# Exercises: Docker Compose

Exercises for the "Containers and Clouds" course @ SoftUni

## \*MariaDB Client and Server in a Network

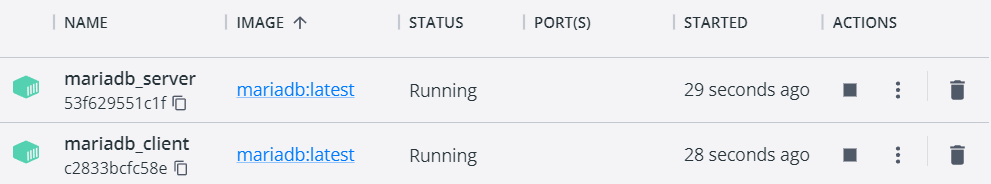
MariaDB Server (a variant of MySQL) is one of the most popular **open-source relational databases**. You should use it **documentation** on Docker Hub to create **two containers**, which will **work** **together**:

* **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.

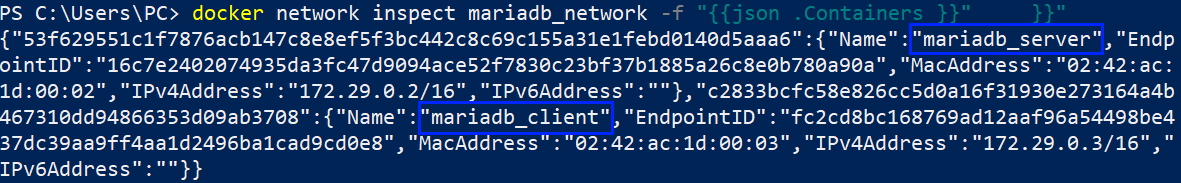
Both containers will use **the same Docker image**. The **image** is available here: <https://hub.docker.com/_/mariadb>.

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.

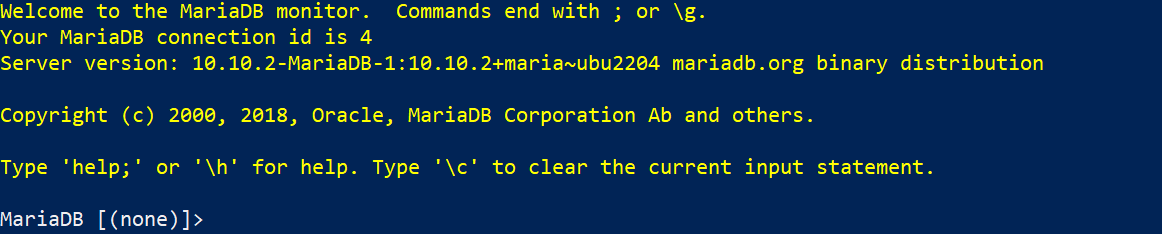
At the end, you should have **two containers** like this:



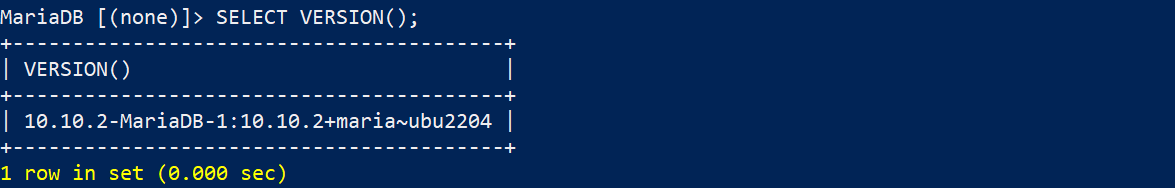
They should be **connected in the same network**. With the command below you can see all **containers** **in a specified network**:



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



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



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.

## 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**:

Graphical user interface, text, application

Description automatically generated

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…]:

Graphical user interface, application

Description automatically generated

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

Graphical user interface

Description automatically generated with medium confidence

The Dockerfile should be **created successfully**:

Graphical user interface, text, application

Description automatically generated

🡪Text

Description automatically generated

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:

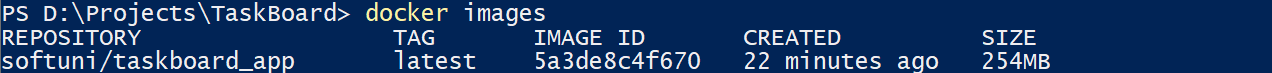
* Navigate to the TaskBoard **solution directory**
* Use the docker build command to **build the image**
* Set the **local directory** as the **working directory**
* With the -f option, set the **path to the** Dockerfile
* With the -t option, set the **name of the image** in format {your Docker Hub username}/{app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

Text

Description automatically generated

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



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**:

Text, timeline

Description automatically generated

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

Graphical user interface, text, email

Description automatically generated

Graphical user interface, text, email

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

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).

## Tracker App

Your task now is to **run** **a** **simple JavaScript front-end app based on** Vue.jsfor 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**:

Graphical user interface, application

Description automatically generated

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

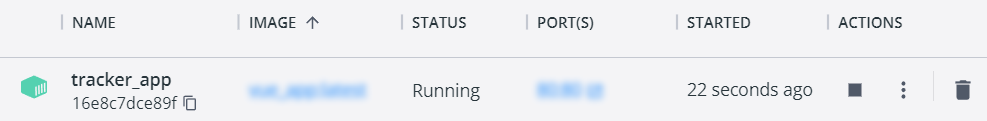


**Push the image** to Docker Hub:

Graphical user interface, text, application, email

Description automatically generated

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:

Graphical user interface, application

Description automatically generated

## 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**.

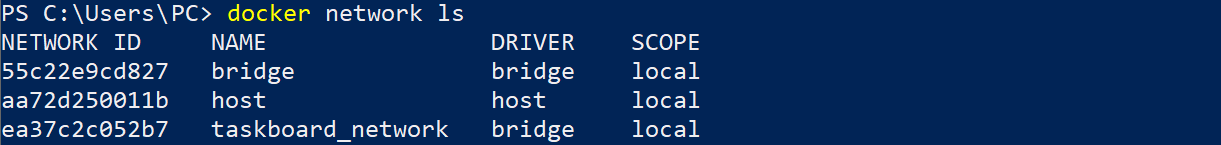
After we have an **image** **for the** TaskBoard **app** and know how to **run a** SQLServer **container**, let's learn how to **create and connect them to a network**.

### Step 1: Create a Network

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



You can see **all networks** with:



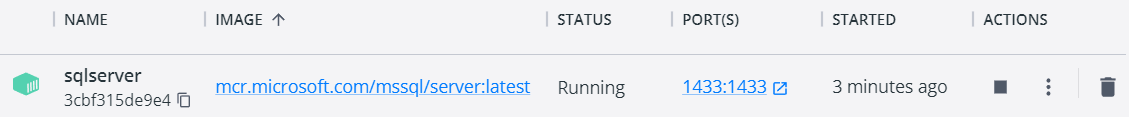
### Step 2: Create and Connect a SQL Server Container

Now we want to **run a** SQLServer **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**:

Text

Description automatically generated

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** databaseconnectionstring of the app, so that it can **connect to the** SQLServer **database** we created. Open the appsettings.json **file** of the "TaskBoard.WebApp" **project** and **modify it** according to the following requirements:

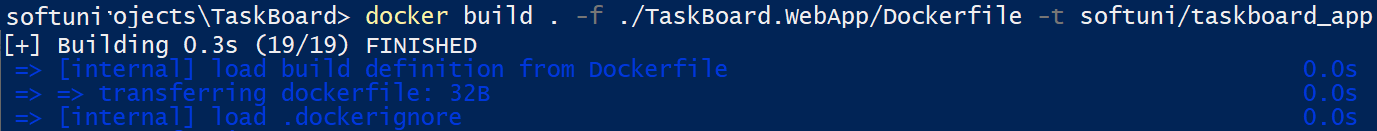
* **Server** should be sqlserver
* **Database name** is of your choice
* User Id should be sa (the default database system admin user)
* **Password** should be the **admin password** we set in the previous command – yourStrongPassword12#
* Allow **multiple connections**

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

Graphical user interface, text, application, Word

Description automatically generated

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



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

Graphical user interface, text

Description automatically generated

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

### Step 4: Containers Together

These are **our containers**:

Graphical user interface, application

Description automatically generated

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

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

And when you go to http://localhost:5000 you have the **fully working** TaskBoardWeb **app** with a **database**:

Graphical user interface, text, application, email

Description automatically generated

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.**

## TODO App

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

Graphical user interface, application

Description automatically generated A picture containing table

Description automatically generated Graphical user interface, text, application

Description automatically generated

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**:

Diagram

Description automatically generated with medium confidence

### 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:

Graphical user interface, application

Description automatically generated

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

Graphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generated

### 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.

## TaskBoard App: Orchestrating Containers with Docker Compose

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

### Step 1: Build a YAML File

Our first job is to **build a** DockerComposeYAML **file**. It will **replace the separate** dockerrun **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**:

Graphical user interface, text, application

Description automatically generated

**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:

Graphical user interface, text, application

Description automatically generated

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**:

Graphical user interface

Description automatically generated

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:



**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:

Text

Description automatically generated

Text

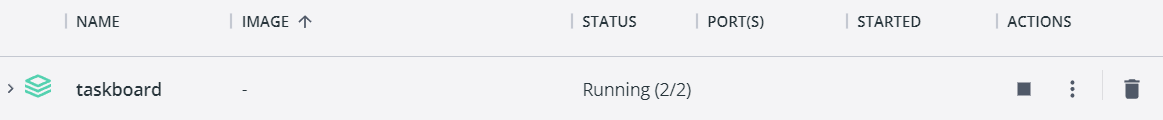
Description automatically generated

Then, **run the containers together** with Docker Compose:

Text

Description automatically generated

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

Graphical user interface, application

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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** TaskBoardWeb **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**:

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface, text, application

Description automatically generated

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

Graphical user interface, text, application

Description automatically generated 🡪

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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

A screenshot of a computer

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface, text, application, chat or text message

Description automatically generated

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

Graphical user interface, text, application, chat or text message, email

Description automatically generated

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

Graphical user interface, application, Word

Description automatically generatedGraphical user interface, text, application

Description automatically generated

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):

Graphical user interface, text

Description automatically generated

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**.

## Blue VS Green App

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

Table

Description automatically generated with medium confidence**simple voting app**, which you should **run with** Docker Compose:

Graphical user interface, application

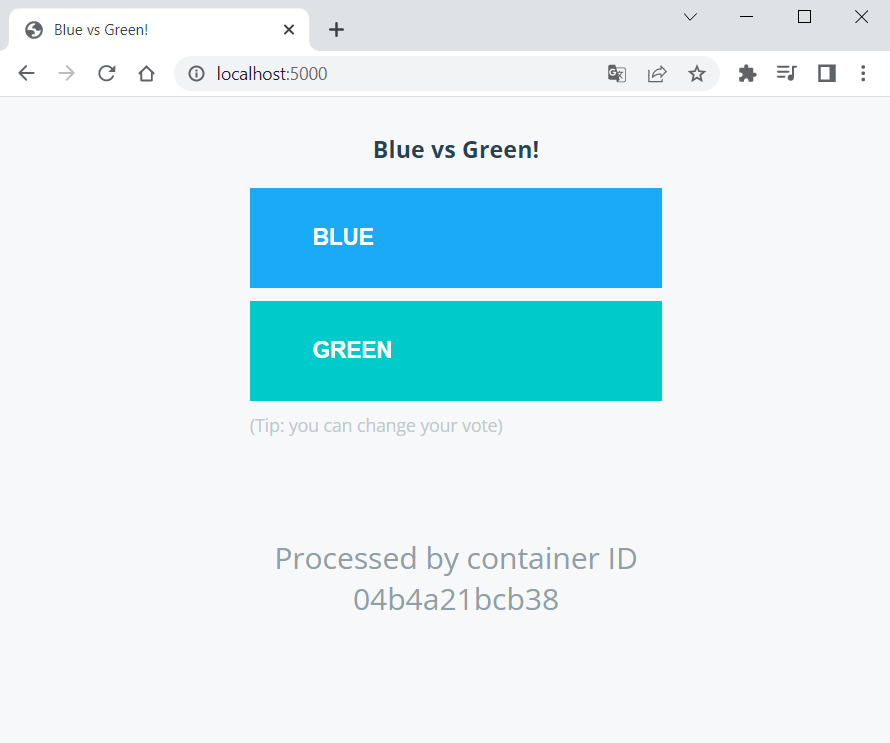
Description automatically generated Graphical user interface, application

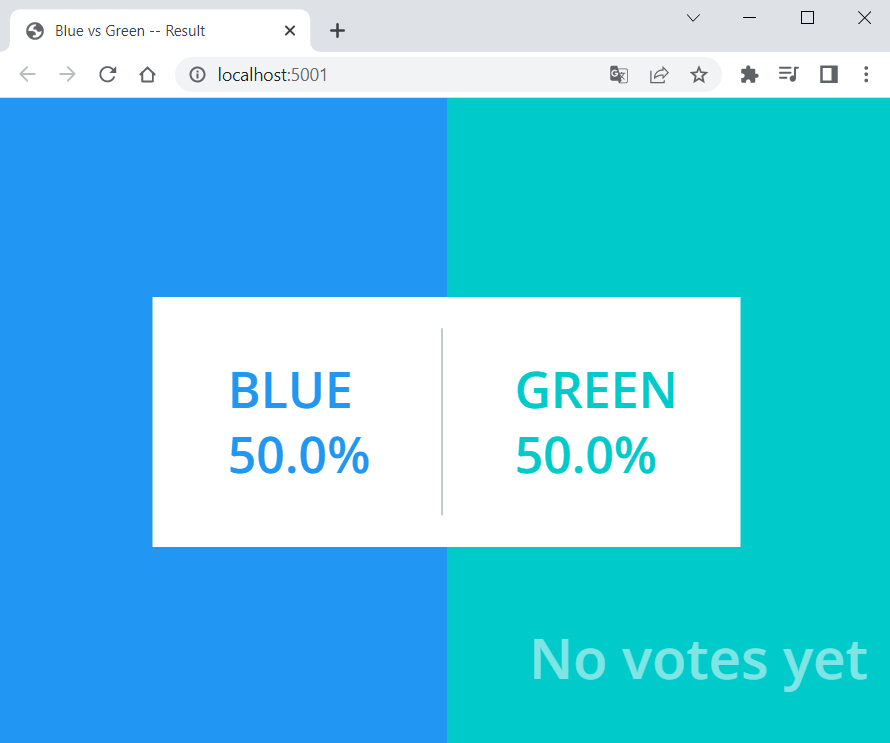
Description automatically generated Graphical user interface, application

Description automatically generated

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

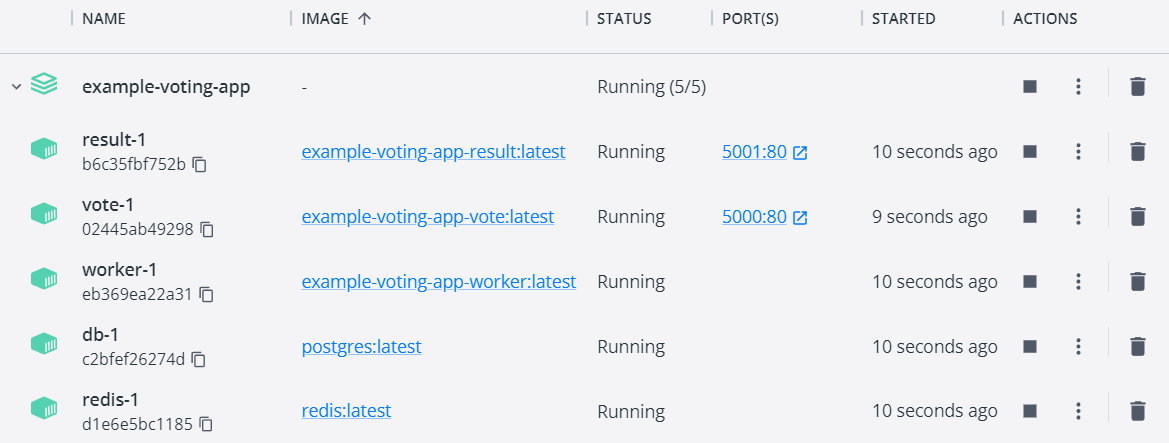
It provides an **interface for a user to vote** and another **interface to show the results**:





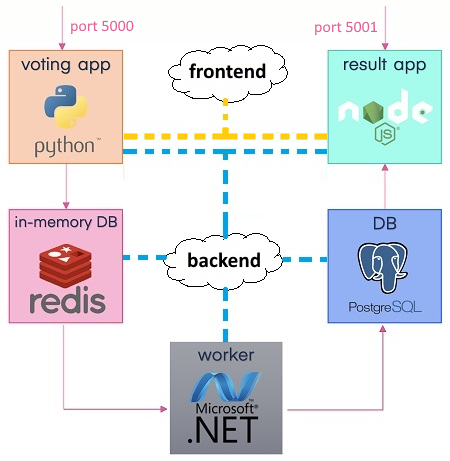
You can **vote** and then **change your vote** and this will **make changes in the results**.

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



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**

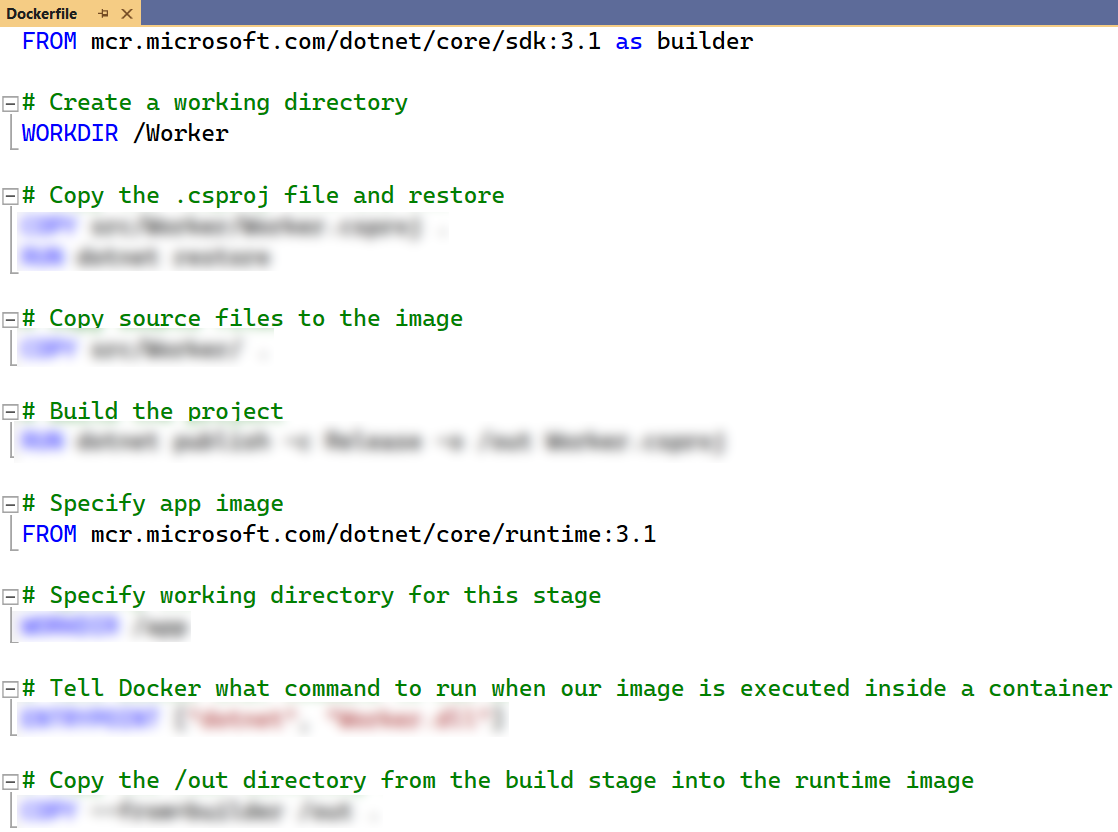
### Hints

**Find out** how to write the Dockerfiles **you need** from the Docker Documentation: [https://docs.docker.com](https://docs.docker.com/).

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



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

## 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 **My**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**. Finally, we will make our Reseller **app and My**SQLServer **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.

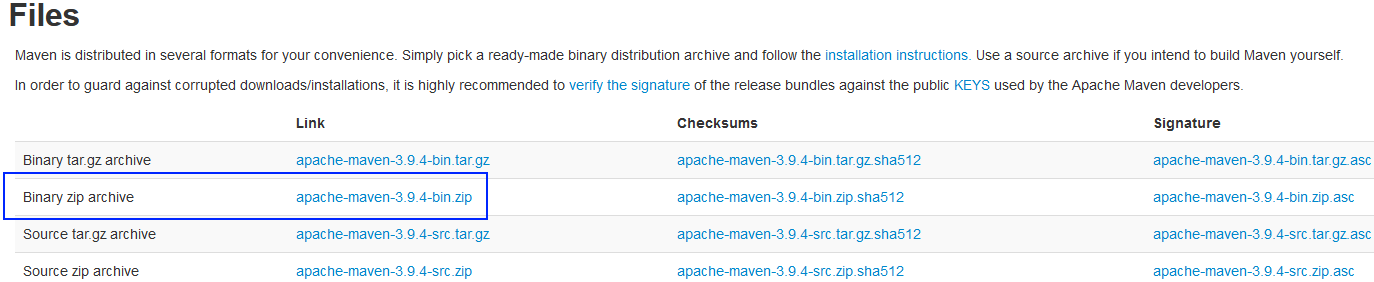
JAR files allow developers to package an application, library, or a set of related classes into a single file, making it easier to distribute and deploy. One of the features of **Spring Boot** is its ability to **produce** **standalone** **Spring** **applications** that can be run from the command line **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](https://maven.apache.org/download.cgi):



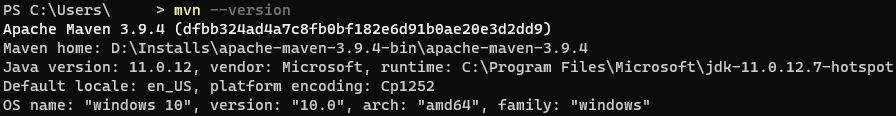
Extract the archive to a directory of your choice and then add the bin directory of the created directory (e.g., apache-maven-3.x.x/bin) to the PATH environment variable.

#### 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:



#### Compile the Project into a JAR File

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



**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:

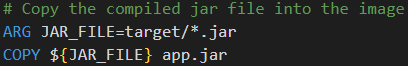


Then, we will set metadata for the **Docker** image. We will specify who the **maintainer** (or **creator**) of the **image** is. The **LABEL** instruction is used for adding **metadata** to an image, and it doesn't **affect** the image's structure or behavior



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:



We will expose the part that the app runs on:



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:



### Step 2: Build the Image

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



### 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).

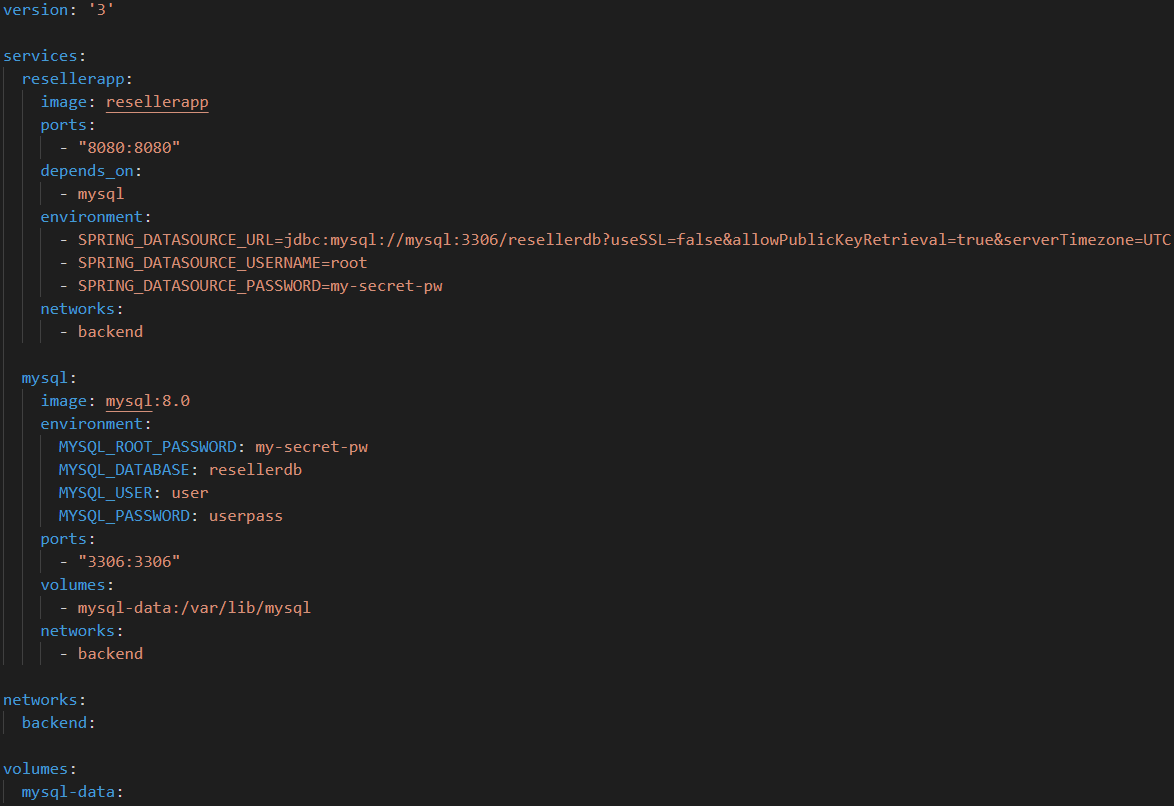
#### Networks

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

#### Volumes

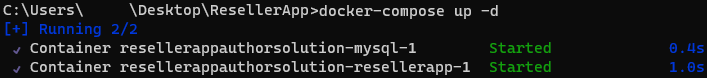
* **Volumes**: Define a volume for MySQL, ensuring that the database data will persist even if the MySQL container is removed.

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



### Step 4: Run the Docker-Compose File

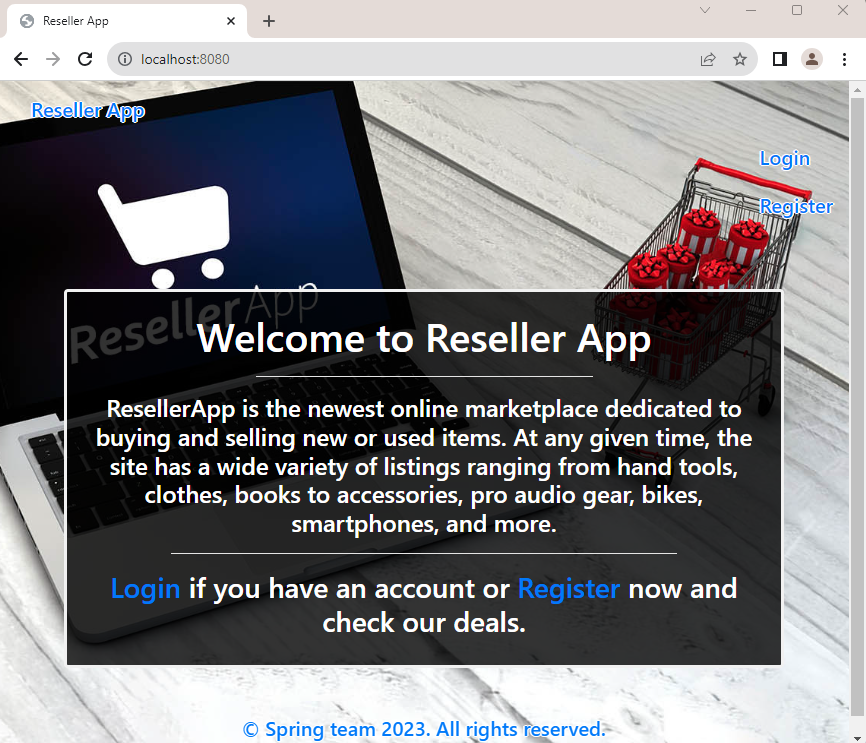
After we have our docker-compose file configured, we can run it with the following command:



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.



## 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.

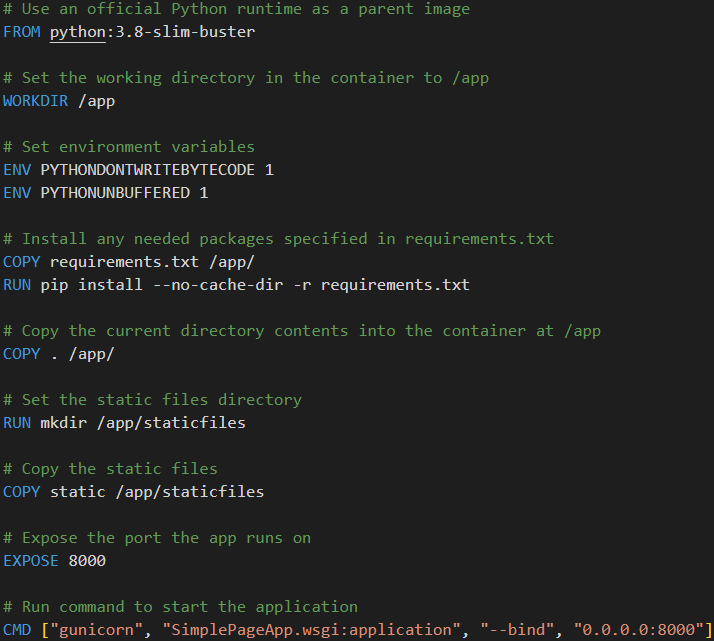
Creates a new directory named **staticfiles** in the **/app/** directory of the container.

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:



### 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

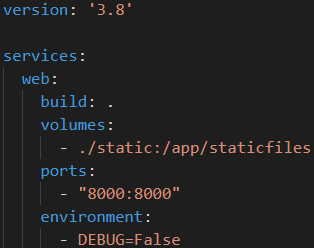
Specify the Docker Compose file format version to use version **3.8**.

#### 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:



### 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:



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

