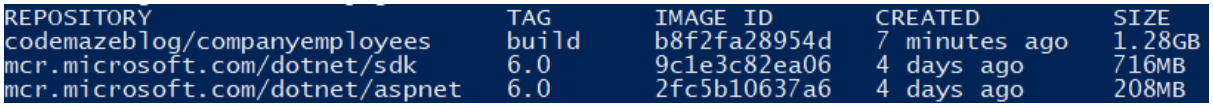
First build is fun and you get to see every step of our Dockerfile resolving in real-time.

After the first build, if you don't change the project files, all the steps will be cached and you'll get something like this:

As you can see every step is cached, and there is no need to rebuild the image. If you want you can force the rebuild with the –no-cache flag.

```
docker build --no-cache -t codemazeblog/companyemployees:build .
```

If you run the **docker images** command, now you can find our image on the list (we've cleaned it a bit with **docker rmi** command):



Okay, now we're ready to run our application.

Running the Image

Finally, to run the application you can spin up the container by typing:

```
docker run --rm -it -p 8080:5000 codemazeblog/companyemployees:build
```

This should run our application as we expect it to so far.

But, take a quick look at the image size. It's the biggest so far. That's because we copied, restored and published our code in the same image.

Let's try to optimize the process.

4.4 Optimizing the Dockerfile

As we have learned by now, Docker images consist of different layers, and we've seen the Docker build process. It goes step by step and executes the steps we defined. Once the step is executed, it gets cached, and if there are no changes related to that step it gets pulled from the image cache next time we run the build.

So, knowing all this can you guess what we could have done better in our Dockerfile?

Well, let's have a look at that third step once again: **COPY . .**, and then we do **RUN dotnet restore**.

What this means is that whenever we make a change to the source code, we need to run **dotnet restore** since that will break the cache.

So instead of copying all the files, we can just copy the project and solution files, do the **dotnet restore**, and then copy the rest of the files:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0
WORKDIR /home/app
COPY ./CompanyEmployees/CompanyEmployees.csproj ./CompanyEmployees/
COPY ./Contracts/Contracts.csproj ./Contracts/
COPY ./Repository/Repository.csproj ./Repository/
COPY ./Entities/Entities.csproj ./Entities/
COPY ./LoggerService/LoggerService.csproj ./LoggerService/
COPY ./Tests/Tests.csproj ./Tests/
COPY ./CompanyEmployees.sln .
RUN dotnet restore
COPY . .
RUN dotnet publish ./CompanyEmployees/CompanyEmployees.csproj -o /publish/
WORKDIR /publish
ENV ASPNETCORE_URLS=http://+:5000
ENTRYPOINT ["dotnet", "CompanyEmployees.dll"]
```

There, now the **dotnet restore** won't trigger whenever we change something in our source code. We'll only need to re-publish the assemblies.

But why do we copy every project and solution file manually?

That's because something like **COPY ./**/*.csproj ./** wouldn't work. It won't recreate the folder structure. So you would just get the flat structure with all the project and solution files. And that's no good.

4.5 Optimizing Even Further

We can push the optimization even further.

The downside of the current way of creating the Dockerfile is that we need to manually copy each solution and project file. That means we need to change the Dockerfile whenever our solution structure changes, and it just looks bad. It's a lot of work.

So let's see how we can remedy that.

Take a look at the next Dockerfile:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0
WORKDIR /home/app
COPY ./*.sln ./
COPY ./*/*.csproj ./
RUN for file in $(ls *.csproj); do mkdir -p ./${file%.*}/ && mv $file
./${file%.*}/; done
RUN dotnet restore
COPY . .
RUN dotnet publish ./CompanyEmployees/CompanyEmployees.csproj -o /publish/
WORKDIR /publish
ENV ASPNETCORE_URLS=https://+:5001;http://+:5000
ENTRYPOINT ["dotnet", "CompanyEmployees.dll"]
```

So, what happened here?

In short, we used some clever syntax as a workaround for the **COPY** command to recreate the folder structure we need. The command uses the project names to create folders for the files to be copied in.

Now we don't need to change the Dockerfile even if the add more projects to our solution or change project names.

Clear and magical.

4.6 Just Run and Test Boys, Run, and Test

There is one more tiny little thing we need to do. We need to run our unit test to see if the project is even worth publishing!

So let's add that to our Dockerfile as well:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0
WORKDIR /home/app
COPY ./*.sln ./
COPY ./*/*.csproj ./
RUN for file in $(ls *.csproj); do mkdir -p ./${file%.*}/ && mv $file
./${file%.*}/; done
RUN dotnet restore
COPY . .
RUN dotnet test ./Tests/Tests.csproj
RUN dotnet publish ./CompanyEmployees/CompanyEmployees.csproj -o /publish/
WORKDIR /publish
ENV ASPNETCORE_URLS=https://+:5001;http://+:5000
ENTRYPOINT ["dotnet", "CompanyEmployees.dll"]
```

That's it. Rebuild the image again and check out the result.

Once you see the tests pass, change the **Assert.IsType<List<Company>>(result);** to **Assert.IsNotType<List<Company>>(result);** in our **CompanyRepositoryTests.cs** file and run the Docker build again.

You should get something like this:

```
[+] Building 6.5s (12/14)
 => [internal] load build definition from Dockerfile                                                0.0s
 => => transferring dockerfile: 32B                                                                 0.0s
 => [internal] load .dockerignore                                                                   0.0s
 => => transferring context: 34B                                                                    0.0s
 => [internal] load metadata for mcr.microsoft.com/dotnet/sdk:6.0                                   0.0s
 => [ 1/10] FROM mcr.microsoft.com/dotnet/sdk:6.0                                                   0.0s
 => [internal] load build context                                                                   0.0s
 => => transferring context: 8.25kB                                                                 0.0s
 => CACHED [ 2/10] WORKDIR /home/app                                                                0.0s
 => CACHED [ 3/10] COPY ./*.sln ./                                                                  0.0s
 => CACHED [ 4/10] COPY ./*/*.csproj ./                                                             0.0s
 => CACHED [ 5/10] RUN for file in $(ls *.csproj); do mkdir -p ./${file%.*}/ && mv $file ./${file%.*}/; done  0.0s
 => CACHED [ 6/10] RUN dotnet restore                                                               0.0s
 => CACHED [ 7/10] COPY . .                                                                         0.0s
 => ERROR [ 8/10] RUN dotnet test ./Tests/Tests.csproj                                              6.4s
```

```
#12 4.957 Starting test execution, please wait...
#12 4.986 A total of 1 test files matched the specified pattern.
#12 6.215 [xUnit.net 00:00:00.79]     Tests.CompanyRepositoryTests.GetAllCompaniesAsync_ReturnsListOfCompanies_WithASing
leCompany [FAIL]
#12 6.314   Failed Tests.CompanyRepositoryTests.GetAllCompaniesAsync_ReturnsListOfCompanies_WithASingleCompany [70 ms]
#12 6.314   Error Message:
#12 6.314    Assert.IsNotType() Failure
#12 6.314 Expected: typeof(System.Collections.Generic.List<Entities.Models.Company>)
#12 6.314 Actual:   typeof(System.Collections.Generic.List<Entities.Models.Company>)
#12 6.314   Stack Trace:
#12 6.314      at Tests.CompanyRepositoryTests.GetAllCompaniesAsync_ReturnsListOfCompanies_WithASingleCompany() in /home
/app/Tests/CompanyRepositoryTests.cs:line 29
#12 6.322
#12 6.326 Failed!  - Failed:     1, Passed:     0, Skipped:     0, Total:     1, Duration: < 1 ms - /home/app/Tests/bin/
Debug/net6.0/Tests.dll (net6.0)
```

The test has failed, and the build has stopped. The publish step hasn't been triggered, which is exactly what we want.

Excellent.

We've gone through the entire process, but there is one thing that some of you might have noticed. That's not something you want to use to run your containers from.

SDK images are powerful and we use to build and run applications. Nevertheless, to deploy our application to a production environment, we should create an image that is optimized for that purpose only.

In comparison to SDK images, runtime-only images are much lighter.

So let's see how to upgrade our **Dockerfile** and publish our application to the runtime-optimized image.

4.7 Creating Multistage Builds in Dockerfiles

For this purpose, we are going to use something called a multistage build in the Docker world. Multistage builds can be created by using FROM command multiple times in a **Dockefile**.

So we are going to do just that to upgrade our process. Instead of using our SDK image to publish our application with, we are going to introduce another base image to the Dockerfile and publish the artifacts inside it.

Let's see how to do that:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 as build-image
WORKDIR /home/app
COPY ./*.sln ./
COPY ./*/*.csproj ./
RUN for file in $(ls *.csproj); do mkdir -p ./${file%.*}/ && mv $file
./${file%.*}/; done
RUN dotnet restore
COPY . .
RUN dotnet test ./Tests/Tests.csproj
RUN dotnet publish ./CompanyEmployees/CompanyEmployees.csproj -o /publish/
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /publish
COPY --from=build-image /publish .
ENV ASPNETCORE_URLS=https://+:5001;http://+:5000
ENTRYPOINT ["dotnet", "CompanyEmployees.dll"]
```

So, what's new here?

First, we added ***as build-image*** to the first FROM command for easier referencing later.

Second, instead of running the application inside the SDK image, we copied the contents of the publish folder inside our SDK image to the publish folder of the runtime image.

Third, we moved the entry point to the runtime image, so that we run the application when we instantiate the runtime container. The SDK container shall no longer be responsible for running our application.

We've also added a https port, which we want going forward. HTTPS is a standard these days and we want to run our application securely.

So, what's the result of all this? Let's see.

Type the **docker build -t codemazeblog/companyemployees:runtime .** command and enjoy the process.

After the build finishes, we can see two images. One is tagged and one is not. One is runtime and one is SDK. In this case, we are not interested in the SDK image, but you can clearly see the size difference.

So that's it, now if we run **docker run --rm -it -p 8080:5000 –p 8081:5001 codemazeblog/companyemployees:runtime** we'll get the following message (again, if you have a problem with pasting this command, type it manually, works for some strange reason):

```
System.InvalidOperationException: Unable to configure HTTPS endpoint. No server certificate was specified, and the default developer certificate could not be found or is out of date.

To generate a developer certificate run 'dotnet dev-certs https'. To trust the certificate (Windows and macOS only) run 'dotnet dev-certs https --trust'.
```

So up to this point, we could run our containers in an insecure manner. To be able to run it in this multistage build, we need to configure our HTTPS and add a server certificate.

4.8 Adding a Server Certificate

To add a localhost trusted certificate for local development, we can start by cleaning existing certificates if they exist. Normally there are some on your machine if you ever run a project on HTTPS.

We had some issues running these commands in Powershell so we recommend Command Prompt in this case.

So let's clean the existing ones (prompt Yes):

```
dotnet dev-certs https --clean
```

After that, we can add our certificate with a password:

```
dotnet dev-certs https -ep
%USERPROFILE%\.aspnet\https\companyemployees.pfx -p awesomepass
```

And we need to make sure to trust that certificate we've added:

```
dotnet dev-certs https --trust
```

After this command, you should be prompted to accept the local certificate.

Now we can run our application:

```
docker run --rm -it -p 8080:5000 -p 8081:5001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="awesomepass" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/companyemployees.pf
x -v %USERPROFILE%\.aspnet\https:/https/
codemazeblog/companyemployees:runtime
```

Seems a bit complicated but all we did is added a few environment variables with -e to define and mounted the folder with the certificate we've created. We'll make it more readable in the next chapter using the docker compose tool.

If you still have problems with the certificate, navigate to the pfx file manually and click on it. Go through the certificate installation wizard, enter the password and then try to run the application again.

That's it, if you did everything correctly, you can access the application at https://localhost:8081/swagger and check if the certificate is indeed valid (click on the tiny lock button left of the URL). You might want to clear the browser cache or run incognito to make sure the certificate is applied.

Let's wrap the chapter up with the commands you might find useful while following these steps.

4.9 Some Useful Commands

Here are some of Docker commands that might help you along the way:

- **docker images** - lists all Docker images on your machine
- **docker ps** - lists all running Docker containers on your machine
- **docker ps -a** - lists all the attached but not running containers
- **docker --help** - Docker help (no s..t Sherlock)
- **docker logs <container_name>** - outputs the container logs (useful when running the container in a detached mode for example)
- **docker events** - outputs the events that happen on the server, depending on the Docker object (attach, detach, copy, pull...)
- **docker stop <container_name>** - stops the container by name
- **docker rm <container_name>** - removes the container by name
- **docker rmi <image_id>** - removes the image
- **docker rmi $(docker images -q -f dangling=true) -- force** - this one is particularly interesting, removes all the images that are not tagged (<none>). There can be a lot of these after some time.
- **docker rm $(docker ps -a -q)** - removes all containers
- **docker rmi -f $(docker images -a -q)** - removes all images

Great, now that we've conquered Dockerfiles and multistage builds, it's the right time to move forward and see how we can improve our Docker skills even more.

5 Multi-Container Applications with Docker Compose

To make all of our efforts so far legit, we are going to add a MSSQL database as another container and connect it with our application. Since we'll have multiple containers running we are going to introduce the Docker Compose tool which is one of the best tools for configuring and running multi-container applications.

Docker Compose comes together with the Docker installation on the Windows machine so we should be all set.

5.1 What is Docker Compose?

As we already mentioned, Docker Compose is a tool for defining and running multi-container applications.

But what does that mean exactly and how does it do that?

Docker Compose uses a YAML file to define the services and then run them with a single command. Once you define your services, you can run them with the **docker-compose up** command, and shut them down with the **docker-compose down** command.

If you've followed along, you might have noticed that running Docker images can get pretty complicated. This is not such a big problem once you learn what each Docker command does, but once you have multiple images, the pain of running all of them manually rises exponentially.

That's where Docker Compose comes in.

Is that all it does?

Well, not quite.

Other features of Docker Compose include:

- Using project names to create multiple isolated environments on a single host
- Creating a network inside the environment
- Preserving the volume data that your containers used
- Only recreating the containers that have changed by caching the configuration
- Allowing usage of variables inside the YAML file so you can customize the configuration for different environments and users
- Working very well with continuous integration tools (like TeamCity or Jenkins for example)

Overall, Compose is a nifty and powerful tool for managing your multi-container apps, and we are going to see just how to use it for our ASP.NET Core App by adding the MSSQL image as our database container.

Let's drill down into it and see how awesome Compose can be.

5.2 Adding Docker Compose to Our Application

Compose relies on a YAML file. This file is usually called **docker-compose.yml** and it's placed at the root of our project.

So let's start by adding a **docker-compose.yml** file to the root of our solution.

Once we've created the file, we can start adding some commands to it:

```
version: '3.0'
services:
   companyemployees:
     image: codemazeblog/companyemployees:runtime
     ports:
       - "8080:5000"
       - "8081:5001"
```

This is the simplest of Compose files, and it's practically the same thing we did with the **docker run** command in the previous chapter, but without HTTPS stuff.

To jog your memory, we used to run our application like this:

```
docker run --rm -it -p 8080:5000 -p 8081:5001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="awesomepass" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/companyemployees.pf
x -v %USERPROFILE%\.aspnet\https:/https/
codemazeblog/companyemployees:runtime
```

To accommodate for this, we need to expand our docker-compose.yml a bit:

```
version: '3.0'
services:
   companyemployees:
     image: codemazeblog/companyemployees:runtime
     ports:
       - "8080:5000"
       - "8081:5001"
     environment:
       - ASPNETCORE_Kestrel__Certificates__Default__Password=awesomepass
       -
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/companyemployees.pf
x
       - SECRET=CodeMazeSecretKey1234

     volumes:
       - ${USERPROFILE}/.aspnet/https:/https/
```

What we did is simply add the environment variables and volume to the docker compose file itself.

We'll also going to add the environment variable SECRET which will help us work with JWT.

As you might have noticed, the file contains sensitive information now, so the important thing to remember is not to commit it to public repos.

Ideally, this information should not be kept in the **docker-compose** file itself.

You can use the power of volumes to add the sensitive data to the container, by mounting volumes with files containing sensitive data. For the simplicity of this example, we will keep it this way.

Now **the only command we need to run** to achieve the same result is **docker-compose up** and our application will be running as usual.

As you can see, the **docker-compose.yml** file contains a set of services. In our case, this is just our application, but soon we're going to add one more service to it. We've described what image we want to run and which port we need to expose.

We can run our services in the background by adding -d flag: **docker-compose up -d**. In that case, we would need to stop the service(s) with **docker-compose down**.

One thing to note here is that if we quit a container with Ctrl+C, it won't kill the container or the network created by Compose. To make sure you release the resources you need to run **docker-compose down**.

5.3 Building the Image With Compose

Okay, so until now, we've just run the existing image with docker compose. But if we make some changes to the application or the **Dockerfile**, Compose would still run the image we've built before those changes.

We would need to build the image again with the **docker build -t codemazeblog/companyemployees .** command, and then run the **docker-compose up** command to apply those changes.

To automate this step we can modify our **docker-compose.yml** file with the build step:

```
version: '3.0'
services:
   companyemployees:
     image: codemazeblog/companyemployees:runtime
     build:
       context: .
     ports:
       - "8080:5000"
       - "8081:5001"
     environment:
       - ASPNETCORE_Kestrel__Certificates__Default__Password=awesomepass
       -
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/companyemployees.pf
x
     volumes:
       - ${USERPROFILE}/.aspnet/https:/https/
```

The build step sets the context to the current directory and enables us to build our image using the **Dockerfile** defined in that context.

So, now we can add the **--build** flag to our **docker-compose** command to force the rebuild of the image: **docker-compose up --build**.

Now try running the command and see for yourself if that's the case. We have removed the need for the **docker build** command just like that.

Okay, great.

Let's move on to the main attraction where the real fun begins.

5.4 Adding a MSSQL Database with Docker Compose

While preparing our ASP.NET Core application for dockerization, we switched from the MSSQL database to the in-memory one. We did this to demonstrate Docker commands easier.

Now that we introduced Docker Compose, we can revert the changes we made to the **ServiceExtensions.cs** class:

```
public static void ConfigureSqlContext(this IServiceCollection services,
IConfiguration configuration) =>
            services.AddDbContext<RepositoryContext>(opts =>

opts.UseSqlServer(configuration.GetConnectionString("sqlConnection"),
                    b => b.MigrationsAssembly("CompanyEmployees")));
```

Now, we can start using MSSQL database again.

There is one more thing we need to change, and that's the connection string in the **appsettings.json** file since we don't want to use the root user. We are also changing the server from **.** to **db**. You'll see why in a moment:

```
"ConnectionStrings": {
    "sqlConnection": "server=db; database=CompanyEmployee; User Id=sa;
Password=AwesomePass_1234"
  }
```

Okay, excellent.

Now that we prepared our application, let's add the MSSQL image to our **docker-compose.yml**. We are going to use mcr.microsoft.com/mssql/server:2019-latest image since it's compatible with our application:

```
version: '3.0'
services:
   db:
     image: mcr.microsoft.com/mssql/server:2019-latest
     ports:
       - "1433:1433"
     environment:
       - ACCEPT_EULA=Y
       - SA_PASSWORD=AwesomePass_1234
     restart: always
   companyemployees:
     depends_on:
       - db
     image: codemazeblog/companyemployees:runtime
     build:
       context: .
     ports:
       - "8080:5000"
```

```
      - "8081:5001"
    environment:
      - ASPNETCORE_Kestrel__Certificates__Default__Password=awesomepass
      -
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/companyemployees.pf
x
      - SECRET=CodeMazeSecretKey
    volumes:
      - ${USERPROFILE}/.aspnet/https:/https/
```

So, as you may see, we added the **db service** to our **docker-compose.yml**. This service name **is also a server name for that container** and that's why we've used it in our connection string. In the db service, we are configuring the MSSQL container once we run the image.

We need to open the port 1433 if we want to access the server locally.

A MSSQL image has some predefined environment values that it looks for while initializing.

These are:

ACCEPT_EULA=Y we need to accept the End-User Licencing Agreement

SA_PASSWORD=AwesomePass_1234 creates the "sa" user with this password

MSSQL_PID we are not using this one because the default value is "Developer" and that's what we need in this case. Other values are: Express, Standard, Enterprise, EnterpriseCore. Use the one that suits your needs.

With **restart: always** we are instructing Docker Compose to restart our service in case the container goes down for whatever reason.

And finally, we've added **depends_on: - db** to our app service, to make db service be the first to load. This mechanism tells the containers in

which order to start, but can't guarantee that the containers are indeed ready. For example, we can try to seed the data even though the database has not started yet.

We can also use this opportunity to add the environment variable "SECRET" which is used for our JWT authentication. Once again, environment variables with sensitive data like this one should not be kept in the docker-compose.yml file. One of the right ways is to keep it in a file on a local system and then mount a volume with that file so we can read and use the values.

And now that we use the real database, let's create a simple Migrations manager class that will run our migrations on **docker-compose up**.

So, let's navigate to the Repository project and create **MigrationsManager** class:

```
private static int _numberOfRetries;

public static IHost MigrateDatabase(this IHost host)
{
      using (var scope = host.Services.CreateScope())
      {
            using var appContext =
scope.ServiceProvider.GetRequiredService<RepositoryContext>();
            try
            {
                  appContext.Database.Migrate();
            }
            catch (SqlException)
            {
                  if (_numberOfRetries < 6)
                  {
                        Thread.Sleep(10000);

                        _numberOfRetries++;
                        Console.WriteLine($"The server was not found or
was not accessible. Retrying... #{_numberOfRetries}");

                        MigrateDatabase(host);
                  }
```

```
                    throw;
                }
            }

            return host;
        }
```

This MigrationsManager class has a simple mechanism to seed data and retry 6 times if the SQL server is not available. This will probably not happen in most cases, but the mechanism is here to prevent that. The main part of this class is the **appContext.Database.Migrate()** method, which initiates the seed migration when the application is started.

Make sure to import **Microsoft.Data.SqlClient** and **Microsoft.Extensions.Hosting.Abstractions** NuGet packages to compile this class.

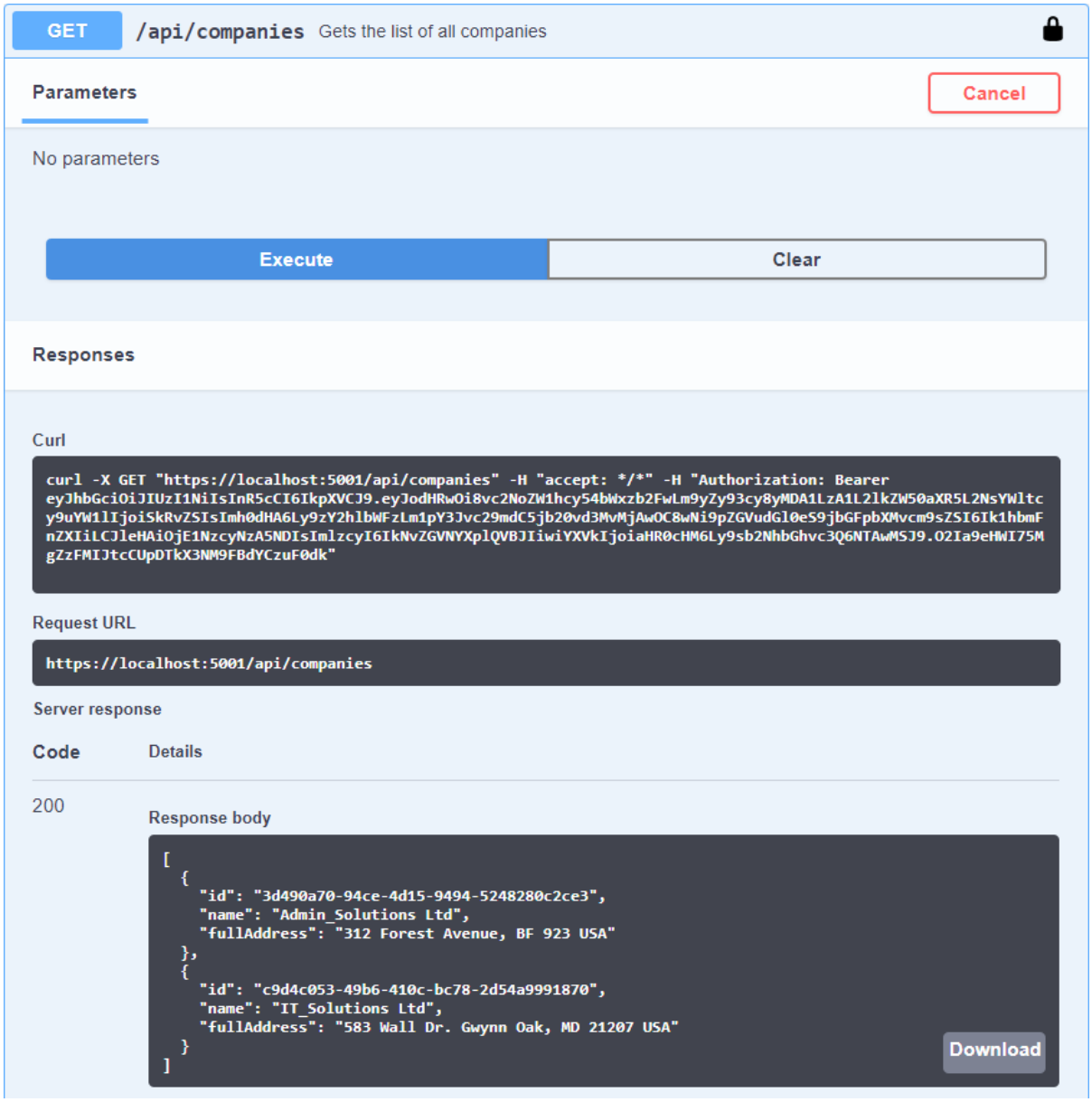
Now the only thing we need to do is to extend our Program.cs code a bit:

```
app.MigrateDatabase().Run();
```

We've just added MigrateDatabase() before we actually run the application.

If we run the **docker-compose up --build**, we should get both our database and application up and running and the database should have the seed data.

Great, now let's try it out through swagger interface with api/companies request:



***NOTE**: If you have problems with authentication, refer to the chapter "**JWT and Identity in ASP.NET Core**" in the book. We have explained how to create a user, login and then authorize in Swagger UI.*

Having done all this, we can conclude this chapter.

We've learned how to make a Compose file and how to make it build and run our images with the **docker-compose up** and **docker-compose down** commands.

Docker Compose has allowed us to easily add a MSSQL database container that our ASP.NET Core app can persist its data in. This setup will help us in the next chapter where we are going to learn how to make a local registry and push our images to it instead to the Docker Hub.

6 Docker Hub vs Creating a Local Docker Registry

In this chapter, we will learn more about image management and distribution. There are several ways to do that, whether locally or in the cloud, so you should probably take some time to learn these concepts before starting with continuous integration and application deployment.

We are also going to learn the **difference between a Docker registry and a Docker repository** and how to persist the changes we made to our images. Then, we'll cover **Docker Hub** and how to make our own local Docker registry.

6.1 Difference Between Docker Repository and Docker Registry

Besides sounding similar these terms can sometimes cause confusion so it seems appropriate to start by explaining what each one means.

Docker Registry (Docker Trusted Registry - DTR) is an enterprise-grade storage solution for Docker images. In other words, it's an image storage service. Think about GitHub, but for Docker Images.

We can use one of the existing and well-established cloud registries like Docker Hub, Quay, Google Container Registry, Amazon Elastic Container Registry or any other. We can also make our own registry and host it locally.

One docker registry can contain many different docker repositories.

Docker Repository is a collection of Docker images with the same name and different tags. For example, the repository we've used several times so far, **mcr.microsoft.com/dotnet/aspnet** repository, has many different images in it.

Each image has it's own tag. For example, for the entire series, we've been using the **mcr.microsoft.com/dotnet/aspnet:6.0** image which contains ASP.NET Core runtime with version 6.0. We can choose which one we want to pull by typing **docker pull image-name:tag** similar to GitHub repo and commits. We can go back to whichever commit we want and pull it to the local machine.

That's putting it in very simple terms. But now that we cleared the air around these terms we can proceed to the next section.

6.2 More About Docker Hub

As we've mentioned, Docker Hub is just one of the registry providers. And a good one at that. We can find all sorts of images over there and push our own. We can create unlimited public repositories and one private repo free of charge. If you need more private repositories, you can choose one of the Docker Hub monthly plans.

Besides providing a centralized resource for image discovery and distribution, Docker Hub's functionality extends to:

- Automated builds of images on source code changes and parallel builds
- Webhooks on image creation and push
- Groups and organizations management
- GitHub and BitBucket integration

You can create your own account on Docker Hub right now and try it out. We did something like that in chapter 2, but now we are going in-depth.

To push the image from the local machine to Docker Hub we need to type **docker login** and enter the credentials of your account in the prompt. After that, you can easily push the image by typing **docker push accountname/imagename:tag**.

If we don't specify the tag Docker will apply the **:latest** tag to it.

If we want to pull the image from the Docker Hub to the local machine, we need to type **docker pull accountname/imagename:tag**. The same rule applies here. If you don't specify the tag, you are going to pull the image tagged **:latest**.

6.3 Creating a Local Docker Registry

Docker Hub is super neat and very intuitive and offers a great deal of functionality for free.

But what if we need more privacy? Or our client wants to use its own server. If that's the case, we can make **our own Docker Registry**.

So how do we do that?

Well, we can set up the registry in two different ways:

- directly with the Docker command
- using Docker compose

Creating a local registry using docker is pretty straightforward and shouldn't be such a big deal if you followed the book so far. The registry repository is located on the Docker Hub here. Aren't you glad now that we talked about the differences between these terms :)

So if we want to set up the local registry we can type:

```
docker run -d -p 50000:5000 --restart always --name my-registry
registry:latest
```

Now we can navigate to **http://localhost:50000/v2/_catalog** and see for yourself that your registry is up and running and that you have no repositories pushed to it.

You should be able to see something like this:

```
// 20191218210805

// http://localhost:50000/v2/_catalog

{

  "repositories": [

  ]

}
```

The same thing can be done with Docker Compose that we introduced in the previous chapter of the series.

Let's navigate to the root of our solution and make a new folder **Infrastructure** and in it, another one called **Registry**. In the **Registry** folder, we are going to create a **docker-compose.yml** file:

```
version: '3.0'

services:
  my-registry:
    image: registry:latest
    container_name: my-registry
    volumes:
      - registry:/var/lib/registry
    ports:
      - "50000:5000"
    restart: unless-stopped
volumes:
  registry:
```

We defined the same thing we did with Docker command but with some additional goodies. We added a volume to persist our data and defined the restart policy as **unless-stopped**, to keep the registry up unless it is explicitly stopped.

Now we can stop the registry we've spin up before with **docker stop my-registry** and **docker rm my-registry** to remove the attached container.

After that run **docker-compose up -d** in the **/Infrastructure/Registry** folder.

And guess what?

We have the exact same registry we spin up before and we can access it by navigating to **http://localhost:50000/v2/_catalog**.

We should also add the entry to the windows hosts file so we can use my-registry instead of localhost (usually at **C:/Windows/system32/drivers/etc/hosts**):

```
127.0.0.1     my-registry
```

6.4 Pushing Images to a Local Docker Registry

Ok, so now we have our own local registry. Let's push some images to it.

If you've followed the series you have the **codemazeblog/companyemployees:runtime** image on your local machine. If not, please refer to the previous part of this bonus where you can learn how to create images with Docker Compose.

To push the image to local registry we need to tag it appropriately first:

```
docker tag codemazeblog/companyemployees:runtime my-
registry:50000/codemazeblog/companyemployees:runtime
```

And then we can push it to the registry:

```
docker push my-registry:50000/codemazeblog/companyemployees:runtime
```

Now, if we browse to **http://localhost:50000/v2/_catalog** we will see that our repository list is no longer empty. We've successfully added our image to the local registry!

Moreover, if we navigate to **http://localhost:50000/v2/codemazeblog/companyemployees/tags/list** we'll see:

```
// 20191218210646

// http://localhost:50000/v2/codemazeblog/companyemployees/tags/list

{

  "name": "codemazeblog/companyemployees",

  "tags": [

    "runtime"

  ]

}
```

Here we can find the list of all the available tags of our image.

To test this out we can remove the local image with **docker rmi my-registry:50000/codemazeblog/companyemployees:runtime**. Now you can pull the image from the local registry by typing **docker pull my-registry:50000/codemazeblog/companyemployees:runtime** and behold, once again, the image is on your local machine!

That wasn't too hard, was it?

There is one important note here. **Pushing images to the insecure registries is not recommended** and we can easily be the target of the MITM attack if we are not careful.

Setting the registry certificate is beyond the scope of this bonus book, but you can find more info on how to set a certificate for a local registry here.

So be careful and secure your registries if you want to use them in production.

Let's wrap this up with some examples of when we might to set up a local Docker registry.

6.5 Use Cases for the Local Docker Registry

Now that you know pretty much everything you need to run a local registry, you might wonder: "But why should I use a local registry when I have all those nice options available?".

There are a few reasons for that:

- Total control of our registry and repositories
- We need to set up a local network and to easily distribute images throughout it, to save the bandwidth
- We have a closed network without internet access
- Setting up a local CI build server that uses that registry
- We don't want to pay some crazy plans to cloud providers to host our repositories

So there you go.

This wraps it up for this bonus material book.

Now you know how to properly create and persist your Web API application as a docker image.

We've given you the tools to play with. Docker is a big topic and the one that can easily take a whole book to cover. We encourage you to play around with it and explore other possibilities.

Hopefully, this bonus book gave you some ideas and at least a peek at one bit of potential that Docker has.

Happy dockerization, and make sure to share your results with us!