Exercises: Docker Compose

Exercises for the "Containers and Clouds" course @ SoftUni

1. *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.

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	mariadb:latest	Running		29 seconds ago	
	mariadb_client c2833bcfc58e	mariadb:latest	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","Endp
ointID":"16c7e2402074935da3fc47d9094ace52f7830c23bf37b1885a26c8e0b780a90a","MacAddress":"02:42:ac:
1d:00:02","IPv4Address":"172.29.0.2/16","IPv6Address":""},"c2833bcfc58e826cc5d0a16f31930e273164a4b
467310dd94866353d09ab3708":{"Name":"mariadb_client","EndpointID":"fc2cd8bc168769ad12aaf96a54498be4
37dc39aa9ff4aa1d2496ba1cad9cd0e8","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)]>
```

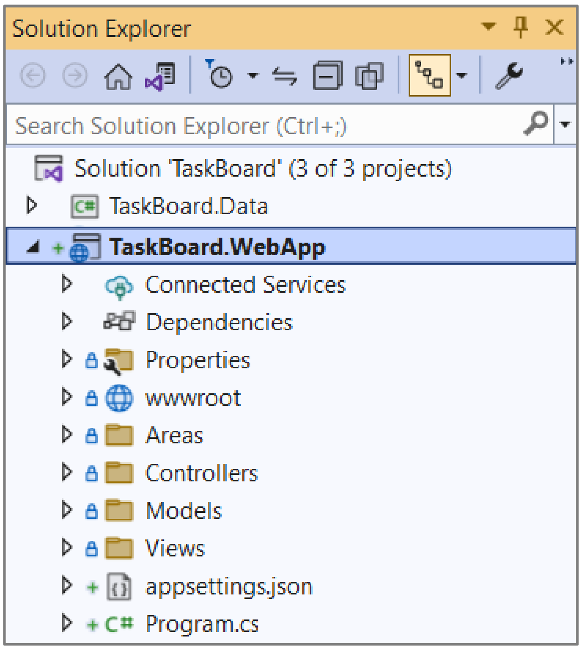
And you should be able to **run commands** on it:

```
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.

2. TaskBoard App: Building a Custom Image

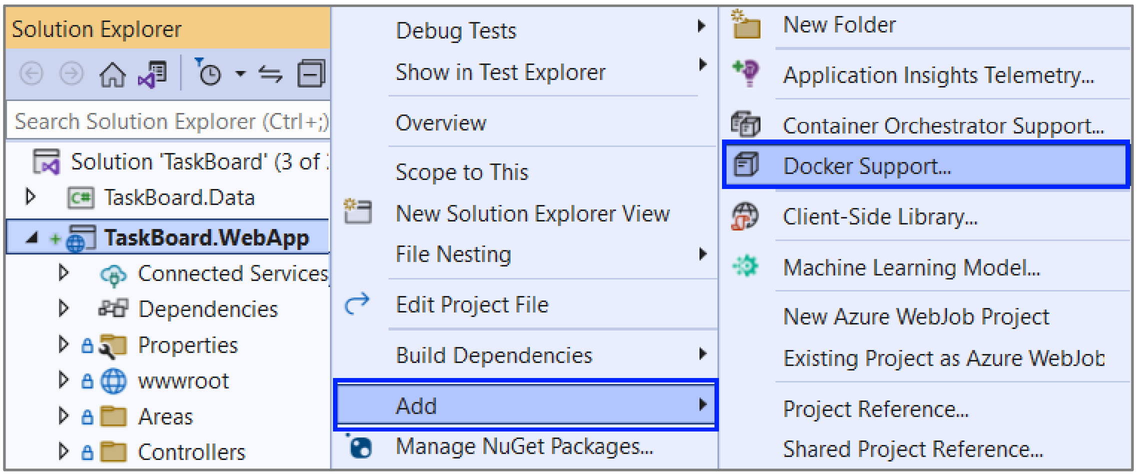
In this **task** and in the **other two tasks** connected to the **TaskBoard app**, we will work on the following **ASP.NET 6 MVC app** with a **SQL Server database**, provided in the **resources**:



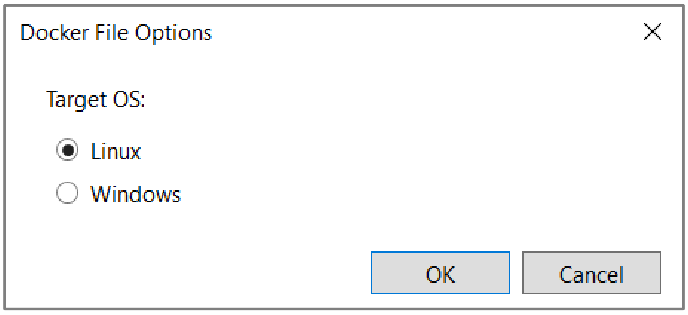
Our task is to create a custom image for this app. Later, we will also **publish this image** in **Docker Hub**.

Step 1: Create a Dockerfile in Visual Studio

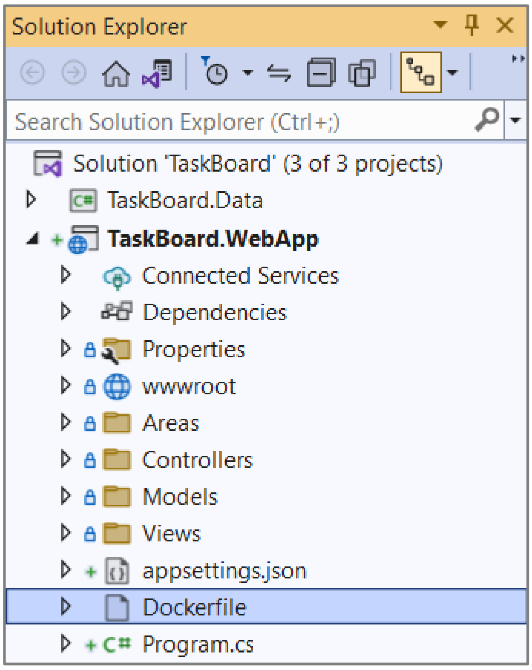
Our first job is to **create a Dockerfile for the app**, which will allow us to **run it in a Docker container** and later **connect it to a network**. Creating a **Dockerfile** is easy in **Visual Studio**, as it is **done for you** – you should only **right-click** on the "**`TaskBoard.WebApp`**" **project**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



→

```
#See https://aka.ms/containerfastmode to understand how Visual Studio uses this Dockerfile

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["TaskBoard.WebApp/TaskBoard.WebApp.csproj", "TaskBoard.WebApp/"]
COPY ["TaskBoard.Data/TaskBoard.Data.csproj", "TaskBoard.Data/"]
RUN dotnet restore "TaskBoard.WebApp/TaskBoard.WebApp.csproj"
COPY . .
WORKDIR "/src/TaskBoard.WebApp"
RUN dotnet build "TaskBoard.WebApp.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "TaskBoard.WebApp.csproj" -c Release -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
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**:

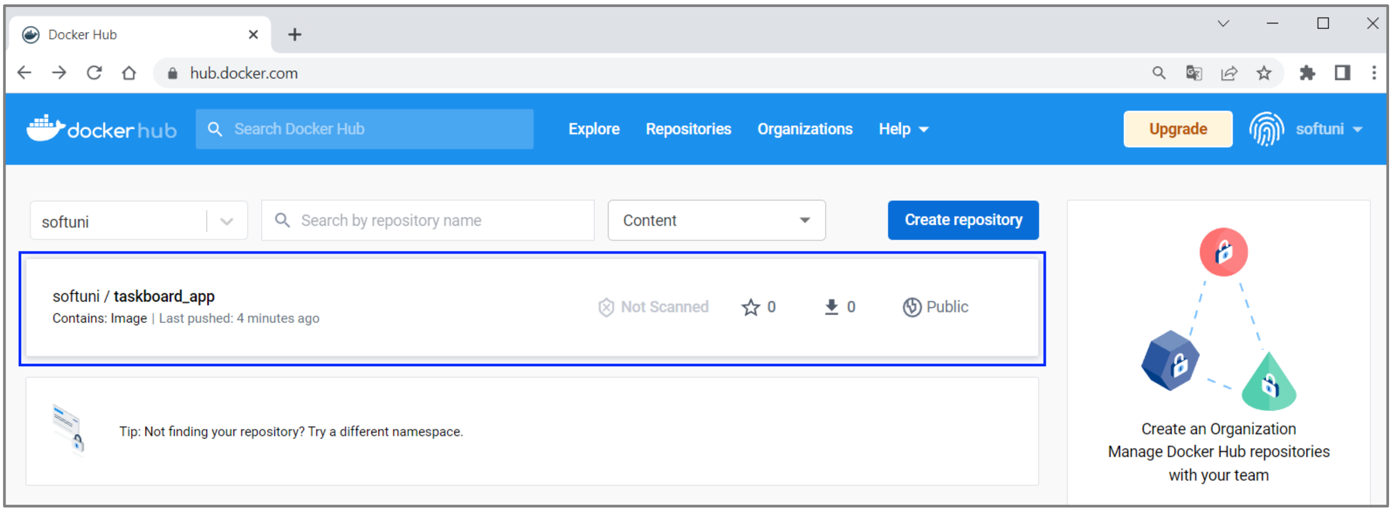
```
PS D:\Projects\TaskBoard> docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Doc
ker ID, head over to https://hub.docker.com to create one.
Username: softuni
Password:
Login Succeeded

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/
```

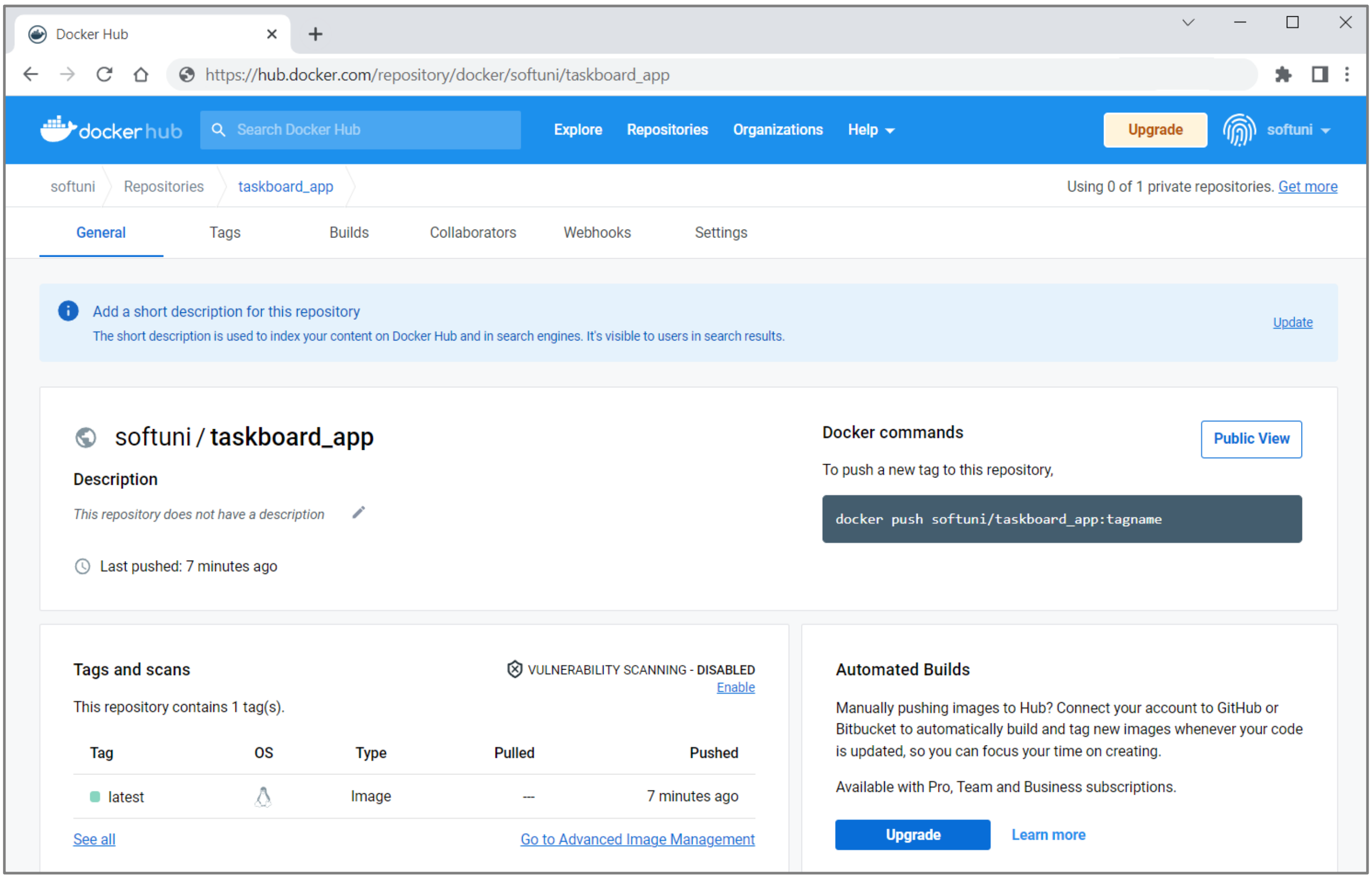
Now you should only **push the image**:

```
PS D:\Projects\TaskBoard> docker push softuni/taskboard_app
Using default tag: latest
The push refers to repository [docker.io/softuni/taskboard_app]
ebe6c6f54e6a: 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 si
ze: 1996
```

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



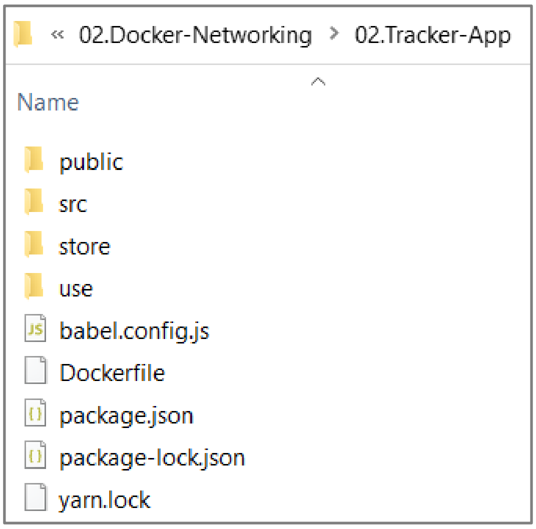


Now **keep the image** because we will use it in the next **TaskBoard app** task to **run a container**. Note that this **image is not enough** to run you whole app, as it **has a database** (and it needs a container too).

3. Tracker App

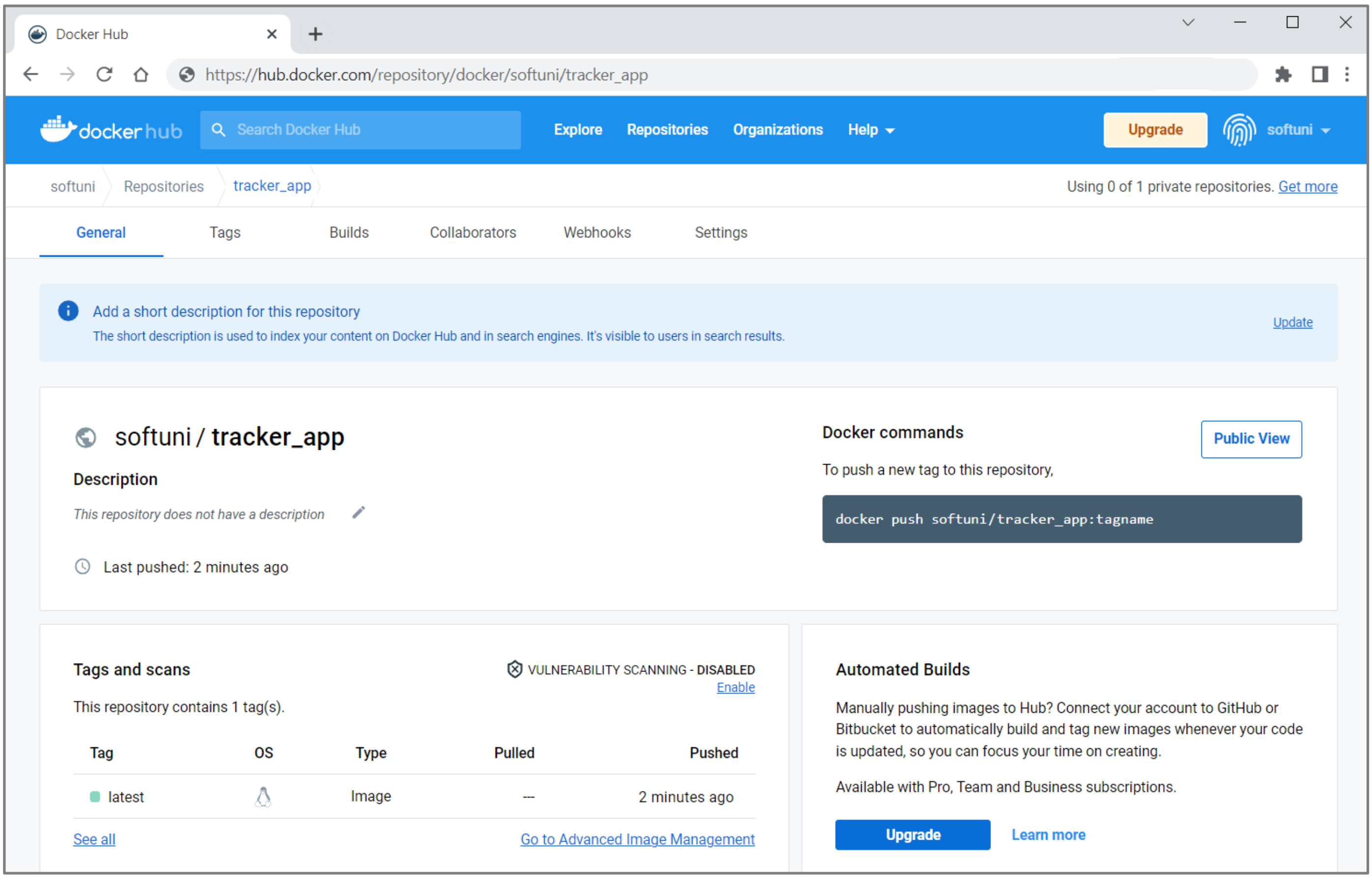
Your task now is to **run a simple JavaScript front-end app based on Vue.js** for keeping track of daily duties in a **Docker container**. It does not need **anything but an image** to run. It does not use a database or any other types of storage.

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

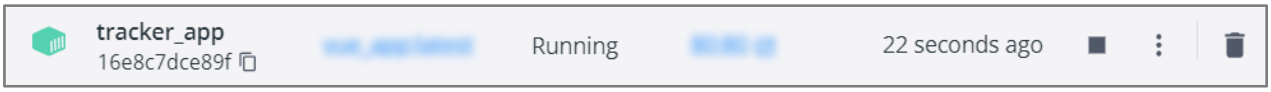


First, build a **custom image `{username}/tracker_app`** from the **given Dockerfile :**

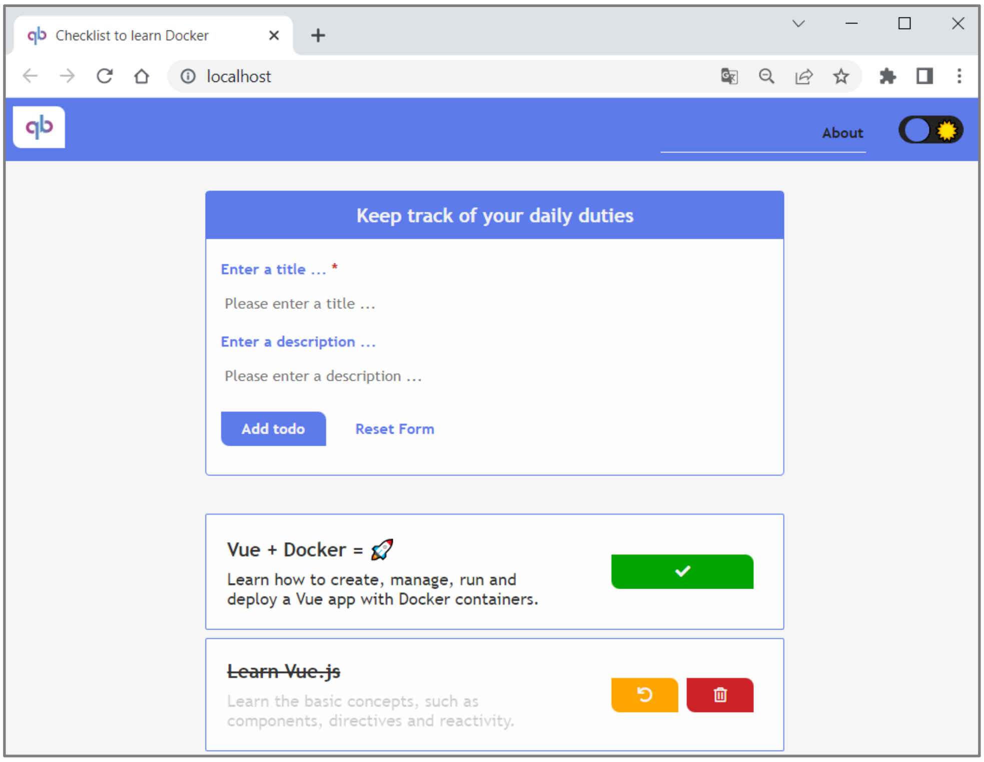
Push the image to **Docker Hub**:



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:



4. 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	mcr.microsoft.com/mssql/server:latest	Running	1433:1433	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

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=sqlserver;Database=MyDB;User Id=sa;
    Password=yourStrongPassword12#;MultipleActiveResultSets=true;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

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

```
softuni·ojects\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**:

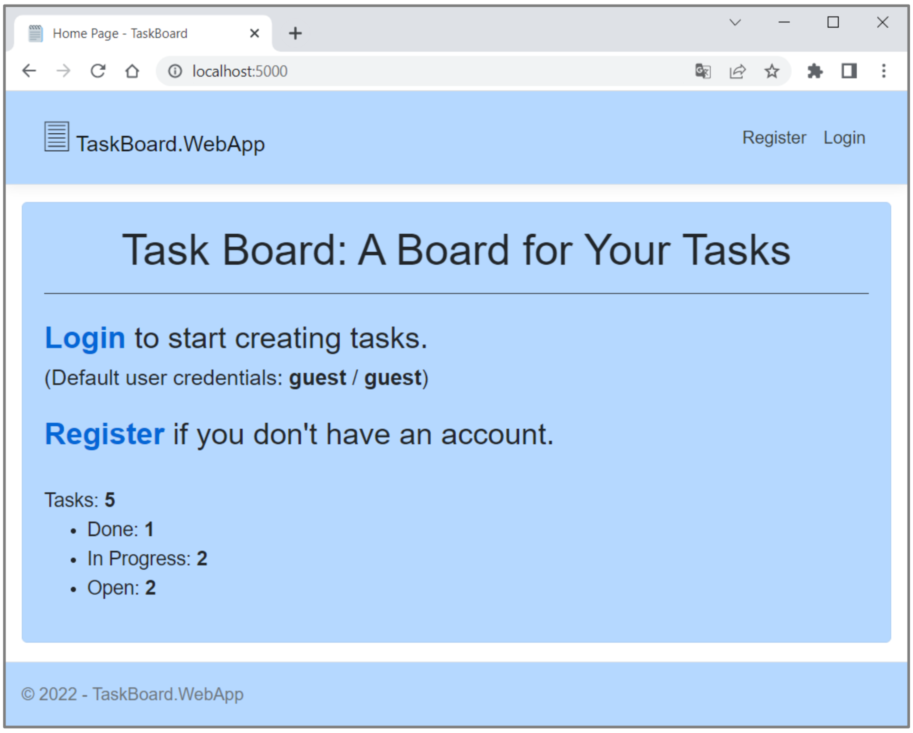
	NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
	web_app 3cbf315de9e4	softuni/taskboard_app:latest	Running	5000:80	40 seconds ago	
	sqlserver dd5d8e29589d	mcr.microsoft.com/mssql/server:latest	Running	1433:1433	4 minutes ago	

You can also see that they are **both connected** to out **`taskboard_network` network** when inspecting it:

```
PS C:\Users\PC> docker network inspect taskboard_network
[
    {
        "Name": "taskboard_network",
        "Id": "ea37c2c052b7f9553edab6d933ecf22fadcba3f378b5c7b63d10b2f0b2ef0ce1",
        "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**:

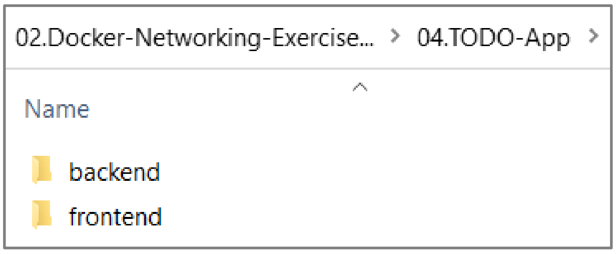


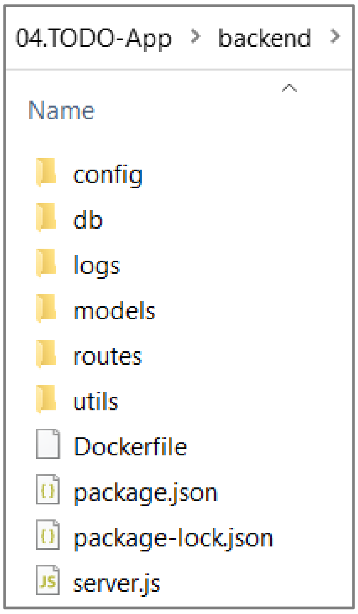
The app should be working – **test it by yourself**. In addition, you can try to **stop the app container** and **create a new one**, connected to the same **taskboard_network network** and you should see that the **database is preserved** because it is on a **SQL Server container**.

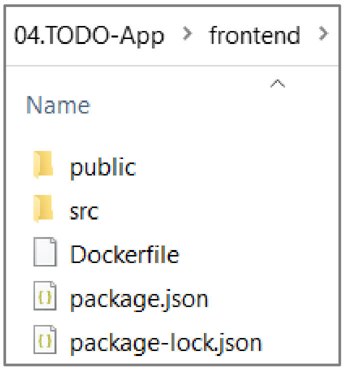
That is how you can **connect containers in a common network** and **use them together** to run **multi-container apps.**

5. TODO App

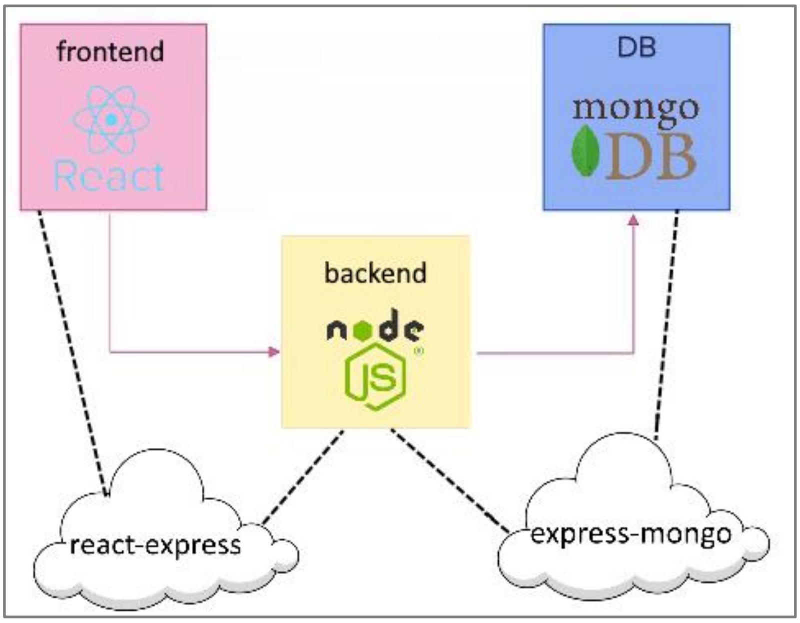
The **TODO app** (provided in the **resources**) is a simple app for **adding tasks**, which you should **Dockerize**:







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**:



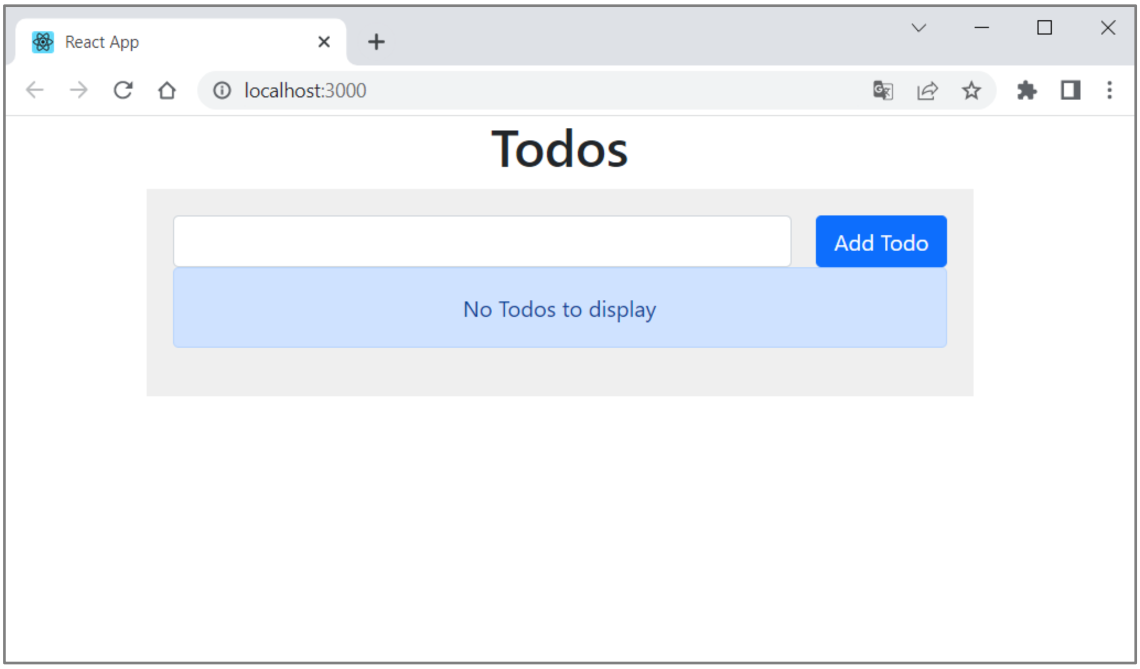
Requirements

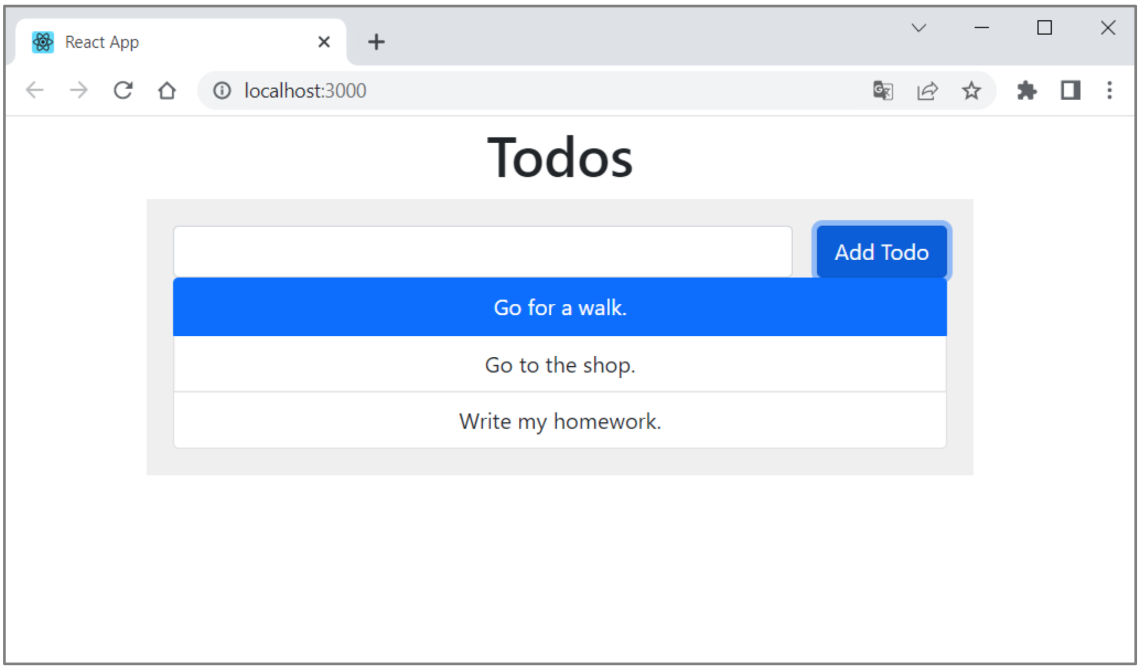
- **Name** the three containers "**frontend**", "**backend**" and "**mongo**"
- **Build images** from the **provided Dockerfiles** for the **frontend** and **backend services**
- Use the **latest image** for **MongoDB** from **Docker Hub**
- Expose the **frontend service** on port **3000** (see on which 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
	backend 18899a81f6bf	backend_image:latest	Running		10 minutes ago	
	frontend 56a93fa76104	frontend_image:latest	Running	3000:3000	10 minutes ago	
	mongo b56ab73b081f	mongo:latest	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.

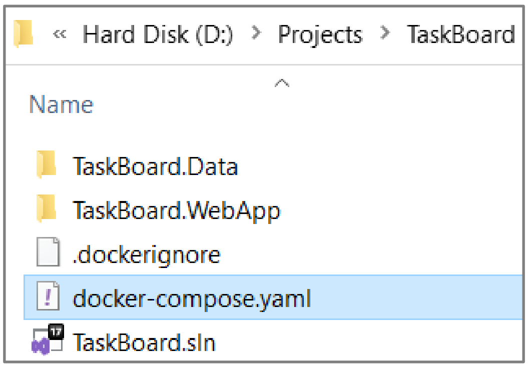
6. TaskBoard App: Orchestrating Containers with Docker Compose

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:

```yaml
version: "3.8"
```

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:

```yaml
services:
    sqlserver:
        container_name: sqlserver
        image: mcr.microsoft.com/mssql/server
        ports:
            - "1433:1433"
        environment:
            - ACCEPT_EULA=Y
            - MSSQL_SA_PASSWORD=yourStrongPassword12#
        volumes:
            - sqldata:/var/opt/mssql
        networks:
            - taskboard_network
```

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:

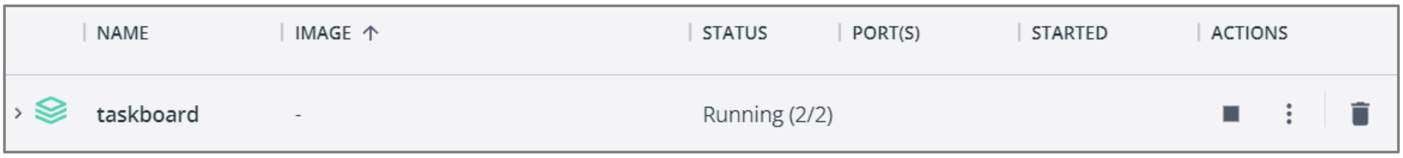
```
PS D:\Projects\TaskBoard> docker compose build
[+] Building 13.2s (19/19) FINISHED
 => [internal] load build definition from Dockerfile                                          0.1s
 => => transferring dockerfile: 875B                                                          0.1s
 => [internal] load .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
```

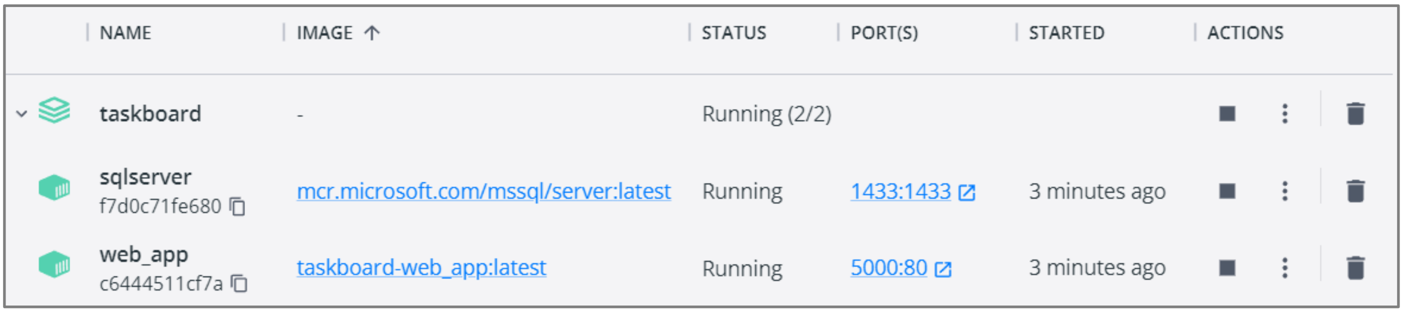
```
 => exporting to image                                                                        0.0s
 => => exporting layers                                                                       0.0s
 => => writing image sha256:2c94c77ed0b624990363db26840c7e9ee6d00b836694bdfc63b1285f          0.0s
 => => 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**:

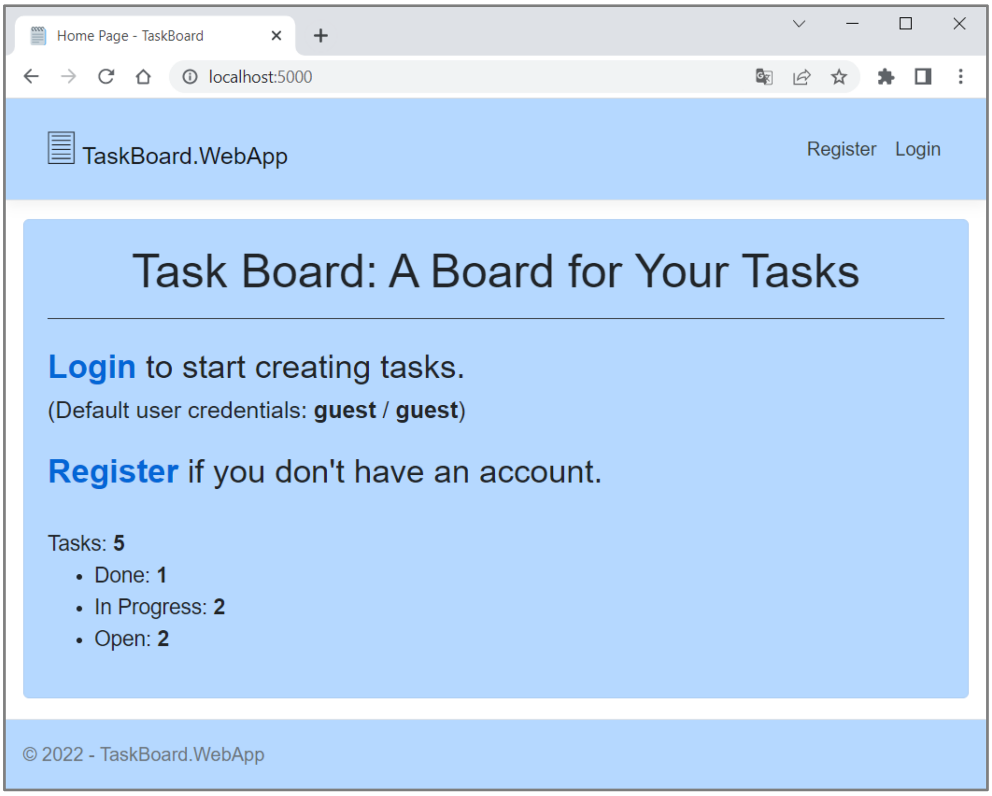
```
PS D:\Projects\TaskBoard> docker compose up
[+] Running 4/3
 - Network taskboard_network        Crea...                                                   0.8s
 - Volume "taskboard_sqldata"       Created                                                   0.0s
 - Container sqlserver              Created                                                   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**:





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**.

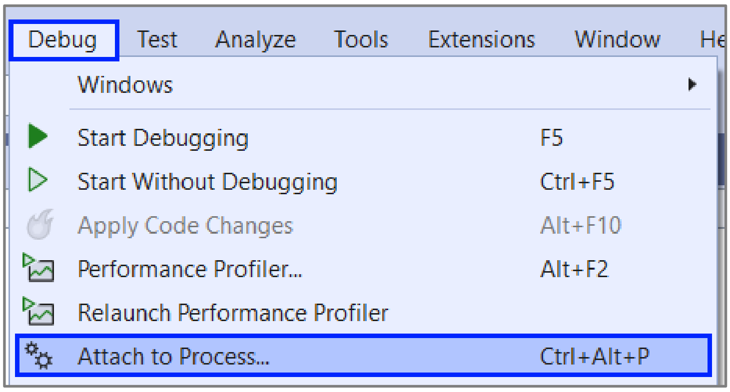
To do this, we should first **make changes to the Dockerfile** – it should have **Debug**, not **`Release` configurations**:

```
#See https://aka.ms/containerfastmode to understand how Visual Studio uses this Dockerfile

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["TaskBoard.WebApp/TaskBoard.WebApp.csproj", "TaskBoard.WebApp/"]
COPY ["TaskBoard.Data/TaskBoard.Data.csproj", "TaskBoard.Data/"]
RUN dotnet restore "TaskBoard.WebApp/TaskBoard.WebApp.csproj"
COPY . .
WORKDIR "/src/TaskBoard.WebApp"
RUN dotnet build "TaskBoard.WebApp.csproj" -c Debug -o /app/build

FROM build AS publish
RUN dotnet publish "TaskBoard.WebApp.csproj" -c Debug -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "TaskBoard.WebApp.dll"]
```

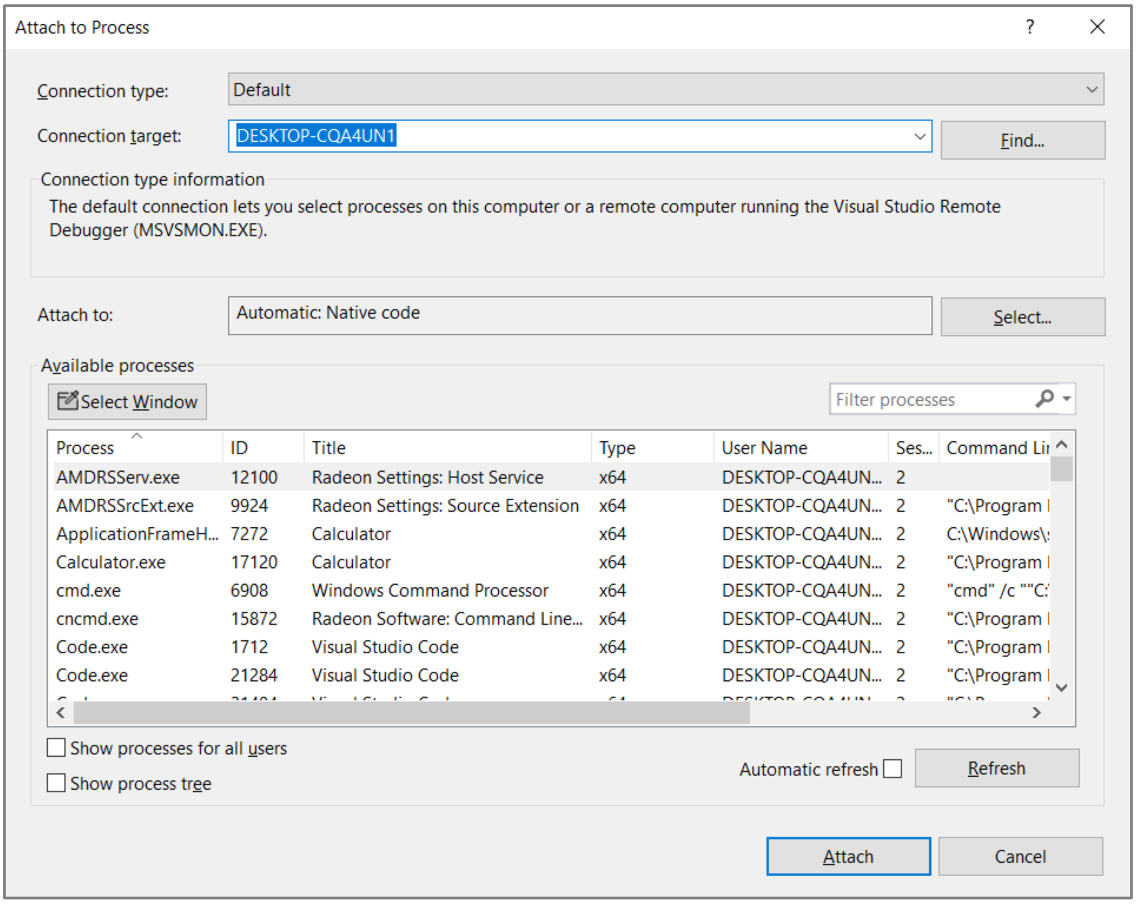
Save the file, **build images again** with **`docker compose build`** and **run new containers** with **`docker compose up`**:

NAME	IMAGE ↑	STATUS	PORT(S)	STARTED	ACTIONS
taskboard	-	Running (2/2)			
sqlserver a44647bb6f9b	mcr.microsoft.com/mssql/server:lates	Running	1433:1433	2 minutes ago	
web_app 74d2b9a815ea	taskboard-web_app:latest	Running	5000:80	2 minutes ago	

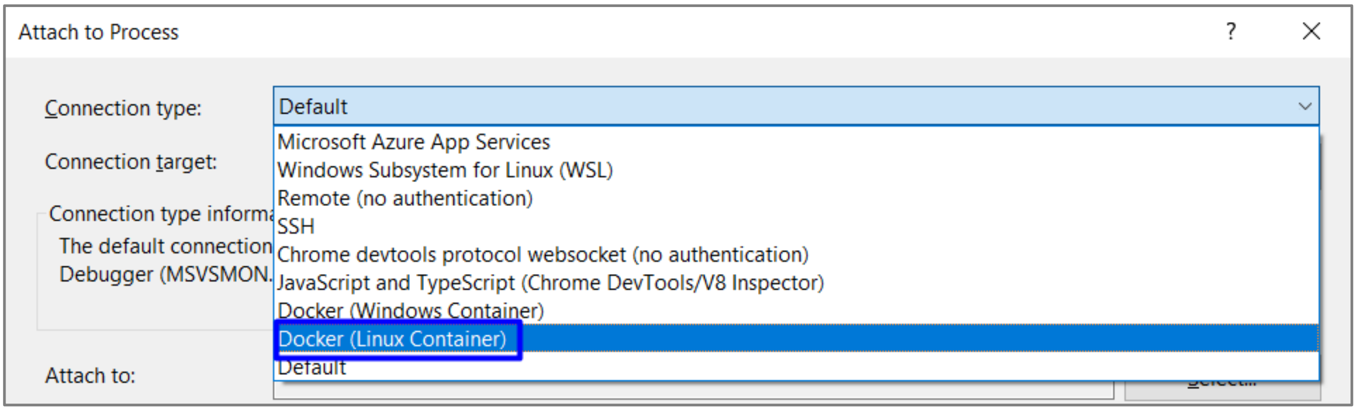
Now, in **Visual Studio**, go to **`[Debug]` → `[Attach to Process…]`** or use the **`[Ctrl]`**+**`[Alt]`**+**`[P]`** keys:



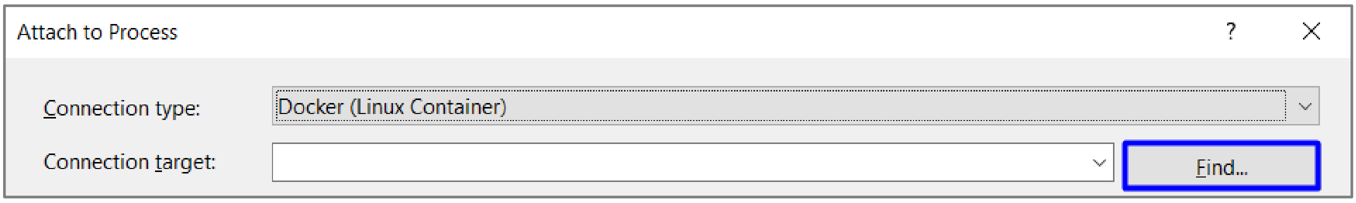
→



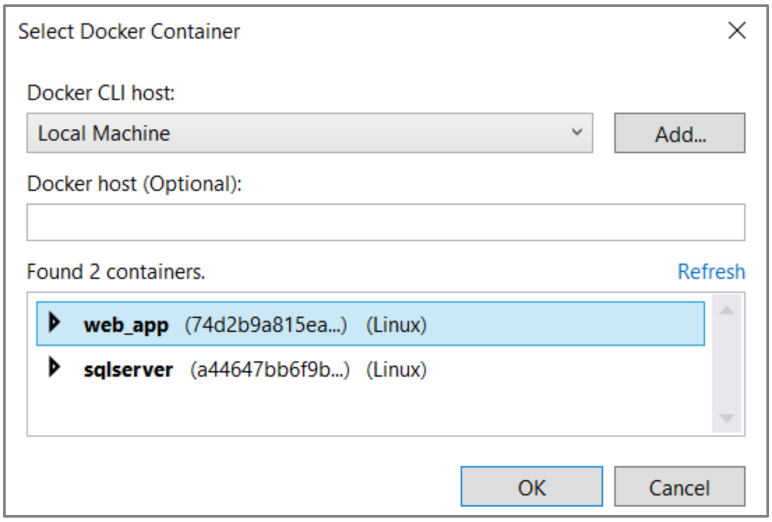
Change the connection type to **[Docker (Linux Container)]**:



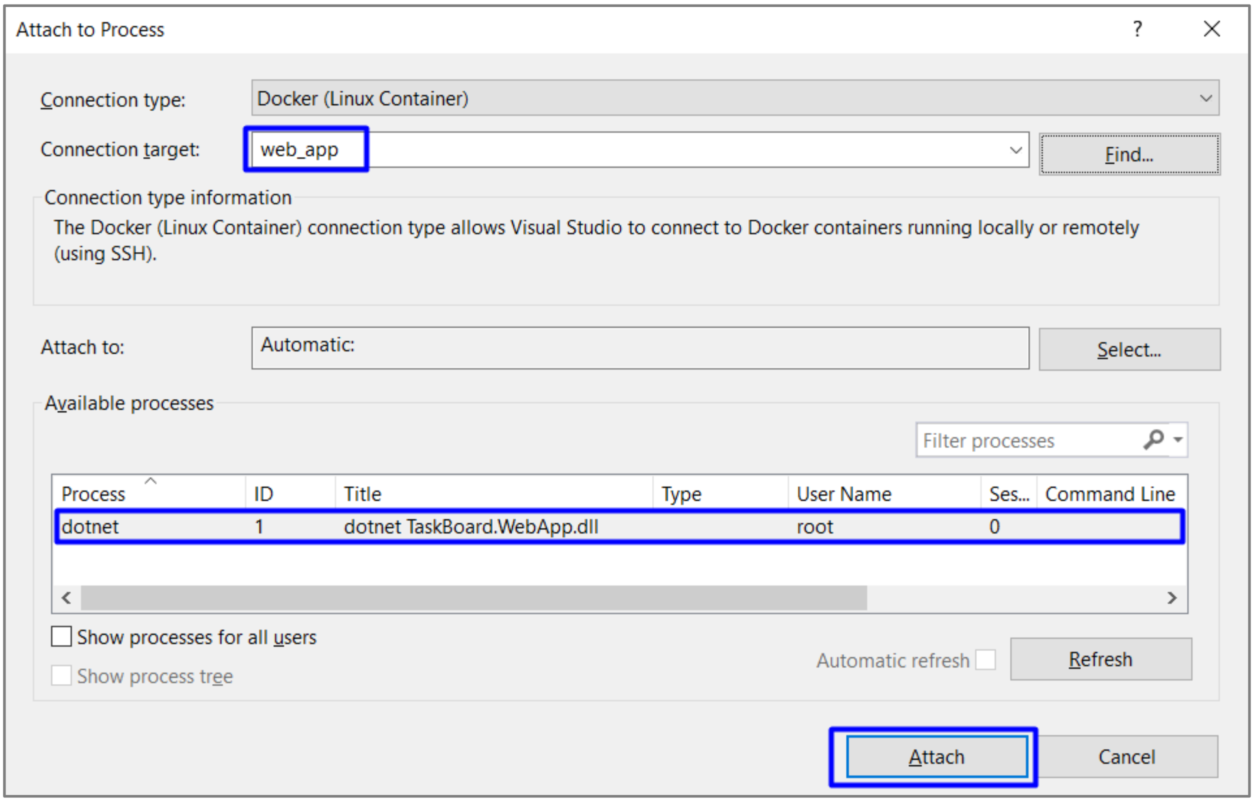
And click on the **[Find] button** to **choose a connection target**:



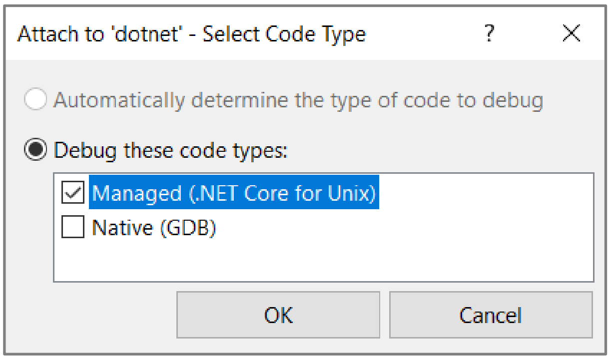
On the next window, select the **Web app container** and click on **[OK]**:



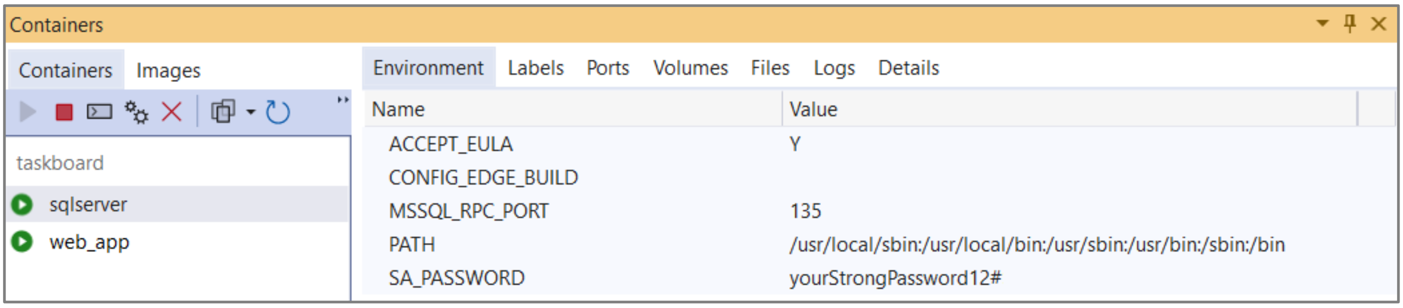
The **correct container** is chosen, so you should only click on the **`[Attach]` button**:



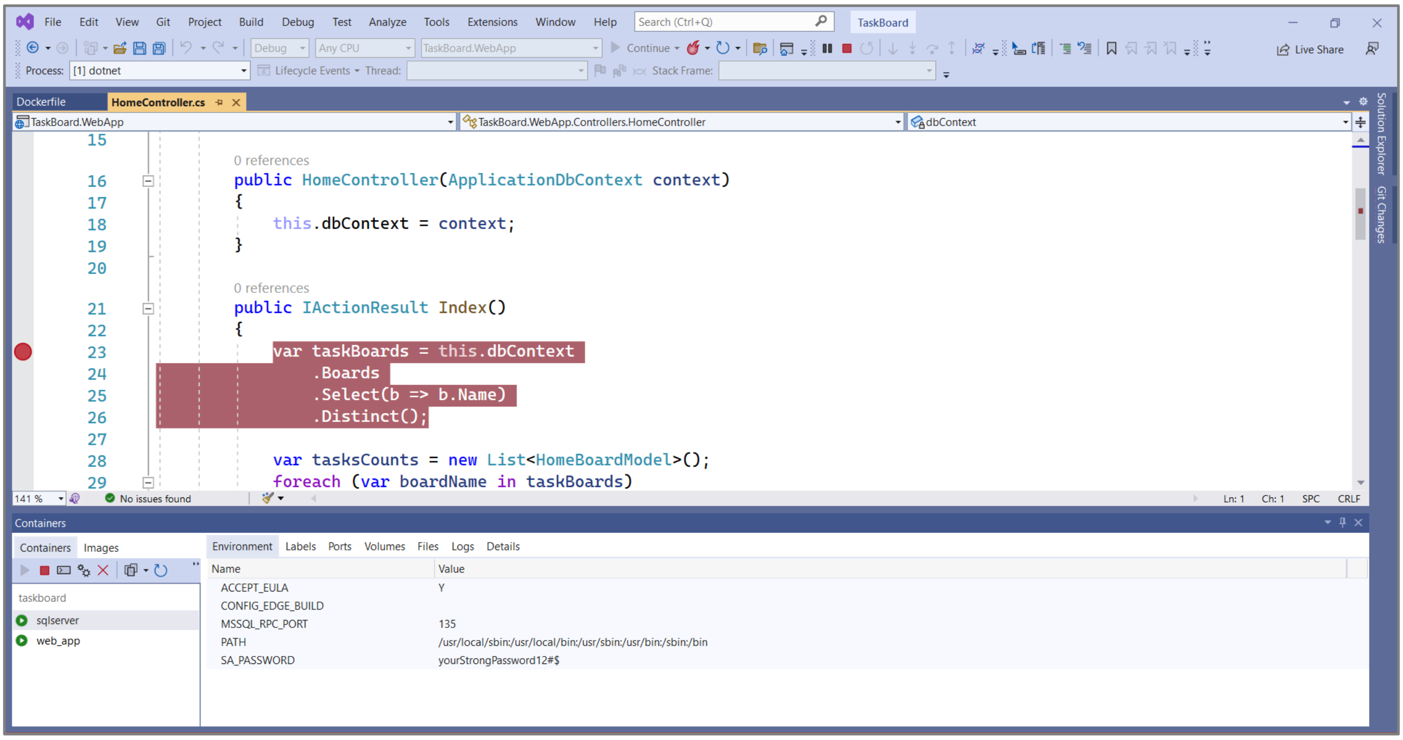
On the final step, choose the **`[Managed (.NET Core for UNIX)]` code type** and click **`[OK]`**:

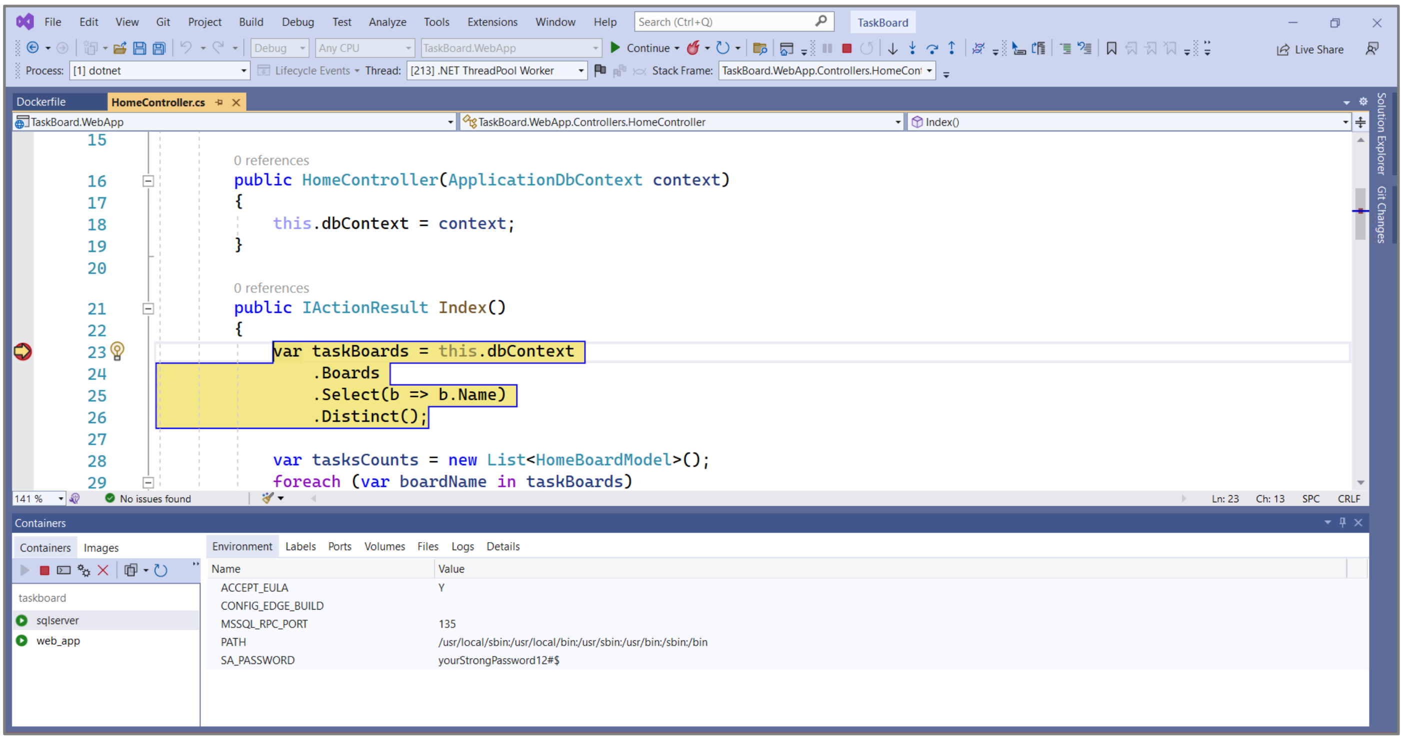


The **debug adapter is launched** and we are in **debug mode**:



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





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):

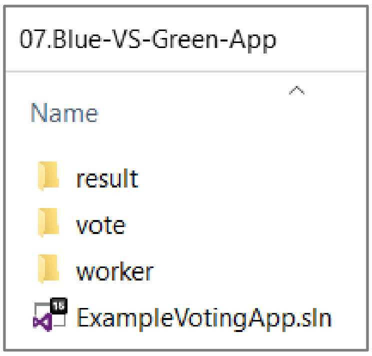
```
PS D:\Projects\TaskBoard> docker compose down --rmi all --volumes
[+] Running 6/6
 - Container web_app                          Removed                                        0.6s
 - Container sqlserver                        Remove...                                      1.0s
 - Volume taskboard_sqldata                   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
```

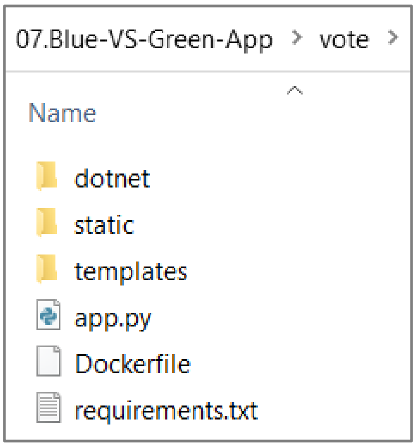
After this task, we now know how to **work with custom images**, **Dockerfiles**, **networks** and **Docker Compose**. We also know how to **run a multi-container ASP.NET Core + SQL Server app**.

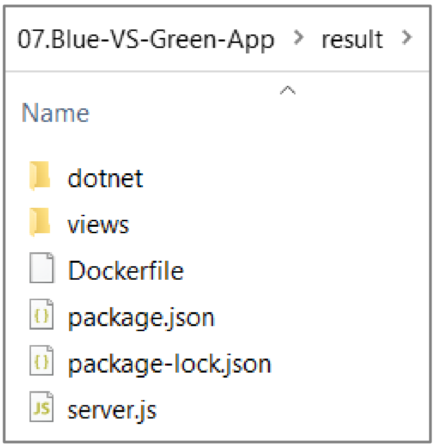
7. Blue VS Green App

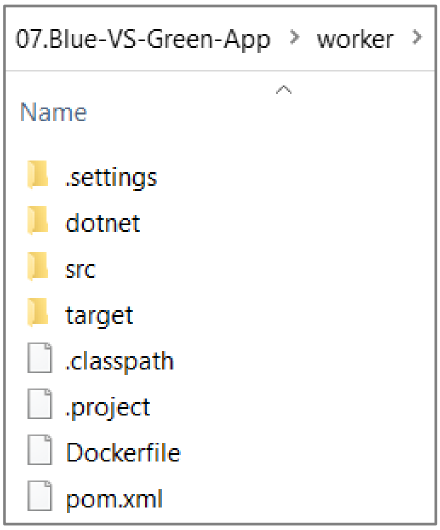
The "**Blue VS Green**" **app** (provided in the **resources**) is a

simple voting app, which you should **run with Docker Compose**:



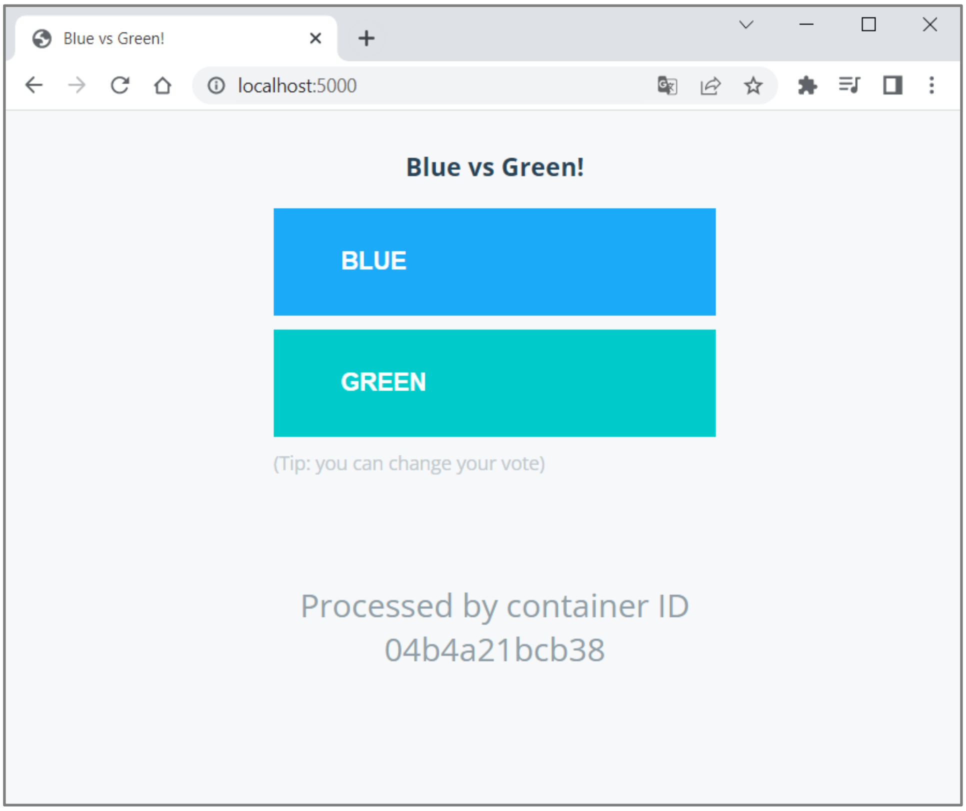


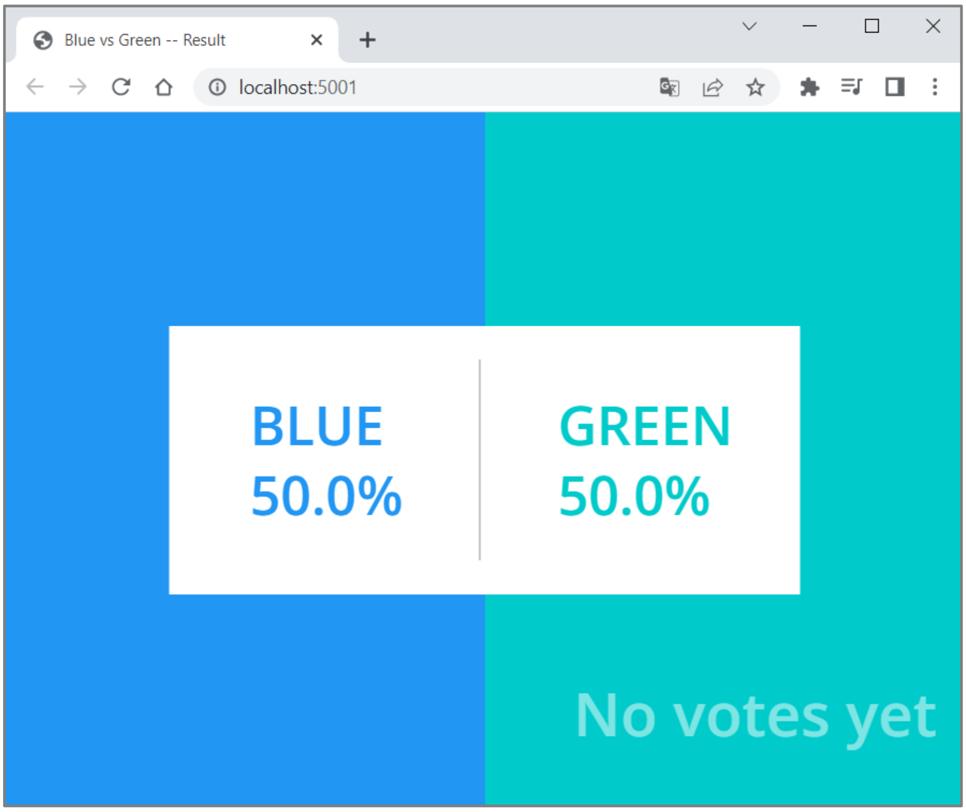




Note that the **Dockerfiles** for the **voting** and **worker apps** you see here are **empty**.

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**.

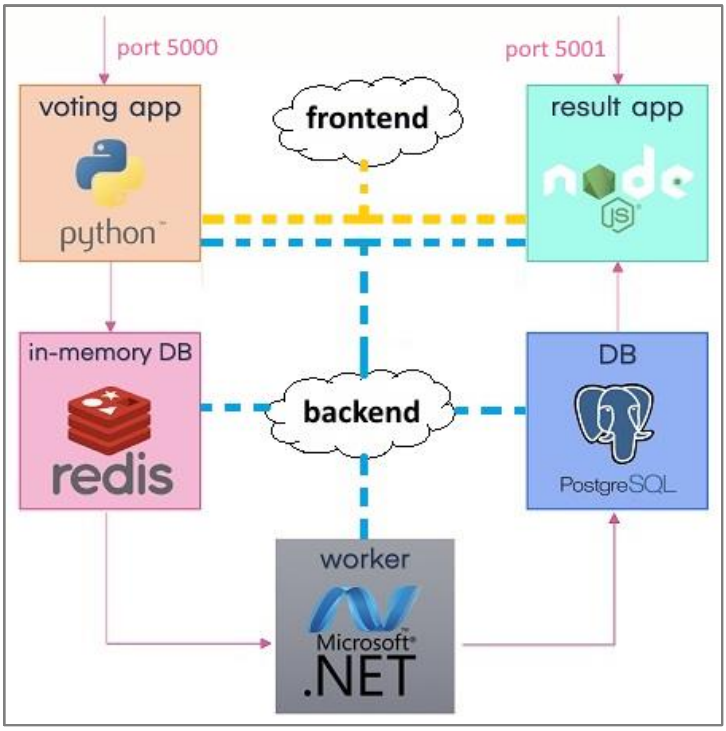
Your task is to **fill in the missing instructions** in the **`Dockerfiles`** and **run the app with Docker Compose**:

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

SoftUni

Follow us:

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**:

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

# Set the application directory


# Install our requirements.txt
COPY
RUN

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


# Make port 80 available for links and/or publish


# Define our command to be run when launching the container
CMD [ ]
```

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

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

# Create a working directory
WORKDIR /Worker

# Copy the .csproj file and restore



# Copy source files to the image


# Build the project


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

# Specify working directory for this stage


# Tell Docker what command to run when our image is executed inside a container


# Copy the /out directory from the build stage into the runtime image

```

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

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

8. Reseller App

In this **task**, we will work on a **Spring Boot app** with a **MySQL Server database**, provided in the **resources**. Our task is to create a custom image for this app. We will also **publish this image** in **Docker Hub**. After that, we will connect the **Reseller app** to its **MySQL 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**. Finally, we will make our **Reseller app and MySQL Server database containers work together** with **Docker Compose**.

Step 0: Create a JAR file

A **JAR** file (**Java ARchive**) is a package file format used to aggregate many **Java** class files and associated metadata and resources (such as text, images, etc.) into one file for distribution. It's used for storing compiled Java classes and associated resources, similar to how ZIP files work. **JAR** files are built on the **ZIP** file format and have the `.jar` file extension.

JAR files allow developers to package an application, library, or a set of related classes into a single file, making it easier to distribute and deploy. One of the features of **Spring Boot** is its ability to **produce standalone Spring applications** that can be run from the command line **without needing an external server**. These "fat" JARs include **embedded servers** and **all** the **necessary dependencies**. This makes deploying and running Spring Boot applications very convenient.

When you **containerize** a **Spring Boot** application using **Docker**, the **JAR** file is often the **artifact** you'll copy into the **Docker container** and **run**. This makes the **Dockerfile simpler**, as it can **focus** on setting up the **environment** and then **running** the application, rather than **compiling** code or **managing** the individual class files.

Now, let's create the **JAR** file for our **Reseller** app. If you already have a JAR file for the application, you can skip this step. However, if you don't have the JAR file, you should fulfill the following steps (the JAR file is **not** provided for our Reseller app, so you should execute the steps):

Install Maven

Download Maven from the official website:

Files

Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and follow the installation instructions. Use a source archive if you intend to build Maven yourself.

In order to guard against corrupted downloads/installations, it is highly recommended to verify the signature of the release bundles against the public KEYS used by the Apache Maven developers.

	Link	Checksums	Signature
Binary tar.gz archive	apache-maven-3.9.4-bin.tar.gz	apache-maven-3.9.4-bin.tar.gz.sha512	apache-maven-3.9.4-bin.tar.gz.asc
Binary zip archive	apache-maven-3.9.4-bin.zip	apache-maven-3.9.4-bin.zip.sha512	apache-maven-3.9.4-bin.zip.asc
Source tar.gz archive	apache-maven-3.9.4-src.tar.gz	apache-maven-3.9.4-src.tar.gz.sha512	apache-maven-3.9.4-src.tar.gz.asc
Source zip archive	apache-maven-3.9.4-src.zip	apache-maven-3.9.4-src.zip.sha512	apache-maven-3.9.4-src.zip.asc

Extract the archive to a directory of your choice and then add the bin directory of the created directory (e.g., apache-maven-3.x.x/bin) to the PATH environment variable.

Setting up the PATH Variable

To add Maven to the **PATH** variable:

- Right-click on "**This PC**" or "**Computer**" on the desktop and choose "**Properties**".
- Click on "**Advanced system settings**".
- Click on the **[Environment Variables]** button.
- Under "System Variables", find the **PATH** or **Path variable**, select it, and click on **[Edit]**.
- In the edit window, add the path to the Maven bin directory to the end of the value field. Make sure it's separated from previous paths with a semicolon (**;**).
- Click **[OK]**, and **[OK]** again to close the environment variable windows.

You can check if the installation was successful by executing the following command in a CMD:

```
PS C:\Users\      > mvn --version
Apache Maven 3.9.4 (dfbb324ad4a7c8fb0bf182e6d91b0ae20e3d2dd9)
Maven home: D:\Installs\apache-maven-3.9.4-bin\apache-maven-3.9.4
Java version: 11.0.12, vendor: Microsoft, runtime: C:\Program Files\Microsoft\jdk-11.0.12.7-hotspot
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Compile the Project into a JAR File

In order to compile the project into a JAR file, you should execute the following command:

```
C:\Users\      \Desktop\ResellerApp>mvn clean package
```

NOTE: If you receive an error that says that there are test failures, add **`-DskipTests`** at the end of the command.

Step 1: Create a Dockerfile

Now that we have the **JAR file** with the compiled project, it's time to create the **Dockerfile**.

First, we need to specify the **base image** that will be used to create the new **Docker** image. In our case, the official **OpenJDK** image with **Java 11** and the **JRE** (Java Runtime Environment) will be used. We will use the "**slim**" variant, which is a smaller version of the image that includes only the minimal packages needed to run Java, making the final **Docker** image size smaller:

```
# Use the official OpenJDK image as the base image
FROM openjdk:11-jre-slim
```

Then, we will set metadata for the **Docker** image. We will specify who the **maintainer** (or **creator**) of the **image** is. The **LABEL** instruction is used for adding **metadata** to an image, and it doesn't **affect** the image's structure or behavior

```
# Set metadata about the maintainer of the image
LABEL maintainer="[redacted]"
```

After that, we will define a variable that users can pass at build-time to the builder with the docker build command. In this case, the variable **`JAR_FILE`** will be set to a default value of **`target/*.jar`**, which would typically match the compiled JAR file of a Maven project.

Then, we will copy files or directories from the source (in this case, the location specified by **`${JAR_FILE}`**) to the destination in the Docker image's filesystem. In our case, we'll be copying the **JAR** file matched by the pattern **`target/*.jar`** from the host machine to the image and naming it **`app.jar`** inside the image:

```
# Copy the compiled jar file into the image
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
```

We will expose the part that the app runs on:

```
# Expose the port the app runs on
EXPOSE 8080
```

Finally, we will configure a container that will run as an executable – we will start the Java application when the container starts by running the JAR file that we previously copied into the image:

```
# Command to run the application
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Step 2: Build the Image

Now that we are ready with our Dockerfile we can build the image using this command:

```
C:\Users\      \Desktop\ResellerApp>docker build -t resellerapp .
```

Step 3: Create a Docker-Compose File

Now let's create a **docker-compose** file so that we can run our app. The **Reseller** app needs a **MySQL** server in order to run properly. We will create **services** for **both** our **app** and the **MySQL database** in the **same network**. The services will be in **separate containers** but **within** the **same network**. This way, the Spring app will **communicate** with the MySQL container.

Make sure that the **docker-compose** file is **placed** in the **root directory** of the **project**, alongside with the **pom.xml** file.

The configuration should be described following the notes below:

MySQL Service

- **`image`**: Use the MySQL image version **8.0**
- **`MYSQL_ROOT_PASSWORD`**: Create a password for the **MySQL root user**
- **`MYSQL_DATABASE`**: Create a database named **`resellerdb`**
- **`ports`**: Expose MySQL on port **`3306`**, so you can connect to it from outside if needed
- **`volumes`**: Ensure data persistence across container restarts

Reseller App Service

- **`depends_on`**: Ensure that the **MySQL** container starts **first**
- **`SPRING_DATASOURCE_URL`**: The hostname part of this URL (**`mysql`**) should match the service name of the MySQL service. The database name is **`resellerdb`** (the same as defined in the MySQL service).

Networks

- Both services should be defined to use the same network (**backend**), so they can communicate with each other.

Volumes

- **`Volumes`**: Define a volume for MySQL, ensuring that the database data will persist even if the MySQL container is removed.

The **docker-compose.yaml** file should look like this:

```yaml
version: '3'

services:
  resellerapp:
    image: resellerapp
    ports:
      - "8080:8080"
    depends_on:
      - mysql
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql:3306/resellerdb?useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=UTC
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=my-secret-pw
    networks:
      - backend

  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: my-secret-pw
      MYSQL_DATABASE: resellerdb
      MYSQL_USER: user
      MYSQL_PASSWORD: userpass
    ports:
      - "3306:3306"
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - backend

networks:
  backend:

volumes:
  mysql-data:
```

Step 4: Run the Docker-Compose File

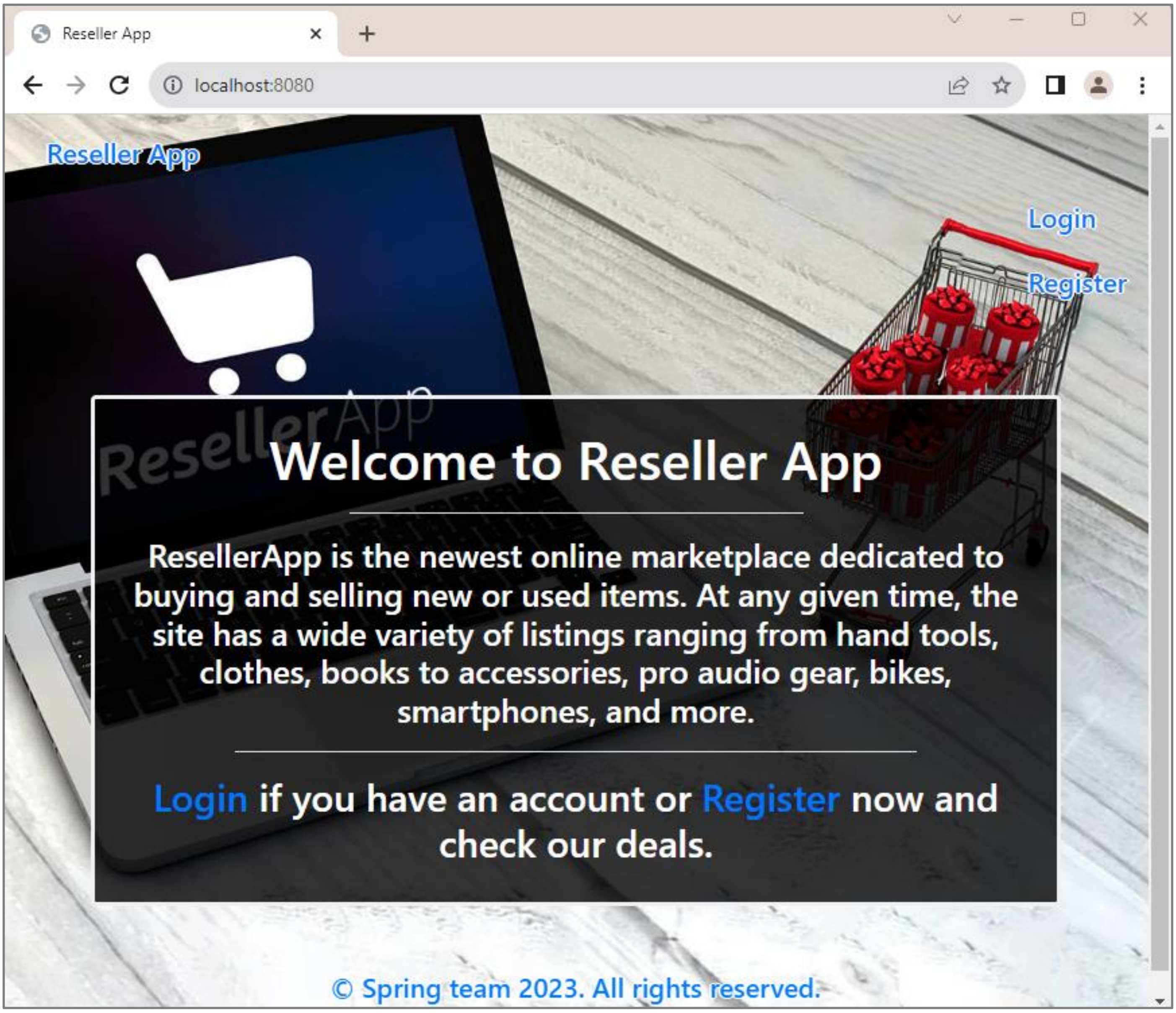
After we have our docker-compose file configured, we can run it with the following command:

```
C:\Users\      \Desktop\ResellerApp>docker-compose up -d
[+] Running 2/2
 ✔ Container resellerappauthorsolution-mysql-1         Started        0.4s
 ✔ Container resellerappauthorsolution-resellerapp-1   Started        1.0s
```

You can add **`-d`** at the end of the command in order to run it in detached mode.

Now, when you try to access the Reseller app on **`http://localhost:8000`**, you should see the Home page of the application.

If you want, you can register and try the app functionality, so that you can be sure that everything is working as expected.



9. Fruitipedia App

In this **task**, we will work on a simple **Django web app** without any databases, provided in the **resources**. Our task is to create a custom image for this app, create a **docker-compose** file and run it.

NOTE: Usually, a **Django** app comes with a **requirements.txt** file that lists all of the Python dependencies needed to run the app properly. Make sure that you are always provided with the **requirements.txt** file.

Step 1: Create a Dockerfile

First, let's create the Dockerfile in the root directory of our project, where **manage.py** is located.

The Dockerfile should follow the **requirements** below:

Specify the base image to use for the container. Use the official Python image based on Debian Buster with Python 3.8 installed. The **`slim-buster`** variant is a minimal version, which makes the image smaller in size.

Set the working directory in the container to **/app**.

Set environment variables:

- **PYTHONDONTWRITEBYTECODE**: Prevent Python from writing .pyc files

- PYTHONUNBUFFERED: Ensure that Python output is sent straight to terminal without being buffered, which is useful for logging

Copies the **requirements.txt** file from your host machine to the **`/app/`** directory in the container.

Install the Python packages specified in **`requirements.txt`** and disable the cache to make the image smaller.

Copy all files and folders from your current directory on your host machine into the **`/app/`** directory in the container.

Creates a new directory named **`staticfiles`** in the **`/app/`** directory of the container.

Copies the **`static`** folder from your host machine into **`/app/staticfiles`** in the container.

Expose port **`8000`**.

Finally, specify the command to run when the container starts. In our case, it should start the Gunicorn HTTP server and runs our application on port **`8000`**.

Here's how it should look:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim-buster

# Set the working directory in the container to /app
WORKDIR /app

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Install any needed packages specified in requirements.txt
COPY requirements.txt /app/
RUN pip install --no-cache-dir -r requirements.txt

# Copy the current directory contents into the container at /app
COPY . /app/

# Set the static files directory
RUN mkdir /app/staticfiles

# Copy the static files
COPY static /app/staticfiles

# Expose the port the app runs on
EXPOSE 8000

# Run command to start the application
CMD ["gunicorn", "SimplePageApp.wsgi:application", "--bind", "0.0.0.0:8000"]
```

Step 2: Create a Docker-Compose File

Now let's create a **docker-compose** file so that we can run our app.

Make sure that the **docker-compose** file is **placed** in the **root directory** of the **project**.

The configuration should be described following the notes below:

Compose Version

Specify the Docker Compose file format version to use version **`3.8`**.

Services

Web Service:

- **`web:`** Define a new service named web

- **build: .**: Build the Docker image for this service using the Dockerfile in the current directory
- **volumes**: Define volume mounts for this service
- **./static:/app/staticfiles**: Map the current directory on the host to **/app** inside the container
- **ports**: Publish port **8000** of the service to port **8000** on the host machine
- **DEBUG=False**

Finally, the file should look like this:

```yaml
version: '3.8'

services:
  web:
    build: .
    volumes:
      - ./static:/app/staticfiles
    ports:
      - "8000:8000"
    environment:
      - DEBUG=False
```

Step 3: Build and Run the Docker Container

After we have the **docker-compose.yaml** file, we can start the service by running the following command from the root directory, where the **docker-compose.yml** file is located:

```
PS C:\Users\      \Desktop\SimplePageApp> docker-compose up --build
```

That's it! Your Django application should now be running, and you should be able to access it at http://localhost:8000