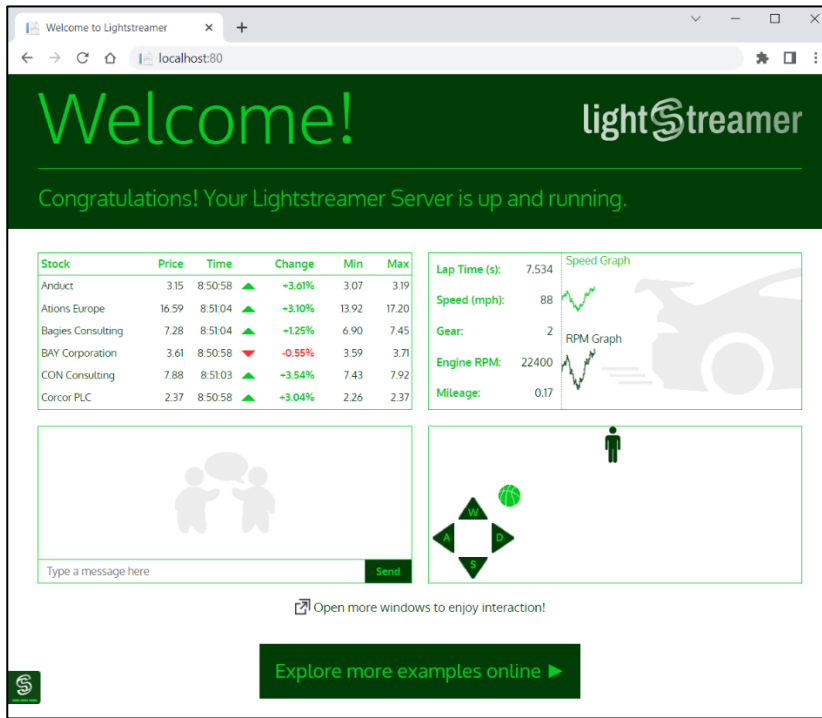


# Exercises: Containers and Docker

Problems for exercises for the ["Containers and Clouds" course @ SoftUni](#).

## 1. Lightstreamer Container





Lightstreamer (<https://lightstreamer.com>) is a **web-based asynchronous messaging project**.



Your task is to **run it in a Docker container**. For running the **Lightstreamer container**:

- The **image** you need is **lightstreamer:latest**
- Your **container's name** should be **ls-server**
- Server works on **port 8080**, but should be **accessed** on **localhost:80**
- Container should be run in **detached mode**

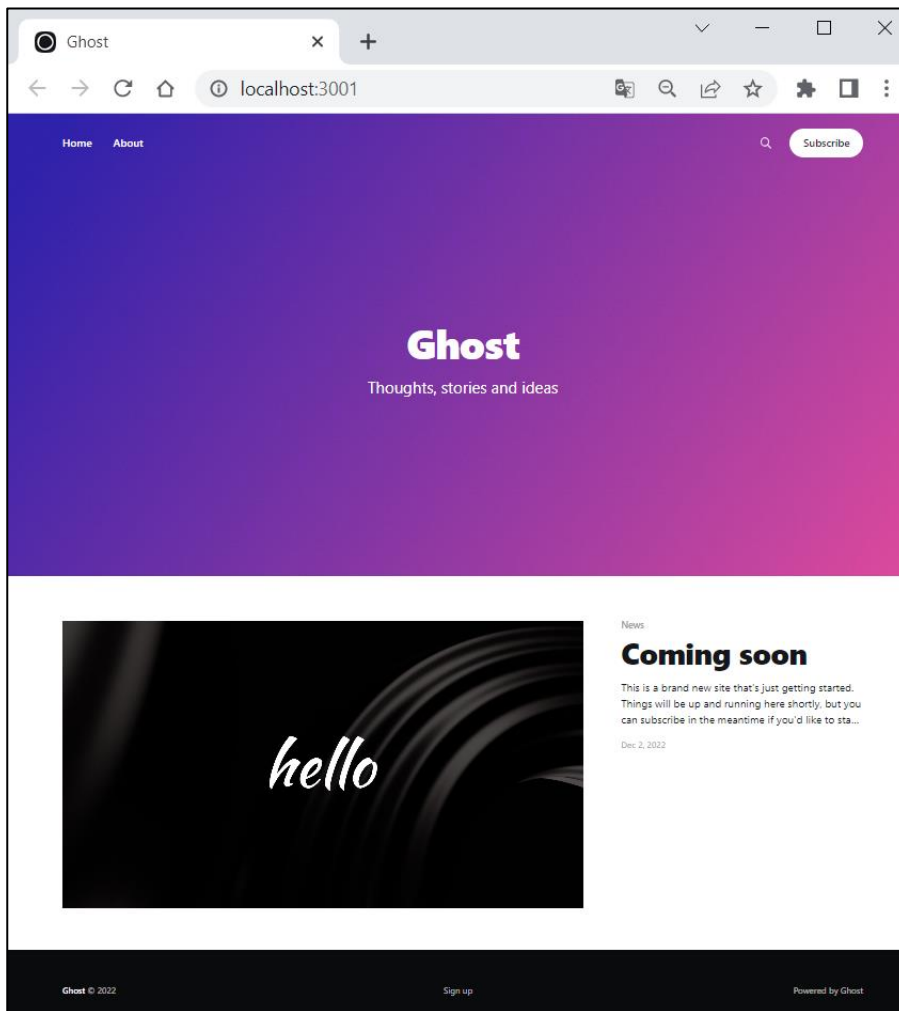
Your **container** should look like this:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
 ls-server b799a4aee59	<a href="#">lightstreamer:latest</a>	Running	<a href="#">80:8080</a>	3 minutes ago	  

Make sure your **container is created** and **Lightstreamer works in the browser**. Then you can **delete the container** and the **image**.

## 2. Ghost Container







Ghost ([https://en.wikipedia.org/wiki/Ghost\\_%28blogging\\_platform%29](https://en.wikipedia.org/wiki/Ghost_%28blogging_platform%29)) is a free and open-source **blogging platform**, written in **JavaScript**. When run in a **Docker container** and **accessed in the browser**, it looks like this:



For running your **Ghost container**, follow these **requirements**:

- The **image** you need is **ghost:latest**
- Your **container's name** should be **ghost-container**
- Server works on **port 2368**, but should be **accessed** on **localhost:3001**
- You should set **NODE\_ENV=development** as an **environment variable** with the **-e** option
- Container should be run in **detached mode**

Your **container** should look like this:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
 ghost-container 3da4bc9b2733 	<a href="#">ghost:latest</a>	Running	<a href="#">3001:2368</a> 	4 minutes ago	  

Note: if a "We'll be right back" message appears in the browser, it means that **Ghost** is still loading, so **refresh the browser** and everything should be alright.

### 3. Apache HTTP Server Container







Now you should run **Apache HTTP Server** in a **Docker container**.

- Use the **latest image: httpd:latest**

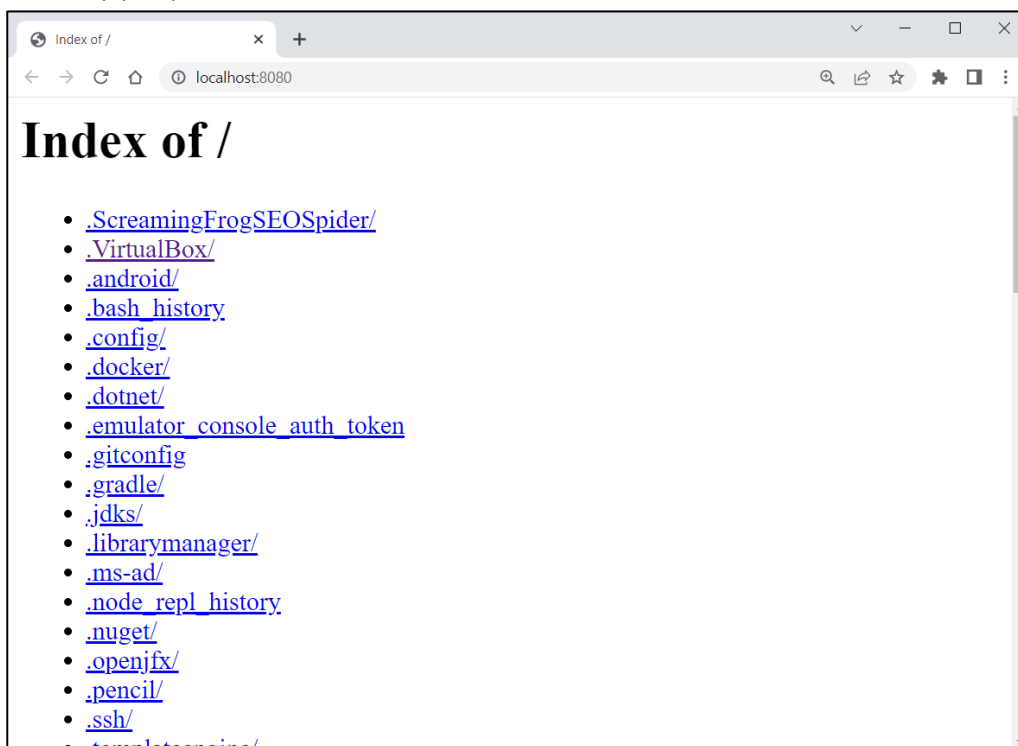


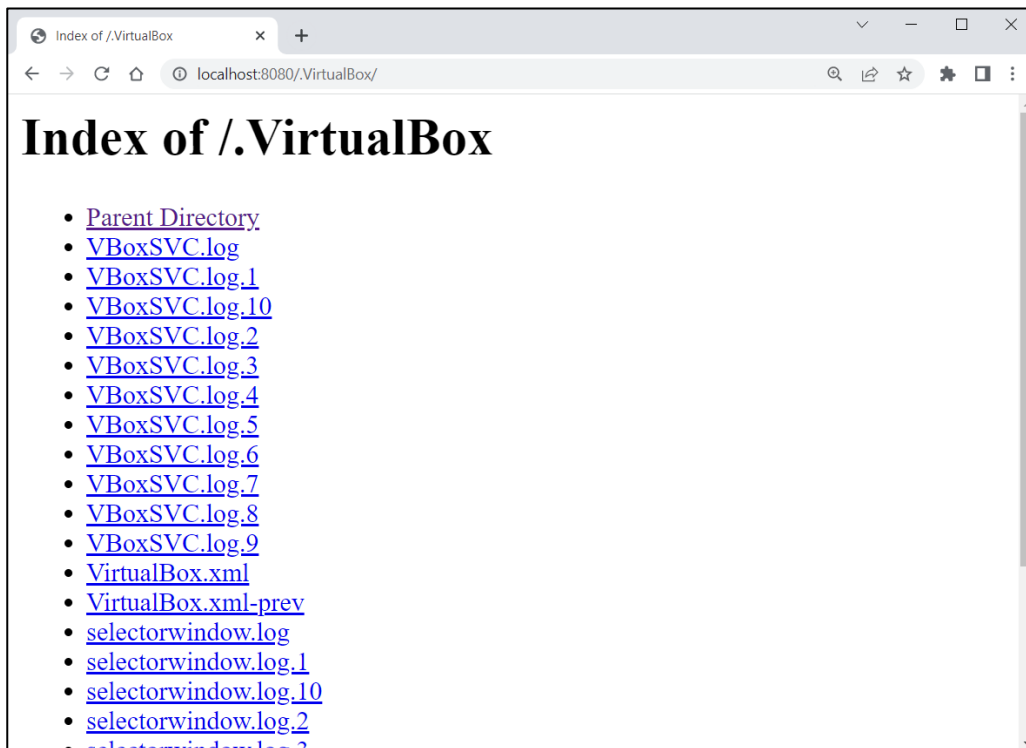
- Your **container's name** should be **my-apache-app**
- **Server** works on **port 80**, but should be **accessed** on **localhost:8080**
- Container should be run in **detached mode**
- You should create a **volume** – map **current PowerShell** (or another) **directory** to the **container's directory** **/usr/local/apache2/htdocs/**

Your **container** should look like this:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
 my-apache-app 79caa9f2a203 	<a href="#">httpd:latest</a>	Running	<a href="#">8080:80</a> 	1 minute ago	  

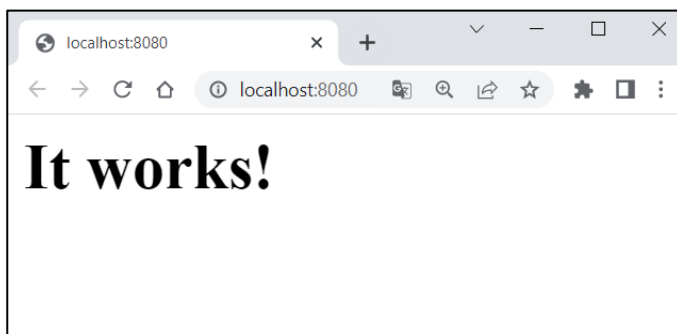
When **accessed** from the **browser**, it should list the files and folders from your local file system in the PowerShell directory you provided the server with, as well as in child directories:





The **local file system** is accessed by the **container** because of the **volume**.

However, if the browser only shows you the "**It works!**" message (see below), then you **didn't succeed in running the container properly** and you should **fix your command and try again**:



## 4. SQL Server Container

Our task is to **run a container** with an **SQL Server database** in it. To do this, we will need the **following image** from **Docker Hub**: [https://hub.docker.com/\\_/microsoft-mssql-server](https://hub.docker.com/_/microsoft-mssql-server).

You can look at the "**How to use this Image**" section to learn how to **run the database container**. However, we will also **show and explain** this step by step.

### Create the Container

Start **writing the multi-line run command** for the **Docker container**:

```
PS C:\Users\PC> docker run `
```

Let's first take care of the **environment variables** needed for the **SQL Server container**. We should **confirm the acceptance to licensing agreement** with **ACCEPT\_EULA=Y**:

```
>> -e ACCEPT_EULA=Y `
```

We should also **set a password** for the **database system administrator (sa)** to **connect to SQL Server** once the container is running:

```
>> -e MSSQL_SA_PASSWORD=yourStrongPassword12#`
```

Note: your **password** should follow the **requirements from the documentation**: "This password needs to include at least 8 characters of at least three of these four categories: uppercase letters, lowercase letters, numbers and non-alphanumeric symbols".

Next, we should **expose a port for the container**. The server works on port **1433** and we will start it **locally on the same** one:

```
>> -p 1433:1433`
```

Then, we should **create a volume**, otherwise **data will be lost** when container is stopped, which is bad for a database container. We will name our **volume sqldata** and map it to the **/var/opt/mssql** directory, where **SQL Server data is stored**:

```
>> -v sqldata:/var/opt/mssql`
```





At the end, we will use the **-d** option to run the container in **detached mode** and will use the **mcr.microsoft.com/mssql/server** image:

```
>> -d mcr.microsoft.com/mssql/server`
```

Note: we **didn't pull the image in advance** but don't worry – it will be **pulled automatically** when the **docker run** command is executed.

**Execute** the above command and the **container should be created**:

```
PS C:\Users\PC> docker run`
>> -e ACCEPT_EULA=Y`
>> -e MSSQL_SA_PASSWORD=yourStrongPassword12#`
>> -p 1433:1433`
>> -v sqldata:/var/opt/mssql`
>> -d mcr.microsoft.com/mssql/server
Unable to find image 'mcr.microsoft.com/mssql/server:latest' locally
latest: Pulling from mssql/server
342d87d17479: Pull complete
112c1458d0bd: Pull complete
04016b3a8e25: Pull complete
Digest: sha256:7c61aeefa1c8eb55bccfa8d536a283ec922c486c7688e51f193b84c5f0aa3768
Status: Downloaded newer image for mcr.microsoft.com/mssql/server:latest
a7b7d5ddcf99b35974ecee1251e3c51df1e33e6578837bb420c6aebd146cbcbd
```

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
 inspiring_chatelet a7b7d5ddcf99	<a href="https://mcr.microsoft.com/mssql/server:latest">mcr.microsoft.com/mssql/server:latest</a>	Running	<a href="#">1433:1433</a>	32 seconds ago	  

## 5. \*MariaDB Client and Server in a Network











**MariaDB Server** (a variant of MySQL) is one of the most popular **open-source relational databases**. You should use it **documentation** on **Docker Hub** to create **two containers**, which will **work together**:

- **MariaDB database server container**, initialized with database user and password.
- Another container, which will run the **MariaDB command line client** against the **MariaDB server container**, allowing you to **execute SQL statements** against your database instance.

Both containers will use **the same Docker image**. The **image** is available here: [https://hub.docker.com/\\_/mariadb](https://hub.docker.com/_/mariadb).

Note: In order for the **containers to work together**, they should be in the **same network**. See in the documentation **how to create a network** and **connect both containers** to it.

At the end, you should have **two containers** like this:

	NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
	mariadb_server 53f629551c1f 	<a href="#">mariadb:latest</a>	Running		29 seconds ago	  
	mariadb_client c2833bcfc58e 	<a href="#">mariadb:latest</a>	Running		28 seconds ago	  

They should be **connected in the same network**. With the command below you can see all **containers in a specified network**:

```
PS C:\Users\PC> docker network inspect mariadb_network -f "{{json .Containers }}"
{"53f629551c1f7876acb147c8e8ef5f3bc442c8c69c155a31e1febd0140d5aaa6":{"Name":"mariadb_server","EndpointID":"16c7e2402074935da3fc47d9094ace52f7830c23bf37b1885a26c8e0b780a90a","MacAddress":"02:42:ac:1d:00:02","IPv4Address":"172.29.0.2/16","IPv6Address":""},"c2833bcfc58e826cc5d0a16f31930e273164a4b467310dd94866353d09ab3708":{"Name":"mariadb_client","EndpointID":"fc2cd8bc168769ad12aaf96a54498be437dc39aa9ff4aa1d2496ba1cad9cd0e8","MacAddress":"02:42:ac:1d:00:03","IPv4Address":"172.29.0.3/16","IPv6Address":""}}
```

The **mariadb\_client** container should **access the mariadb\_server** container:

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 4
Server version: 10.10.2-MariaDB-1:10.10.2+maria~ubu2204 mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

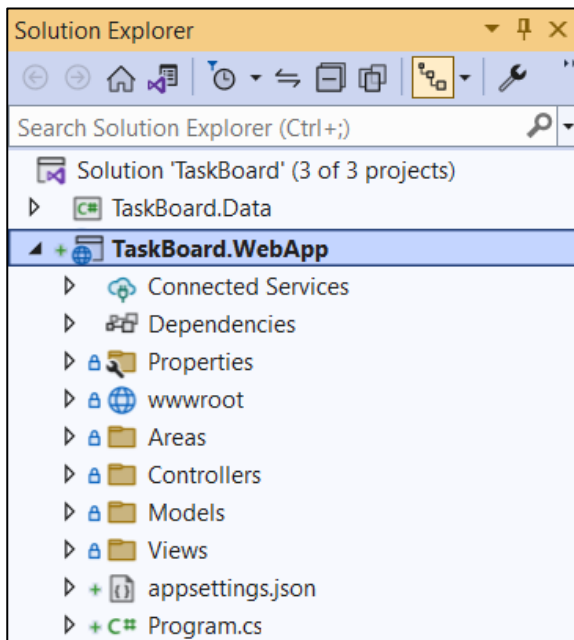
And you should be able to **run commands** on it:

```
MariaDB [(none)]> SELECT VERSION();
+-----+
| VERSION() |
+-----+
| 10.10.2-MariaDB-1:10.10.2+maria~ubu2204 |
+-----+
1 row in set (0.000 sec)
```

This example shows that we can **connect many containers**. It is usually necessary to do so and we will see how in the next lesson.

## 6. TaskBoard App: Building a Custom Image

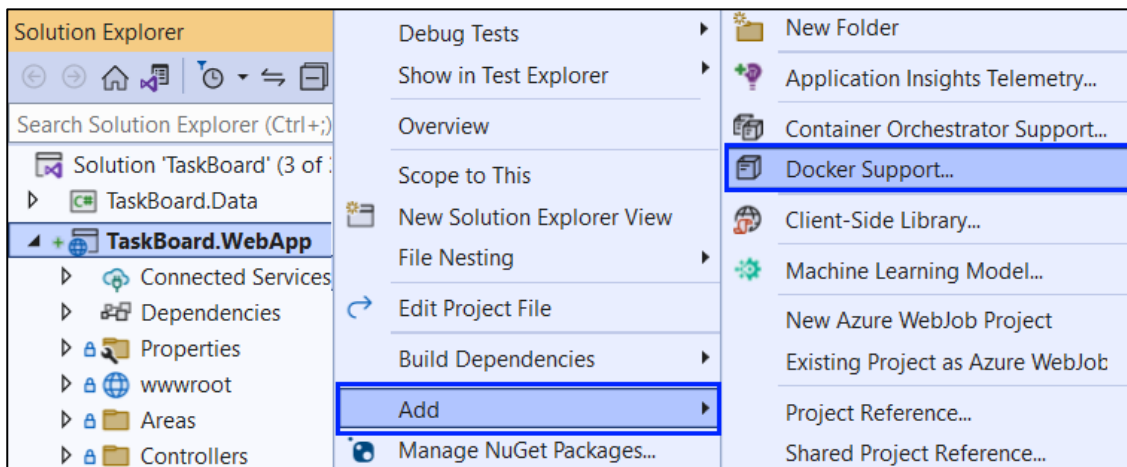
In this **task** and in the **other two tasks** connected to the **TaskBoard app**, we will work on the following **ASP.NET 6 MVC app** with a **SQL Server database**, provided in the **resources**:



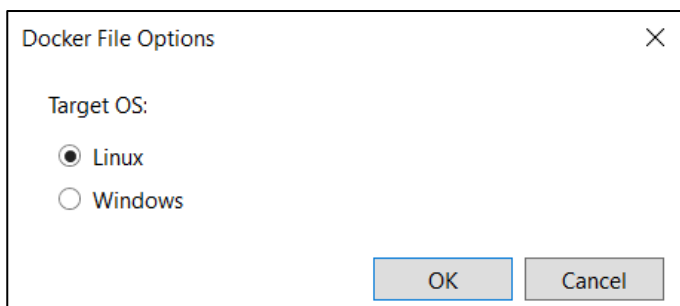
Our task is to create a custom image for this app. Later, we will also **publish this image in Docker Hub**.

## Step 1: Create a Dockerfile in Visual Studio

Our first job is to **create a Dockerfile for the app**, which will allow us to **run it in a Docker container** and later **connect it to a network**. Creating a **Dockerfile** is easy in **Visual Studio**, as it is **done for you** – you should only **right-click** on the **"TaskBoard.WebApp" project** and select **[Add] → [Docker Support...]**:

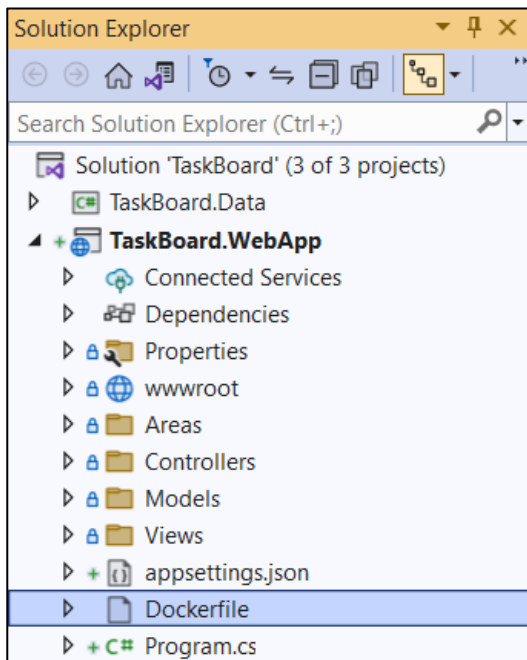


Then, you should **choose a target OS** for the **Dockerfile** – choose **[Linux]**, as we are **running Linux containers**:



The **Dockerfile** should be **created successfully**:





```

Dockerfile*
1  #See https://aka.ms/containerfastmode to understand how Visual Studio uses this Dockerfile
2
3  FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
4  WORKDIR /app
5  EXPOSE 80
6  EXPOSE 443
7
8  FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
9  WORKDIR /src
10 COPY ["TaskBoard.WebApp/TaskBoard.WebApp.csproj", "TaskBoard.WebApp/"]
11 COPY ["TaskBoard.Data/TaskBoard.Data.csproj", "TaskBoard.Data/"]
12 RUN dotnet restore "TaskBoard.WebApp/TaskBoard.WebApp.csproj"
13 COPY . .
14 WORKDIR "/src/TaskBoard.WebApp"
15 RUN dotnet build "TaskBoard.WebApp.csproj" -c Release -o /app/build
16
17 FROM build AS publish
18 RUN dotnet publish "TaskBoard.WebApp.csproj" -c Release -o /app/publish /p:UseAppHost=false
19
20 FROM base AS final
21 WORKDIR /app
22 COPY --from=publish /app/publish .
23 ENTRYPOINT ["dotnet", "TaskBoard.WebApp.dll"]

```

The **Dockerfile** contains instructions on how an image for the app should be created.

## Step 2: Build and Publish the Image to Docker Hub

We can now **build a custom image** with this **Dockerfile**. Open a CLI, for example **Powershell**, and fulfill the **following steps** to do it:

- Navigate to the **TaskBoard** solution directory
- Use the **docker build** command to **build the image**
- Set the **local directory** as the **working directory**
- With the **-f** option, set the **path to the Dockerfile**
- With the **-t** option, set the **name of the image** in format **{your Docker Hub username}/{app name}**, as we will later **add our image to Docker Hub**



The **whole command** should look similar to this (use **your Docker Hub username** instead of "softuni"):

```
PS D:\Projects\TaskBoard> docker build . -f ./TaskBoard.WebApp/Dockerfile -t softuni/taskboard_app
[+] Building 65.3s (19/19) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 32B                                                0.0s
...
=> [base 2/2] WORKDIR /app                                                         0.5s
=> [final 1/2] WORKDIR /app                                                        0.1s
=> [build 2/8] WORKDIR /src                                                        2.6s
=> [build 3/8] COPY [TaskBoard.WebApp/TaskBoard.WebApp.csproj, TaskBoard.WebApp/] 0.0s
=> [build 4/8] COPY [TaskBoard.Data/TaskBoard.Data.csproj, TaskBoard.Data/]       0.0s
=> [build 5/8] RUN dotnet restore "TaskBoard.WebApp/TaskBoard.WebApp.csproj"     8.6s
=> [build 6/8] COPY . .                                                            0.1s
=> [build 7/8] WORKDIR /src/TaskBoard.WebApp                                     0.0s
...
use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

You can see how the **instructions from the Dockerfile** are followed to **build the image**. You can see the **ready image**:

```
PS D:\Projects\TaskBoard> docker images
REPOSITORY          TAG         IMAGE ID        CREATED         SIZE
softuni/taskboard_app latest      5a3de8c4f670   22 minutes ago 254MB
```

Now let's see how to **push our custom image to Docker Hub**. Know that this is **not needed** for running a container with that image – you can have the **image only locally** and still use it. However, it is good to know **how to push images**.

To **push our image to Docker Hub**, we should first **log-in to Docker Hub** with the command below. If this is the **first time** you log in, you should **enter your credentials**. Make sure that **login is successful**:

```
PS D:\Projects\TaskBoard> docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: softuni
Password:
Login Succeeded

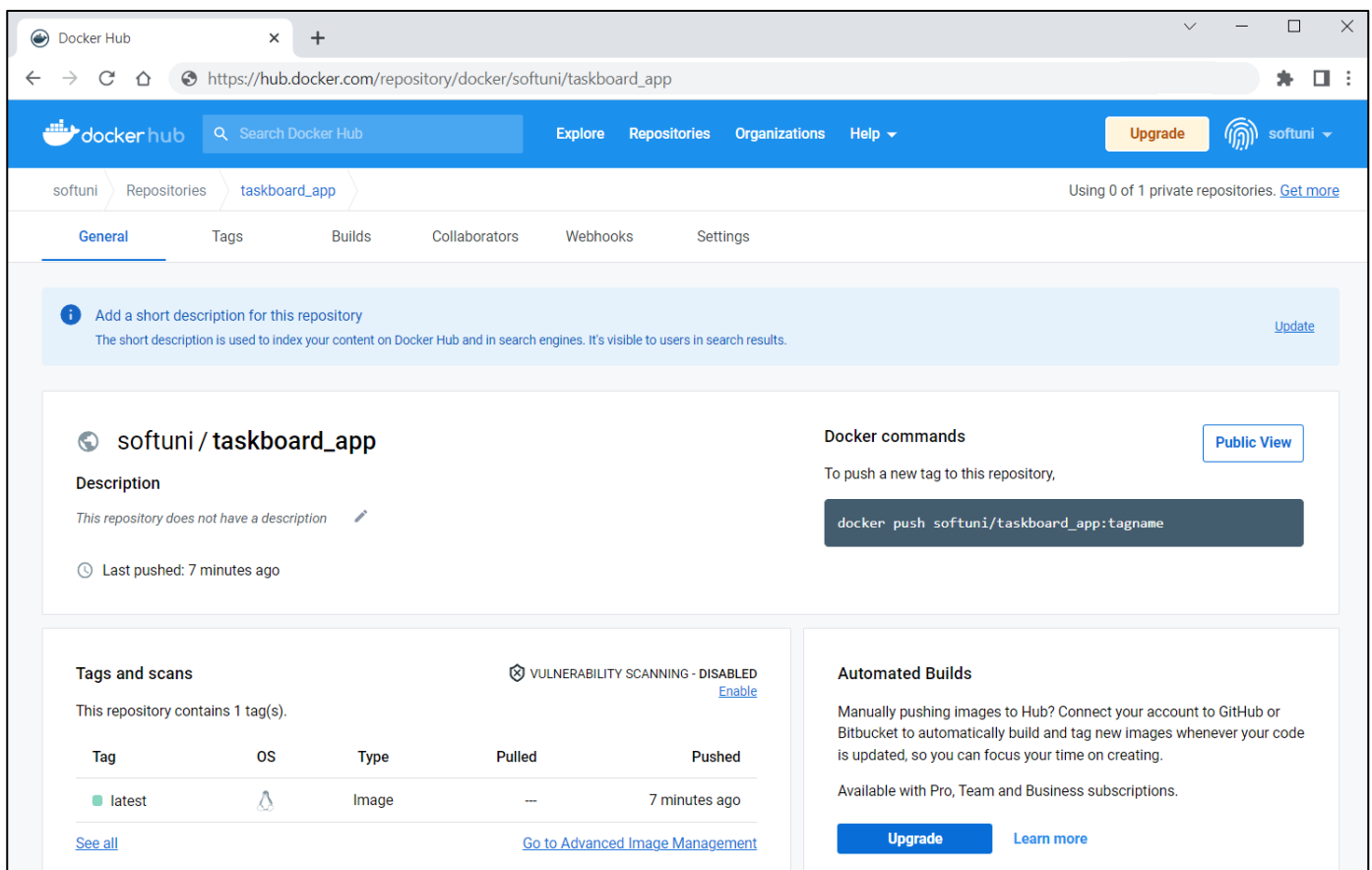
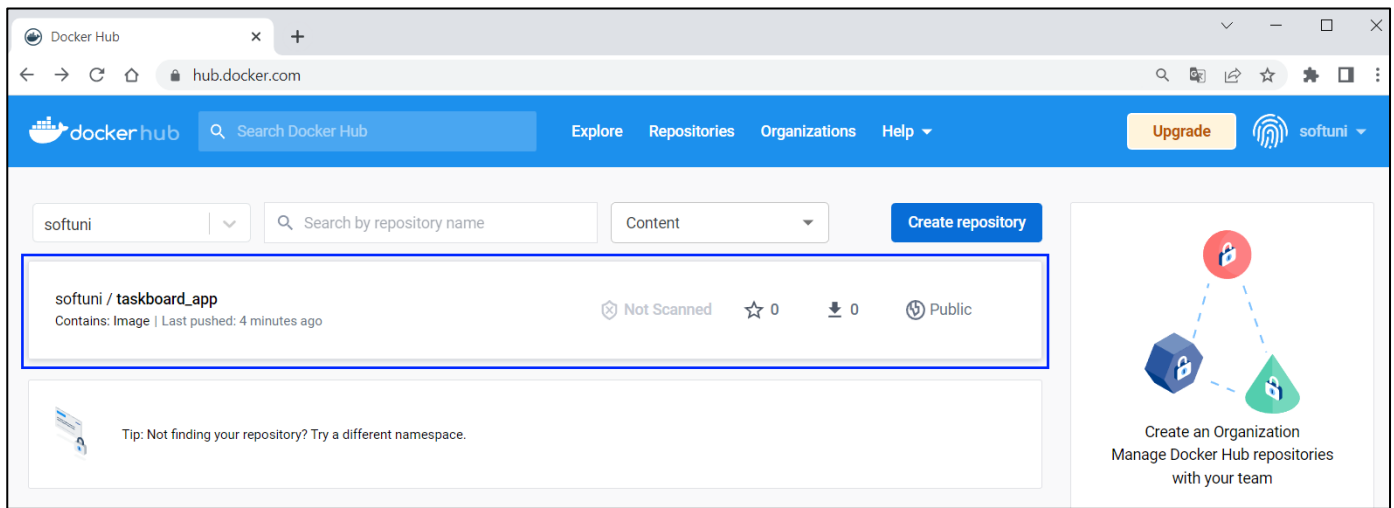
Logging in with your password grants your terminal complete access to your account.
For better security, log in with a limited-privilege personal access token. Learn more at https://docs.docker.com/go/access-tokens/
```

Now you should only **push the image**:

```
PS D:\Projects\TaskBoard> docker push softuni/taskboard_app
Using default tag: latest
The push refers to repository [docker.io/softuni/taskboard_app]
e6e6c6f54e6a: Pushed
5f70bf18a086: Mounted from bmst/h3demo
872d2fd812a2: Pushed
fc47b3bbb3a5: Pushed

4b7415c5302b: Pushed
8407279d92ac: Pushed
48b03e1004df: Pushed
ec4a38999118: Mounted from library/httpd
latest: digest: sha256:ac301372e41f673645f17fb49f3c346afaa26985b70187f92e354a1c7c41134f size: 1996
```

And it is now **available at Docker Hub** as a **public image**:

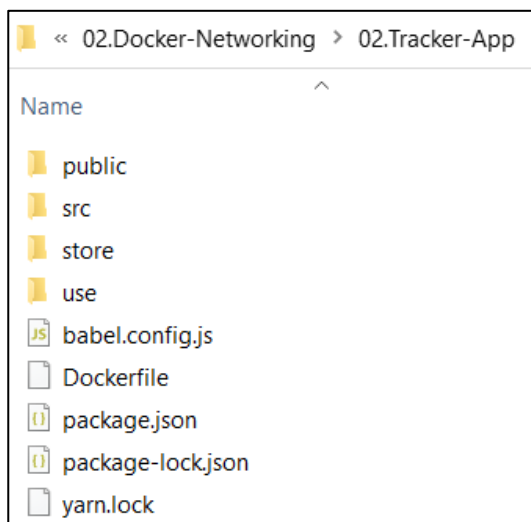


Now **keep the image** because we will use it in the next **TaskBoard app** task to **run a container**. Note that this **image is not enough** to run you whole app, as it **has a database** (and it needs a container too).

## 7. Tracker App

Your task now is to **run a simple JavaScript front-end app based on Vue.js** for keeping track of daily duties in a **Docker container**. It does not need **anything but an image** to run. It does not use a database or any other types of storage.

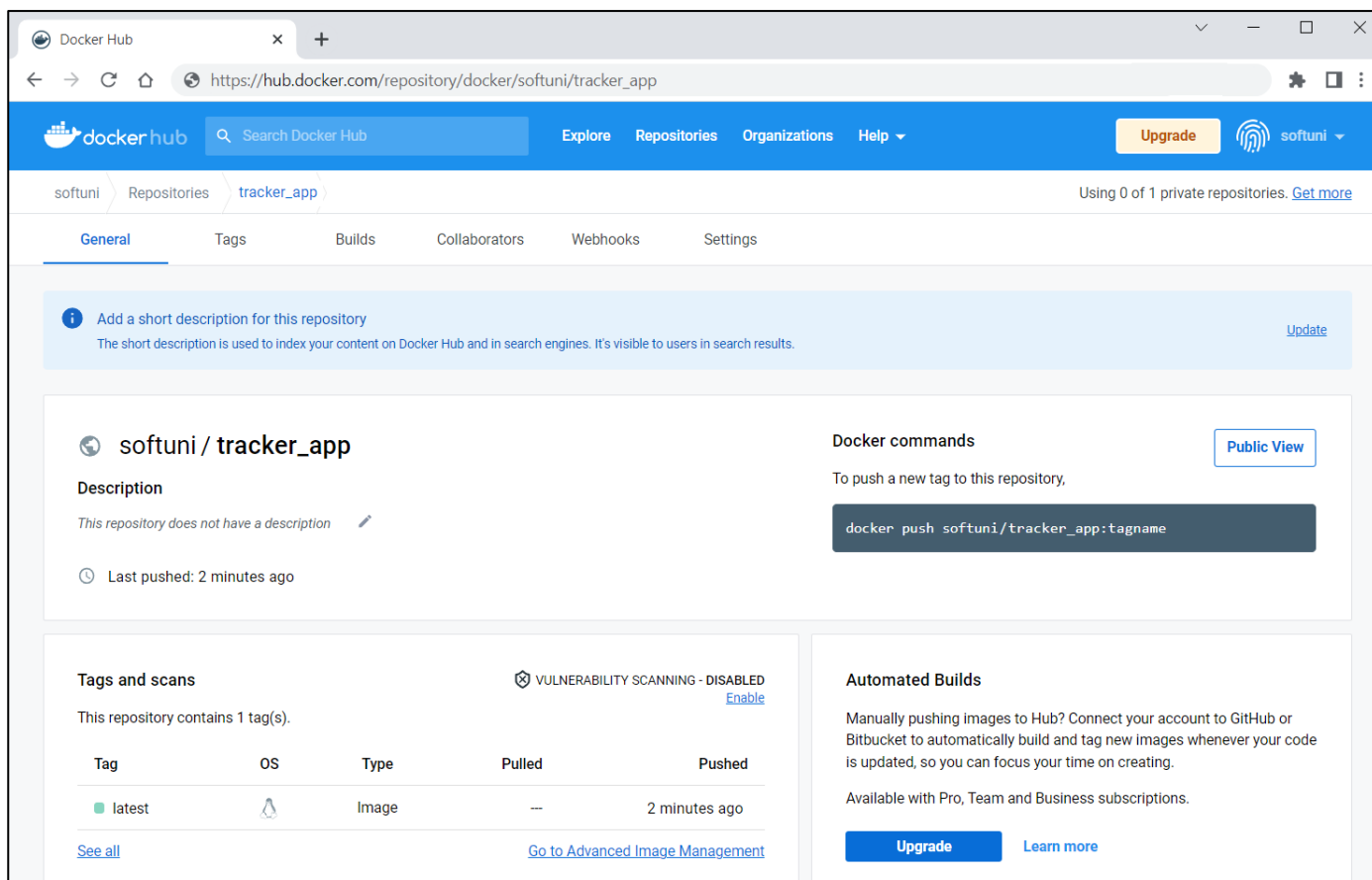
You're provided with **its files** it in the **resources**, together with a **Dockerfile** which runs the **app on NGINX server**:



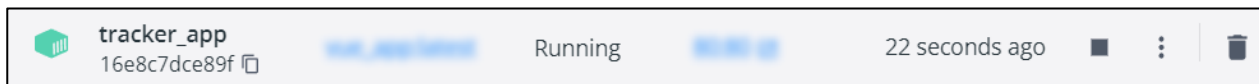
First, build a **custom image {username}/tracker\_app** from the **given Dockerfile**:



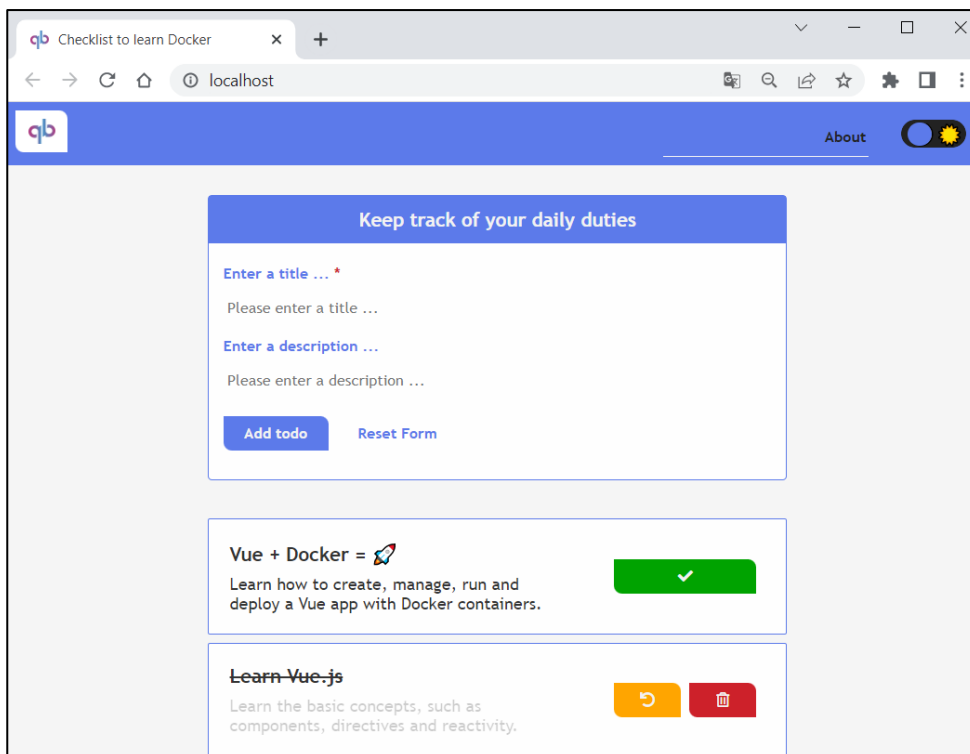
**Push the image to Docker Hub:**



Then, use it to **run the Vue app in a container** (think about the **internal port** on which the app works):



Finally, **access the app** from the **browser** – it should be working:



## 8. TaskBoard App: Connect Containers in a Network

In this task, we will connect the **TaskBoard ASP.NET 6 MVC app** to its **SQL Server database**. They will both be in **separate Docker containers**, which will be **connected to a common network** and this will allow them to **communicate with each other**.

After we have an **image for the TaskBoard app** and know how to **run a SQL Server container**, let's learn how to **create and connect them to a network**.

### Step 1: Create a Network

Create a **network** with name **taskboard\_network**:

```
PS C:\Users\PC> docker network create taskboard_network
ea37c2c052b7f9553edab6d933ecf22fadcba3f378b5c7b63d10b2f0b2ef0ce1
```

You can see **all networks** with:

```
PS C:\Users\PC> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
55c22e9cd827        bridge             bridge              local
aa72d250011b        host               host                local
ea37c2c052b7        taskboard_network  bridge              local
```

### Step 2: Create and Connect a SQL Server Container

Now we want to **run a SQL Server container** inside our **taskboard\_network network**. You already know how to **write the command** for **creating the Docker container**, but we should **add some more options** to it:

- Use the **--network** option with the **name of the network** you want to **connect to**

- Use the **--name** option to **set a name of the container**. This is **important** as other containers use this name to recognize it in the network
- Use the **--rm** option to **automatically remove the container** when it **exits** (not mandatory)

The command is the following and **creates a container in the network**:

```
PS C:\Users\PC> docker run `
>> -e ACCEPT_EULA=Y `
>> -e MSSQL_SA_PASSWORD=yourStrongPassword12# `
>> -p 1433:1433 `
>> -v sqldata:/var/opt/mssql `
>> --rm --network taskboard_network --name sqlserver `
>> -d mcr.microsoft.com/mssql/server
3cbf315de9e42c7bdba90fb0e8d9033d3b4c4882131147431a1de99e436518f7
```

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
sqlserver 3cbf315de9e4	<a href="https://mcr.microsoft.com/mssql/server:latest">mcr.microsoft.com/mssql/server:latest</a>	Running	<a href="#">1433:1433</a>	3 minutes ago	⏏ ⋮ 🗑

The **database container** is now **working**.

### Step 3: Create and Connect a TaskBoard App Container

Our next step is to run the **TaskBoard app** in a **container** in the **same network**.

Before that, however, we should **change the database connection string** of the app, so that it can **connect to the SQL Server database** we created. Open the **appsettings.json** file of the "TaskBoard.WebApp" project and **modify it** according to the following requirements:

- **Server** should be **sqlserver**
- **Database name** is of your choice
- **User Id** should be **sa** (the default database system admin user)
- **Password** should be the **admin password** we set in the previous command – **yourStrongPassword12#**
- Allow **multiple connections**

The **connection string** should be the following:

```
appsettings.json*
https://json.schemastore.org/appsettings.json
1 {
2   "ConnectionStrings": {
3     "DefaultConnection": "Server=sqlserver;Database=MyDB;User Id=sa;
4     Password=yourStrongPassword12#;MultipleActiveResultSets=true;"
5   },
6   "Logging": {
7     "LogLevel": {
8       "Default": "Information",
9       "Microsoft": "Warning",
10      "Microsoft.Hosting.Lifetime": "Information"
11    }
12  },
13  "AllowedHosts": "*"
14 }
```

We should **build the app image** again, so that **changes are reflected**:

```
softuni\objects\TaskBoard> docker build . -f ./TaskBoard.WebApp/Dockerfile -t softuni/taskboard_app
[+] Building 0.3s (19/19) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 32B 0.0s
=> [internal] load .dockerignore 0.0s
```

Now you are ready to **run the app**:

```













PS D:\Projects\TaskBoard> docker run
>> -p 5000:80 --rm
>> --name web_app
>> --network taskboard_network
>> softuni/taskboard_app
warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]
      Storing keys in a directory '/root/.aspnet/DataProtection-Keys' that may not be persi
sted outside of the container. Protected data will be unavailable when container is destroy
ed.
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
      No XML encryptor configured. Key {6d7ea3e8-aad7-4d05-9e41-b3d305c4dae1} may be persis
ted to storage in unencrypted form.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://[::]:80
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /app/

```

Our **Web app** is running in a **Docker container**, too.

## Step 4: Containers Together

These are **our containers**:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
 <b>web_app</b> 3cbf315de9e4 	<a href="#">softuni/taskboard_app:latest</a>	Running	<a href="#">5000:80</a> 	40 seconds ago	  
 <b>sqlserver</b> dd5d8e29589d 	<a href="#">mcr.microsoft.com/mssql/server:latest</a>	Running	<a href="#">1433:1433</a> 	4 minutes ago	  

You can also see that they are **both connected** to our **taskboard\_network network** when inspecting it:

```

PS C:\Users\PC> docker network inspect taskboard_network
[
  {
    "Name": "taskboard_network",
    "Id": "ea37c2c052b7f9553edab6d933ecf22fadcbab3f378b5c7b63d10b2f0b2ef0ce1",
    "Created": "2022-11-29T12:04:00.25603721Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.20.0.0/16",
          "Gateway": "172.20.0.1"
        }
      ]
    }
  }
],

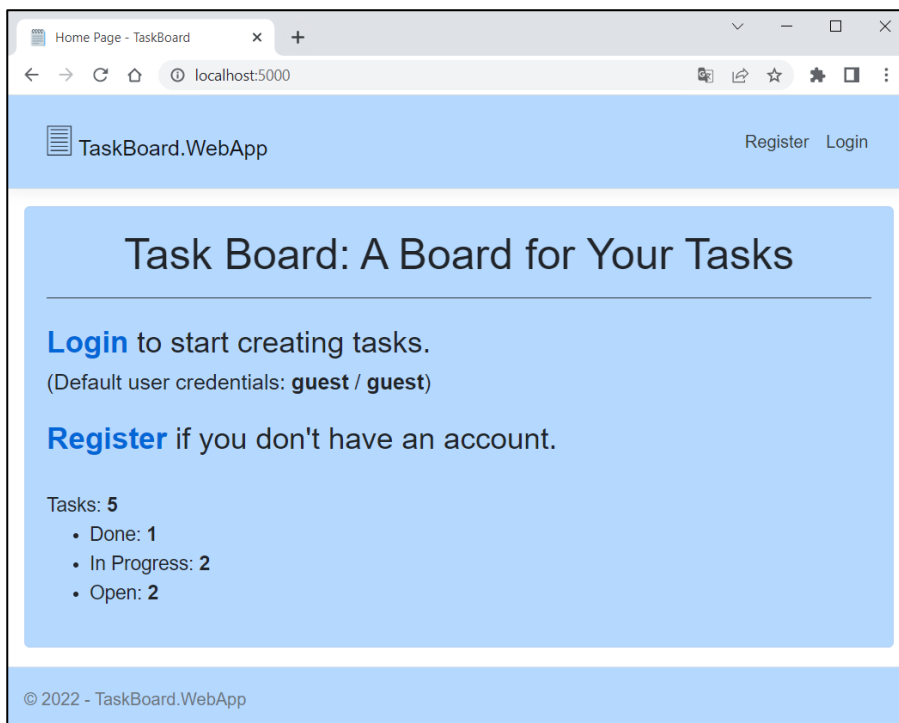
```

```

"Internal": false,
"Attachable": false,
"Ingress": false,
"ConfigFrom": {
  "Network": ""
},
"ConfigOnly": false,
"Containers": {
  "9e4fa0a11b6f06883cf05dbaf0bbb5e428ac32b0b0f1276f206235b0271be476": {
    "Name": "web_app",
    "EndpointID": "b7d11da3bd0011003de365cd8efca42b6ef2eac0091abddd670a0fa227e2b7e8",
    "MacAddress": "02:42:ac:14:00:03",
    "IPv4Address": "172.20.0.3/16",
    "IPv6Address": ""
  },
  "3cbf315de9e451bce23369d28c843736a7e498f953a9842c8395a2ea37f44a53": {
    "Name": "sqlserver",
    "EndpointID": "d740db32c433f8d6952c90eec89c46fccc06ed347879a3c1f7f48a17deb78540",
    "MacAddress": "02:42:ac:14:00:02",
    "IPv4Address": "172.20.0.2/16",
    "IPv6Address": ""
  }
},
"Options": {},
"Labels": {}
}

```

And when you go to **<http://localhost:5000>** you have the **fully working TaskBoard Web app** with a **database**:



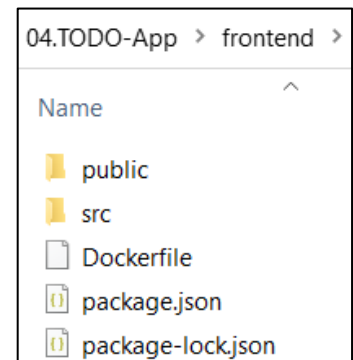
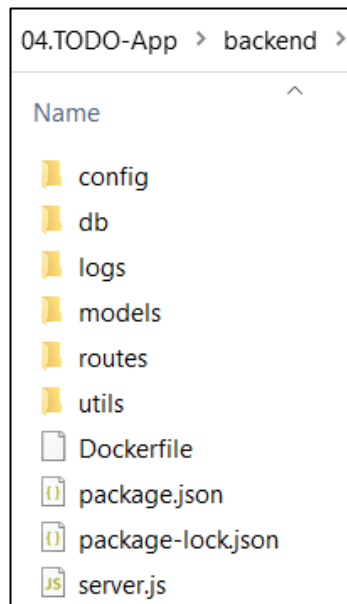
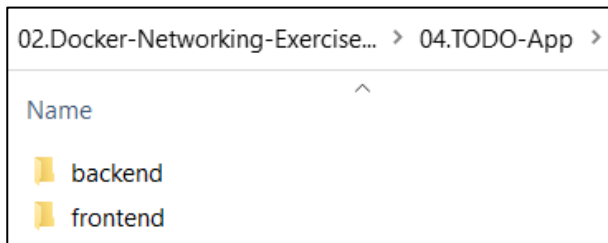
The app should be working – **test it by yourself**. In addition, you can try to **stop the app container** and **create a new one**, connected to the same **taskboard\_network** network and you should see that the **database is preserved** because it is on a **SQL Server** container.

That is how you can **connect containers in a common network** and **use them together** to run **multi-container apps**.

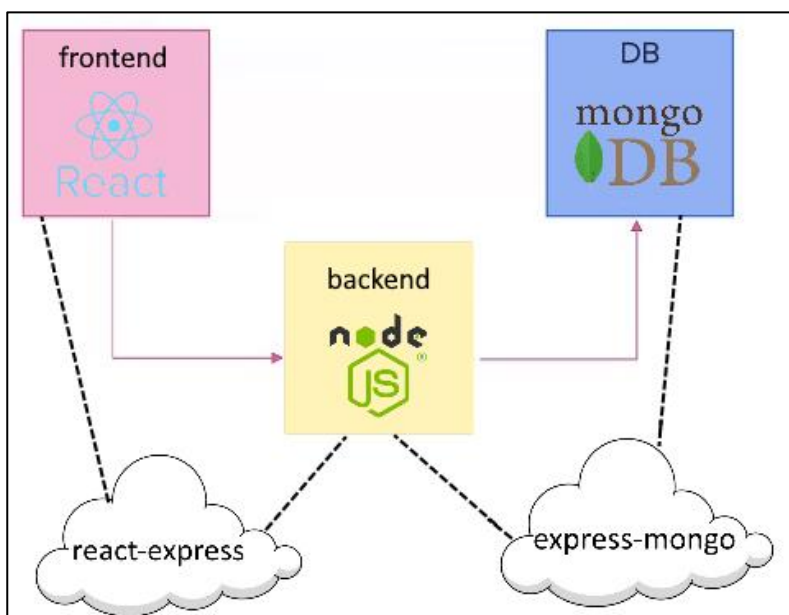
## 9. TODO App

The **TODO app** (provided in the **resources**) is a simple app for **adding tasks**, which you should **Dockerize**:





















It is a **React application** with a **NodeJS backend** and a **MongoDB database**. You should **create the separate Docker containers** and **connect them in two networks** as shown below to make the **three containers work together**:



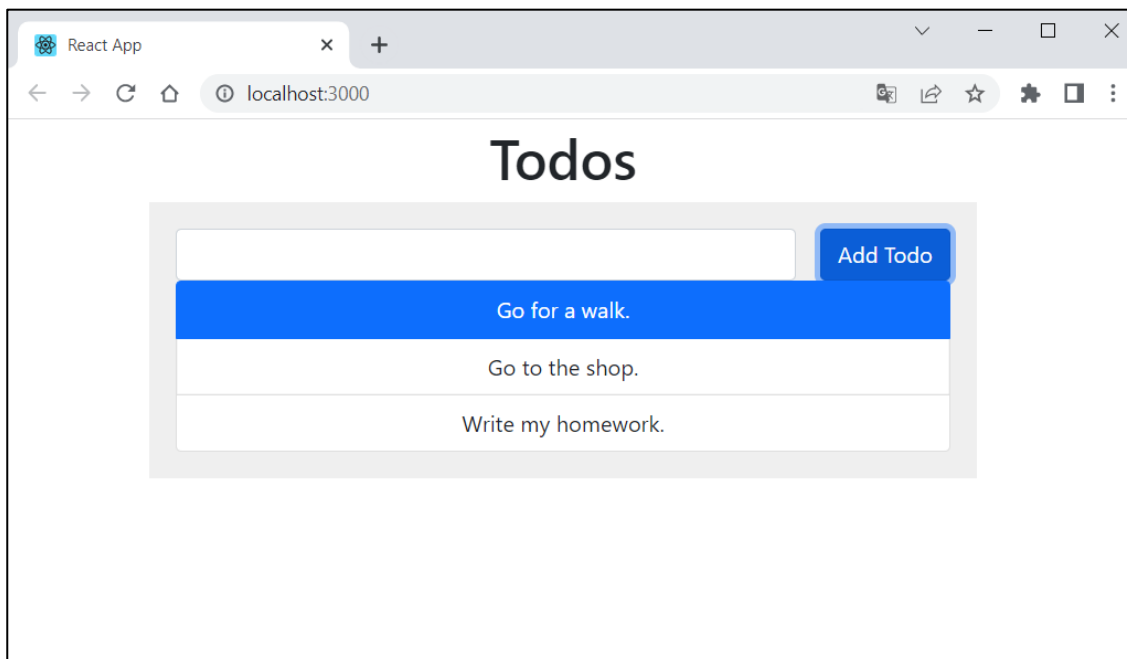
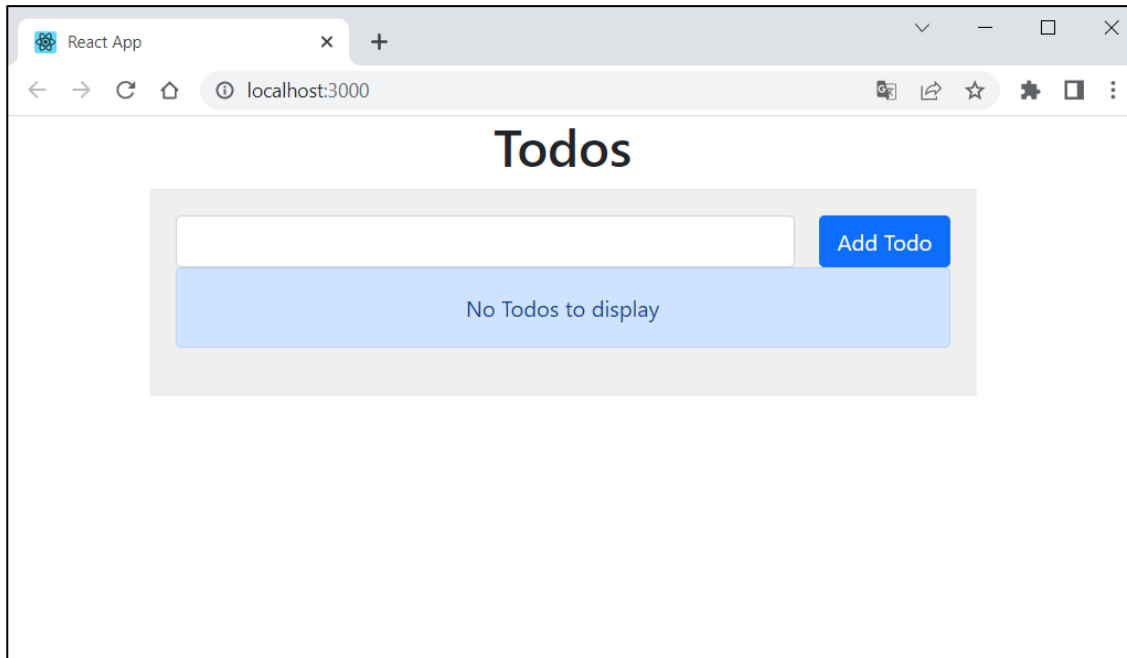
## Requirements

- **Name** the three containers "**frontend**", "**backend**" and "**mongo**"
- **Build images** from the **provided Dockerfiles** for the **frontend** and **backend** services
- Use the **latest image** for **MongoDB** from **Docker Hub**
- Expose the **frontend service** on port **3000** (see on which port the app works by yourself)
- **Mount** the following **host directories** as **volumes**:
  - For **mongo** service: **./data:/data/db**
- Connect the **frontend** and **backend** services to the **react-express network** and the **backend** and **mongo** services to the **express-mongo network**

These are the **containers** that should appear:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
 <b>backend</b> 18899a81f6bf 	<a href="#">backend_image:latest</a>	Running		10 minutes ago	  
 <b>frontend</b> 56a93fa76104 	<a href="#">frontend_image:latest</a>	Running	<a href="#">3000:3000</a> 	10 minutes ago	  
 <b>mongo</b> b56ab73b081f 	<a href="#">mongo:latest</a>	Running		9 minutes ago	  

When ready, you should be able to **add tasks** to the **TODO list in the app**:



## Hints

- Use the **docker build** command to **build the frontend and backend services images** in their **corresponding folders**.
- Create the **two networks**.

- Run the containers following the **requirements** and using the images you created and the `mongo:latest` image.
- To mount a host directory as **volume**, do it with "`-v {host directory path}:{container directory}`".
- As you may have seen, you **cannot run a container in two networks** with the `docker run` command. For this reason, you should **add the container to a network after the container creation** with the `docker network connect` command.

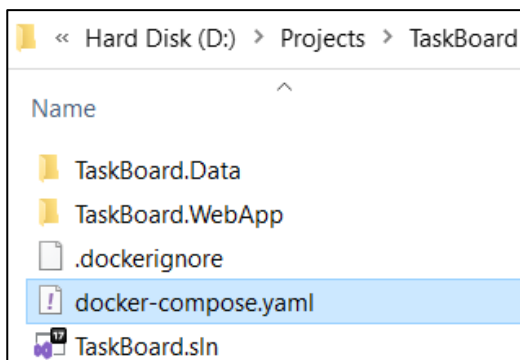
## 10. TaskBoard App: Orchestrating Containers with Docker Compose

In this task, we will make our TaskBoard app and SQL Server database containers work together with Docker Compose.

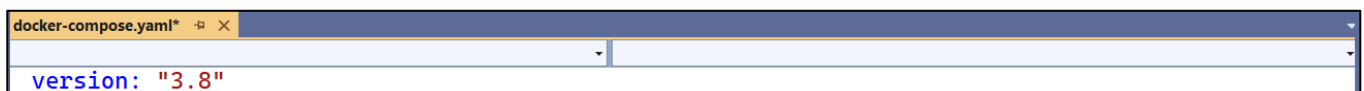
### Step 1: Build a YAML File

Our first job is to **build a Docker Compose YAML file**. It will **replace the separate docker run commands** for the **two containers** and combine them into a **single file**.

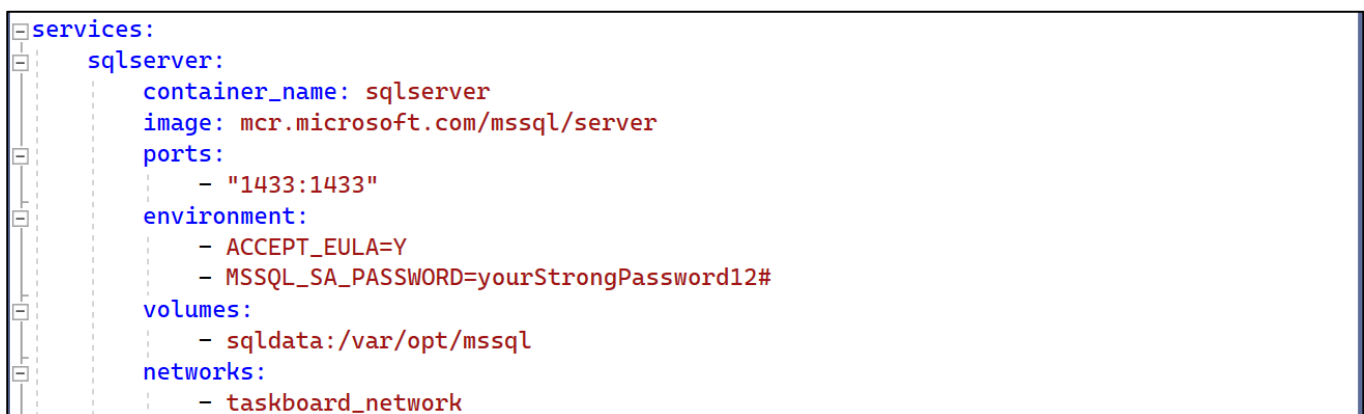
Go to your "TaskBoard" solution directory and add a new text file `docker-compose` with an `.yaml` extension:



Open the file with any editor and let's write it. Start with the **file version** – choose the latest one:



Next, we will **describe the steps for each service** (container). Start with the **database service**: **set the container name, image, ports, environment variables, volume** and a **custom network** – it is all from the `docker run` command we ran for the container, but in a **different format**. It should look like this:



Now write the **service for the Web app**, which should contain a **container name**, **Dockerfile path**, **ports**, and the **same custom network**. It may also be set to **restart on fail**:

```
web_app:
  container_name: web_app
  build:
    dockerfile: ./TaskBoard.WebApp/Dockerfile
  ports:
    - "5000:80"
  restart: on-failure
  networks:
    - taskboard_network
```

Finally, you should **point out the volumes and network** you used in the services. You have a single volume and a single network in our case:

```
volumes:
  sqldata:
networks:
  taskboard_network:
```

Save the file and open a CLI to execute commands on the file.

## Step 2: Run the YAML File

First, navigate to the **folder of the docker-compose.yaml file** and **build all images**, using the **docker compose build** command:

```
PS D:\Projects\TaskBoard> docker compose build
[+] Building 13.2s (19/19) FINISHED
=> [internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 875B 0.1s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 382B 0.0s
=> [internal] load metadata for mcr.microsoft.com/dotnet/sdk:6.0 0.4s
=> [internal] load metadata for mcr.microsoft.com/dotnet/aspnet:6.0 0.0s
=> [base 1/2] FROM mcr.microsoft.com/dotnet/aspnet:6.0 0.0s
=> [build 1/8] FROM mcr.microsoft.com/dotnet/sdk:6.0@sha256:3dfedfc30f95c93c3e1d41a 0.0s
=> [internal] load build context 0.7s
=> => transferring context: 4.45MB 0.7s
=> CACHED [build 2/8] WORKDIR /src 0.0s
=> CACHED [build 3/8] COPY [TaskBoard.WebApp/TaskBoard.WebApp.csproj, TaskBoard.web 0.0s
=> CACHED [build 4/8] COPY [TaskBoard.Data/TaskBoard.Data.csproj, TaskBoard.Data/] 0.0s
=> CACHED [build 5/8] RUN dotnet restore "TaskBoard.WebApp/TaskBoard.WebApp.csproj" 0.0s
=> [build 6/8] COPY . 0.0s
=> [build 7/8] WORKDIR /src/TaskBoard.WebApp 0.0s
=> [build 8/8] RUN dotnet build "TaskBoard.WebApp.csproj" -c Release -o /app/build 5.8s
=> [publish 1/1] RUN dotnet publish "TaskBoard.WebApp.csproj" -c Release -o /app/pu 5.8s
=> CACHED [base 2/2] WORKDIR /app 0.0s
=> CACHED [final 1/2] WORKDIR /app 0.0s
=> CACHED [final 2/2] COPY --from=publish /app/publish . 0.0s

=> exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:2c94c77ed0b624990363db26840c7e9ee6d00b836694bdfc63b1285f 0.0s
=> => naming to docker.io/library/taskboard-web_app 0.0s
```

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

Then, run the containers together with Docker Compose:

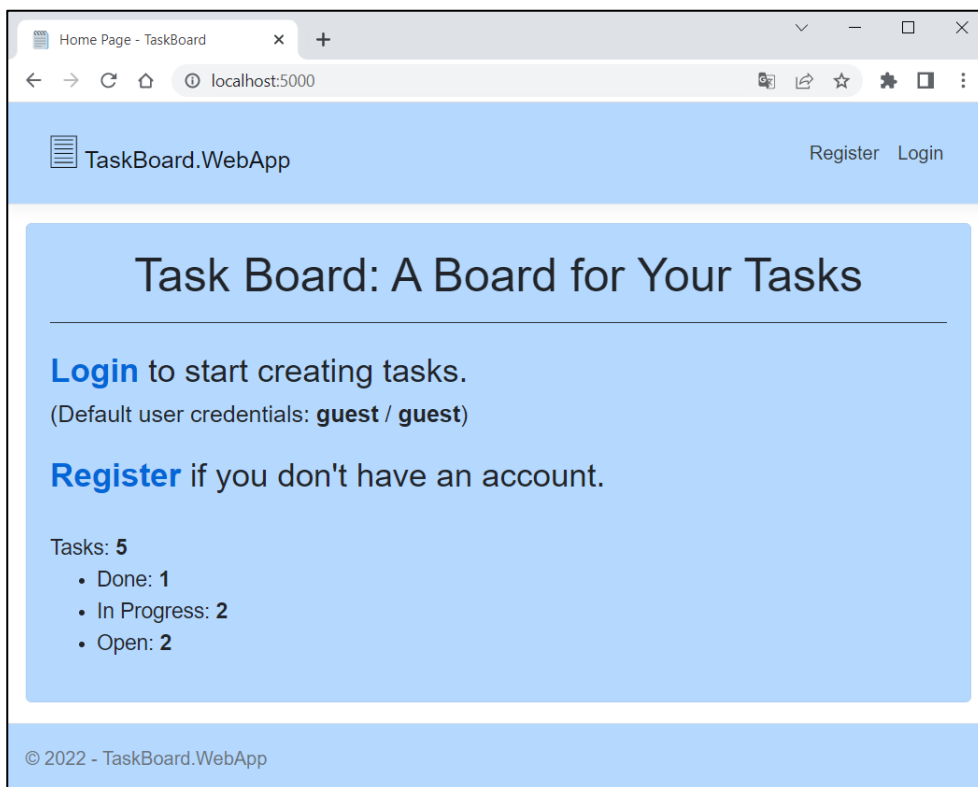
```
PS D:\Projects\TaskBoard> docker compose up
[+] Running 4/3
- Network taskboard_network          Crea...          0.8s
- volume "taskboard_sqldata"        Created          0.0s
- Container sqlserver                Created          0.1s
- Container web_app                  Created          0.0s
Attaching to sqlserver, web_app
```

You can see that **both database and Web app** containers are set and **running** in our **custom network**:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
> taskboard	-	Running (2/2)			■ ⋮ 🗑

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
▼ taskboard	-	Running (2/2)			■ ⋮ 🗑
sqlserver f7d0c71fe680 📄	<a href="https://mcr.microsoft.com/mssql/server:latest">mcr.microsoft.com/mssql/server:latest</a>	Running	<a href="#">1433:1433</a> 🔗	3 minutes ago	■ ⋮ 🗑
web_app c6444511cf7a 📄	<a href="#">taskboard-web_app:latest</a>	Running	<a href="#">5000:80</a> 🔗	3 minutes ago	■ ⋮ 🗑

And they are **working in the browser**, too:



Now you can **stop the containers** as we will have to run them again after a while.

### Step 3: Debug the Web App

Let's see how we are supposed to **debug the TaskBoard Web app** while it is **running inside a container**.

To do this, we should first **make changes to the Dockerfile** – it should have **Debug**, not **Release** configurations:

```

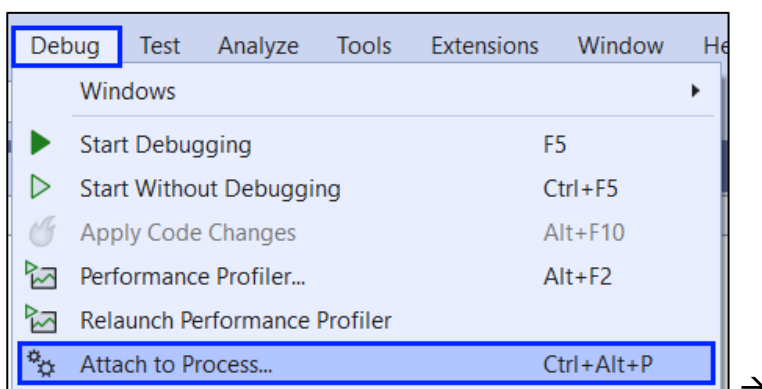
1  #See https://aka.ms/containerfastmode to understand how Visual Studio uses this Dockerfile
2
3  FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
4  WORKDIR /app
5  EXPOSE 80
6  EXPOSE 443
7
8  FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
9  WORKDIR /src
10 COPY ["TaskBoard.WebApp/TaskBoard.WebApp.csproj", "TaskBoard.WebApp/"]
11 COPY ["TaskBoard.Data/TaskBoard.Data.csproj", "TaskBoard.Data/"]
12 RUN dotnet restore "TaskBoard.WebApp/TaskBoard.WebApp.csproj"
13 COPY . .
14 WORKDIR "/src/TaskBoard.WebApp"
15 RUN dotnet build "TaskBoard.WebApp.csproj" -c Debug -o /app/build
16
17 FROM build AS publish
18 RUN dotnet publish "TaskBoard.WebApp.csproj" -c Debug -o /app/publish /p:UseAppHost=false
19
20 FROM base AS final
21 WORKDIR /app
22 COPY --from=publish /app/publish .
23 ENTRYPOINT ["dotnet", "TaskBoard.WebApp.dll"]

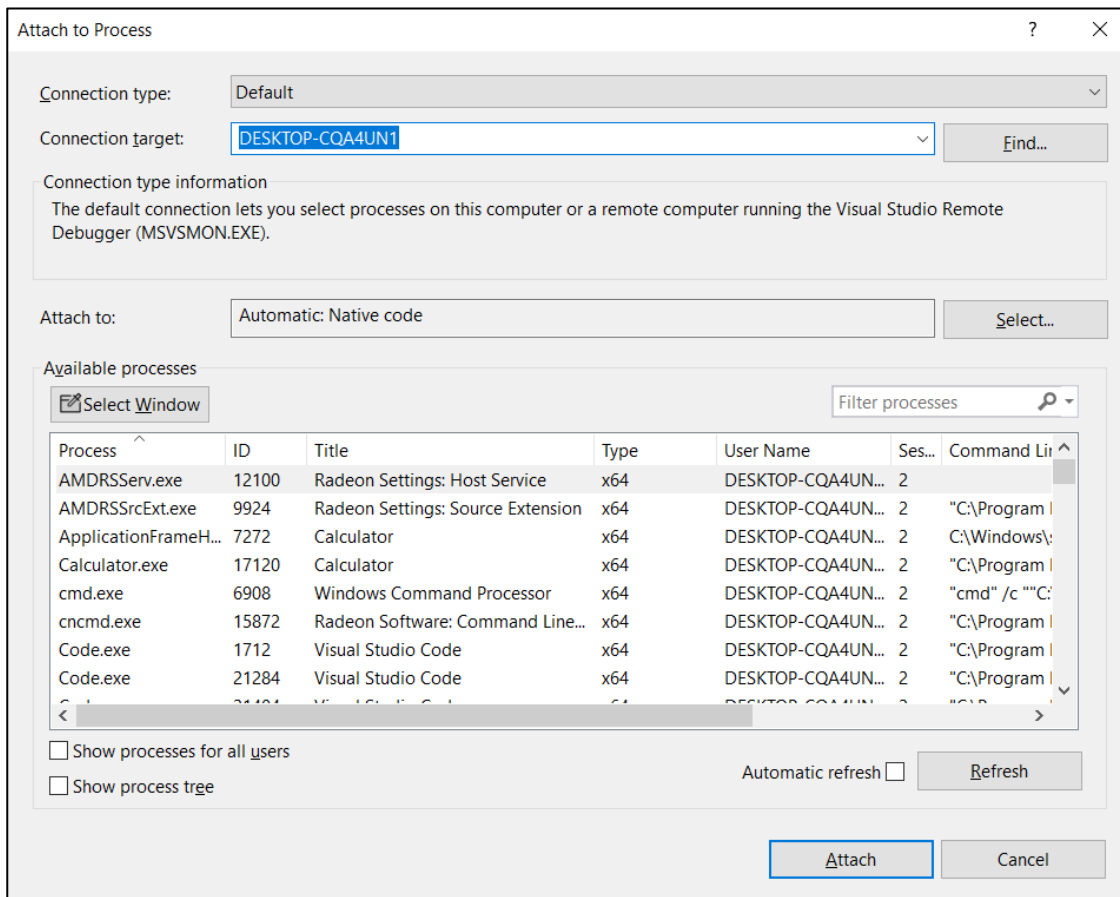
```

Save the file, build images again with **docker compose build** and run new containers with **docker compose up**:

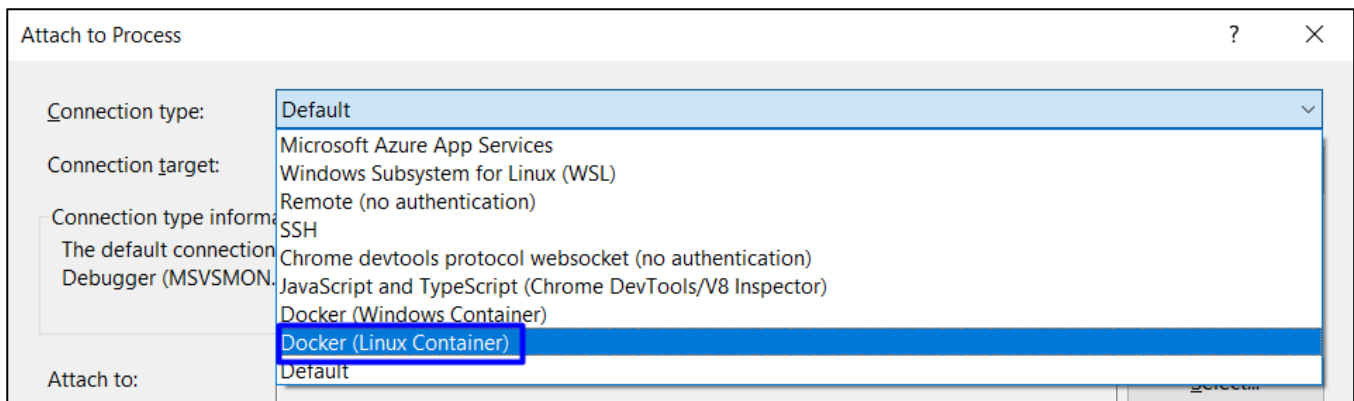
NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
taskboard	-	Running (2/2)			
sqlserver a44647bb6f9b	<a href="https://mcr.microsoft.com/mssql/server:latest">mcr.microsoft.com/mssql/server:latest</a>	Running	1433:1433	2 minutes ago	
web_app 74d2b9a815ea	<a href="#">taskboard-web_app:latest</a>	Running	5000:80	2 minutes ago	

Now, in **Visual Studio**, go to **[Debug] → [Attach to Process...]** or use the **[Ctrl]+[Alt]+[P]** keys:

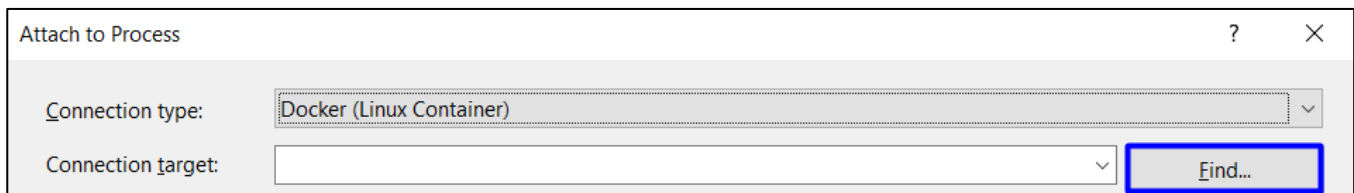




Change the connection type to [Docker (Linux Container)]:

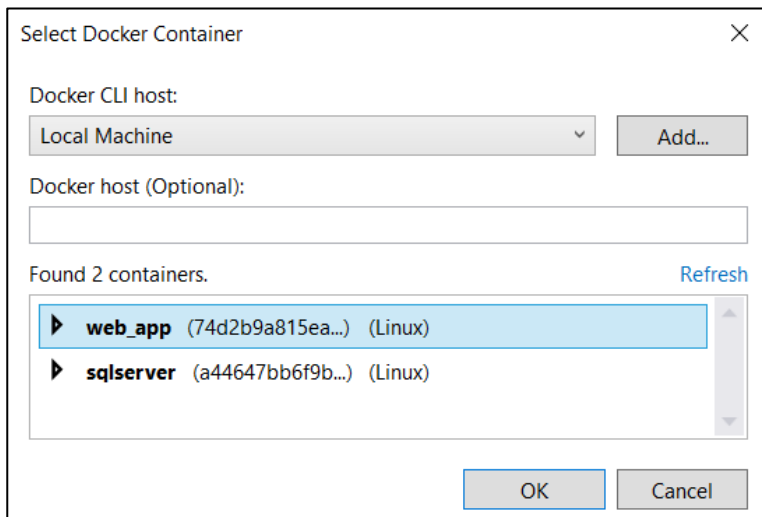


And click on the [Find] button to choose a connection target:

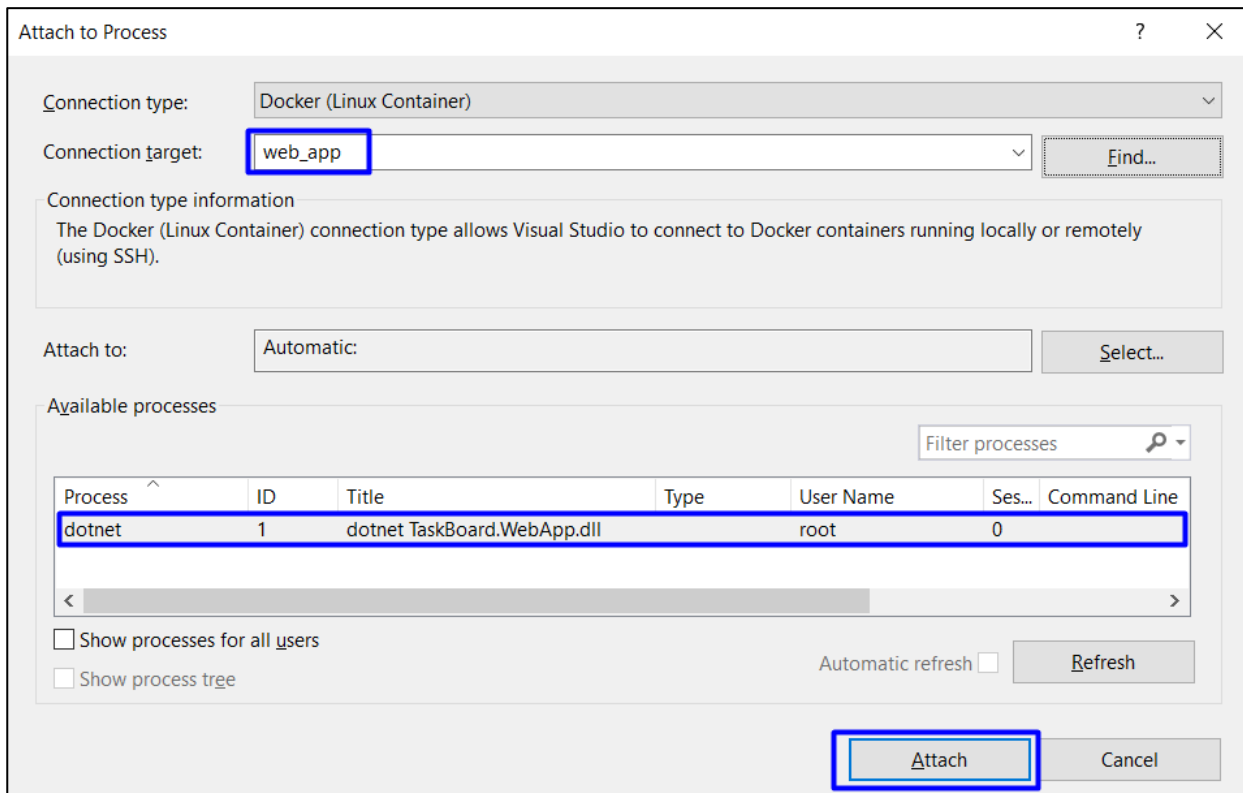


On the next window, select the **Web app container** and click on [OK]:

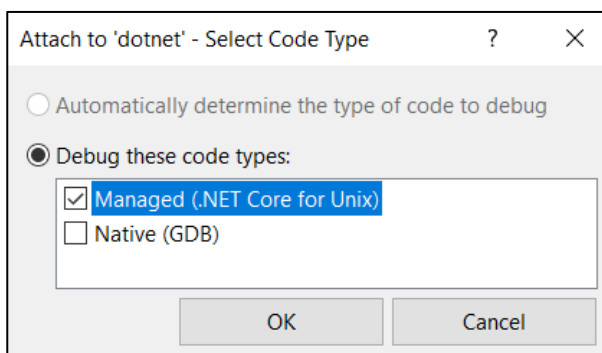




The **correct container** is chosen, so you should only click on the **[Attach]** button:



On the final step, choose the **[Managed (.NET Core for UNIX)]** code type and click **[OK]**:



The **debug adapter is launched** and we are in **debug mode**:

Containers	
Containers	Images
taskboard	
sqlserver	
web_app	
Environment	
Name	Value
ACCEPT_EULA	Y
CONFIG_EDGE_BUILD	
MSSQL_RPC_PORT	135
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SA_PASSWORD	yourStrongPassword12#

Now you can **put a breakpoint**, **refresh the app** in the browser and see if the **breakpoint will be reached**:

```

15
16 0 references
17 public HomeController(ApplicationDbContext context)
18 {
19     this.dbContext = context;
20 }
21
22 0 references
23 public IActionResult Index()
24 {
25     var taskBoards = this.dbContext
26     .Boards
27     .Select(b => b.Name)
28     .Distinct();
29
30     var tasksCounts = new List<HomeBoardModel>();
31     foreach (var boardName in taskBoards)
32     {
33     }
34 }

```

```

15
16 0 references
17 public HomeController(ApplicationDbContext context)
18 {
19     this.dbContext = context;
20 }
21
22 0 references
23 public IActionResult Index()
24 {
25     var taskBoards = this.dbContext
26     .Boards
27     .Select(b => b.Name)
28     .Distinct();
29
30     var tasksCounts = new List<HomeBoardModel>();
31     foreach (var boardName in taskBoards)
32     {
33     }
34 }

```

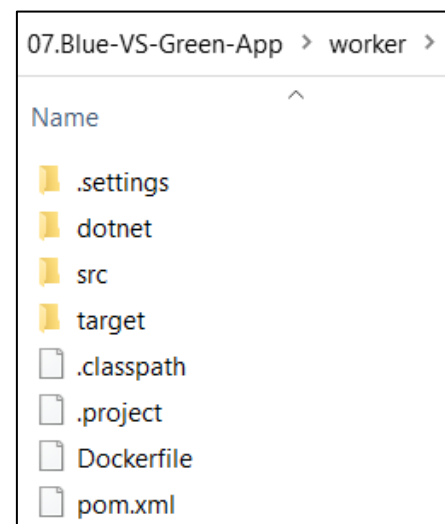
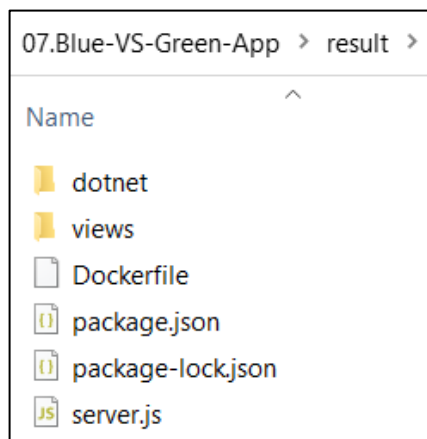
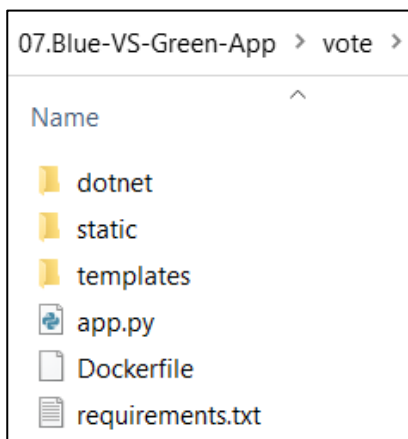
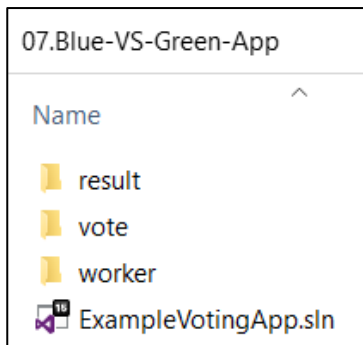
You know how to **debug the container app** if you need to. Finally, you can use the following command to **remove everything together** – the containers, images, volumes, etc. (without the network):

```
PS D:\Projects\TaskBoard> docker compose down --rmi all --volumes
[+] Running 6/6
- Container web_app           Removed           0.6s
- Container sqlserver         Remove...        1.0s
- Volume taskboard_sqldata    R...            0.0s
- Image mcr.microsoft.com/mssql/server:latest Removed          0.6s
- Image taskboard-web_app:latest Removed          0.0s
- Network taskboard_network   Removed          0.8s
```

After this task, we now know how to **work with custom images, Dockerfiles, networks** and **Docker Compose**. We also know how to **run a multi-container ASP.NET Core + SQL Server app**.

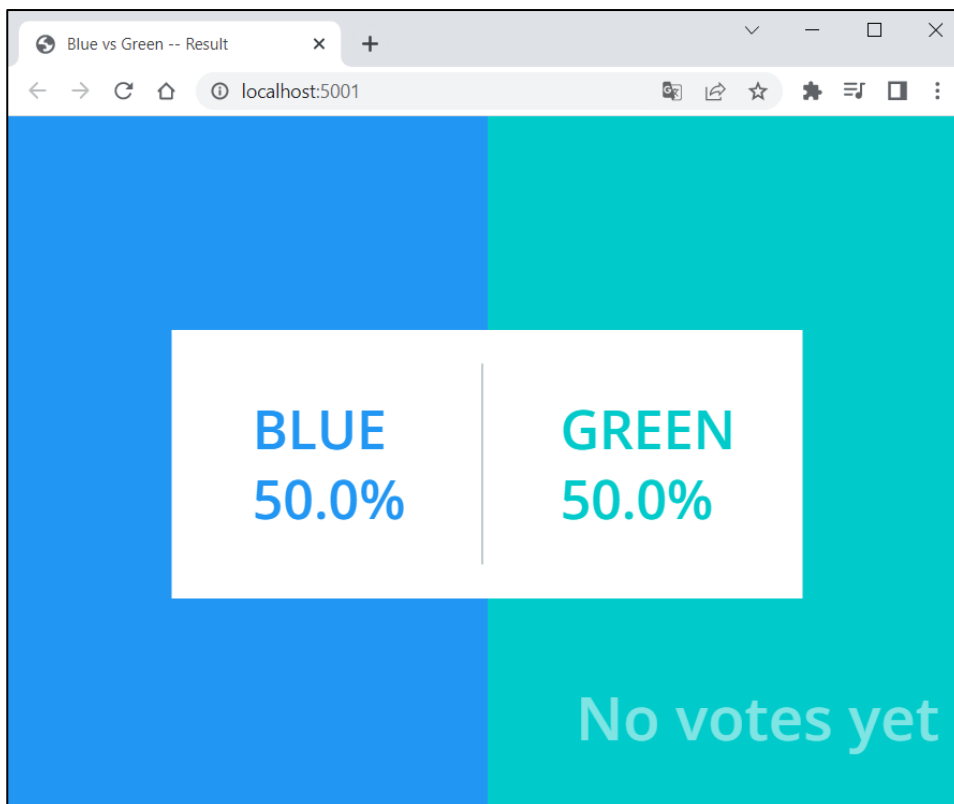
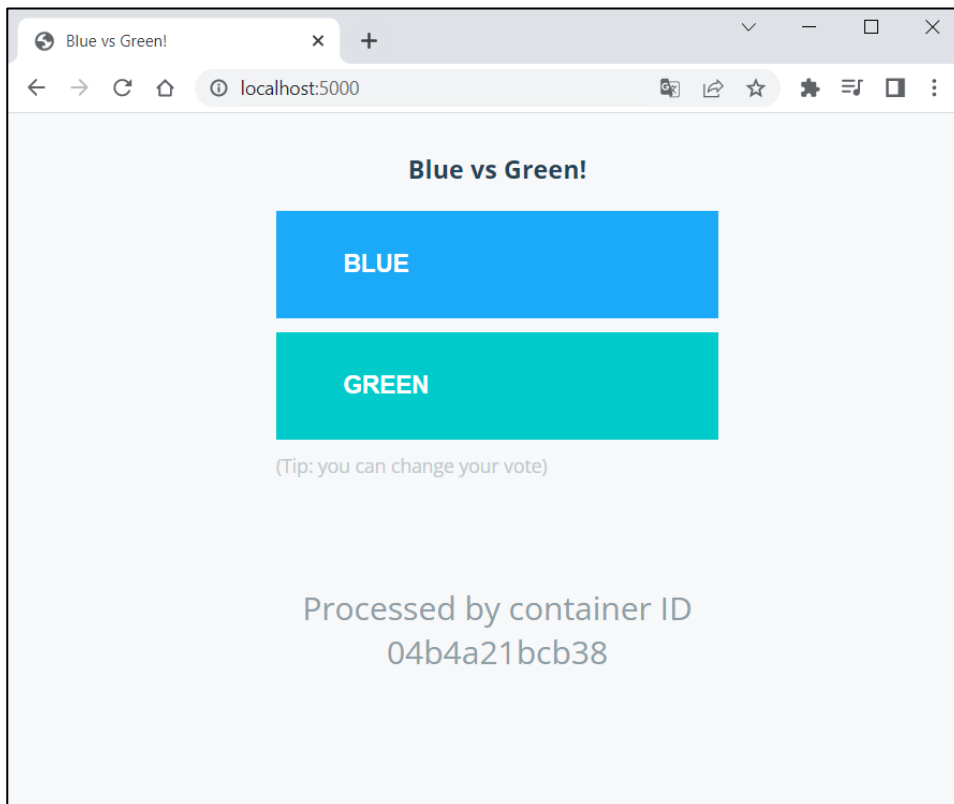
## 11. Blue VS Green App

The **"Blue VS Green" app** (provided in the **resources**) is a **simple voting app**, which you should **run with Docker Compose**:



Note that the **Dockerfiles** for the **voting** and **worker apps** you see here are **empty**.

It provides an **interface for a user to vote** and another **interface to show the results**:



You can **vote** and then **change your vote** and this will **make changes in the results**.

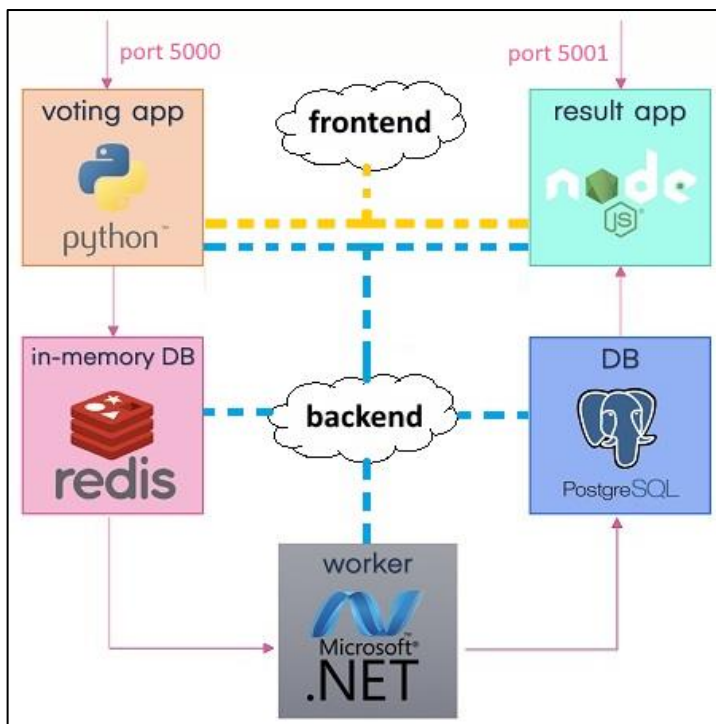
Your task is to **fill in the missing instructions** in the **Dockerfiles** and **run the app with Docker Compose**:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
example-voting-app	-	Running (5/5)			
result-1 b6c35fbf752b	<a href="#">example-voting-app-result:latest</a>	Running	<a href="#">5001:80</a>	10 seconds ago	
vote-1 02445ab49298	<a href="#">example-voting-app-vote:latest</a>	Running	<a href="#">5000:80</a>	9 seconds ago	
worker-1 eb369ea22a31	<a href="#">example-voting-app-worker:latest</a>	Running		10 seconds ago	
db-1 c2bfef26274d	<a href="#">postgres:latest</a>	Running		10 seconds ago	
redis-1 d1e6e5bc1185	<a href="#">redis:latest</a>	Running		10 seconds ago	

When ready, your app should be working.

## Architecture

The app has the following architecture:



And consists of:

- A **voting app** – a **Web app**, developed in **Python**, which provides an **interface** for the **user to choose between two options** (blue and green)
- An **in-memory database** on **Redis**, which **stores the user's vote** from the **voting app**
- A **worker app** on **.NET**, which **processes the new vote by updating the persistent database**
- A **persistent PostgreSQL database**, which has a **single table with the number of votes** for each category (blue and green)
- A **NodeJS Web interface** (app), which **displays the result of the votes** from the PostgreSQL database

## Requirements

- Use the **latest images** for **PostgreSQL** and **Redis** from **Docker Hub** and use the **filled-in Dockerfiles** for the **voting**, **result** and **worker app**
- **PostgreSQL container** needs **user and password** for login: see how to set them in the **image's documentation**
- The **voting app** should be accessed on **localhost:5000** and the **result app** – on **localhost:5001**
- Network traffic should be separated to **two networks** – **frontend** and **backend**:
  - The **frontend** network is for the **users' traffic**. Connect the **voting app** and the **result app** to it
  - The **backend** network is for the traffic within the app. It connects **all app components**
- Run the **voting** and **result apps** in the **containers**
- Use **volumes** for the **voting and result apps** and the **db container**

## Hints

Find out how to write the **Dockerfiles** you need from the **Docker Documentation**: <https://docs.docker.com>.

For the **voting app**, write a **Dockerfile** for **building a Python image**:

```
Dockerfile
# Using official python runtime base image
FROM python:3.9-slim

# Set the application directory
WORKDIR /app

# Install our requirements.txt
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy our code from the current folder to /app inside the container
COPY . .

# Make port 80 available for links and/or publish
EXPOSE 80

# Define our command to be run when launching the container
CMD ["python", "main.py"]
```

For the **worker app**, you should **build a .NET image**:

```
Dockerfile
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 as builder

# Create a working directory
WORKDIR /Worker

# Copy the .csproj file and restore
COPY *.csproj .
RUN dotnet restore

# Copy source files to the image
COPY . .

# Build the project
RUN dotnet publish -c Release -o:out

# Specify app image
FROM mcr.microsoft.com/dotnet/core/runtime:3.1

# Specify working directory for this stage
WORKDIR /Worker

# Tell Docker what command to run when our image is executed inside a container
CMD ["dotnet", "Worker.dll"]

# Copy the /out directory from the build stage into the runtime image
COPY --from=builder /Worker/out .
```

Finally, write the **docker-compose.yaml** file. This is a sample of how it may look like:

```
docker-compose.yml
version: "3.8"

services:
  vote:
    build: .
    command: python app.py
    volumes:
      - .:/app
    ports:
      - 8080:8080
    networks:
      - network
  result:
    build: .
    command: node server.js
    networks:
      - network
  worker:
    build: .
    command: python worker.py
    networks:
      - network
  redis:
    image: redis:latest
    networks:
      - network
  db:
    image: postgres:13
    environment:
      POSTGRES_DB: postgres
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    networks:
      - network
```

Finally, **run the app** and see if it works and **voting is possible** and **reflected in results** as expected.