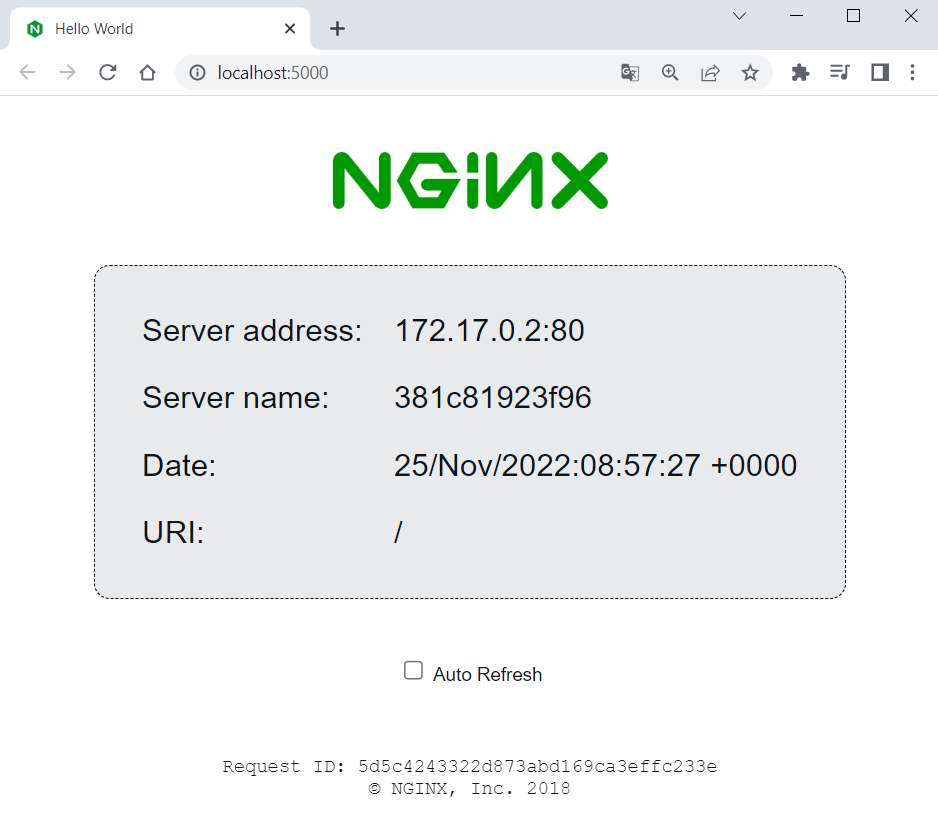
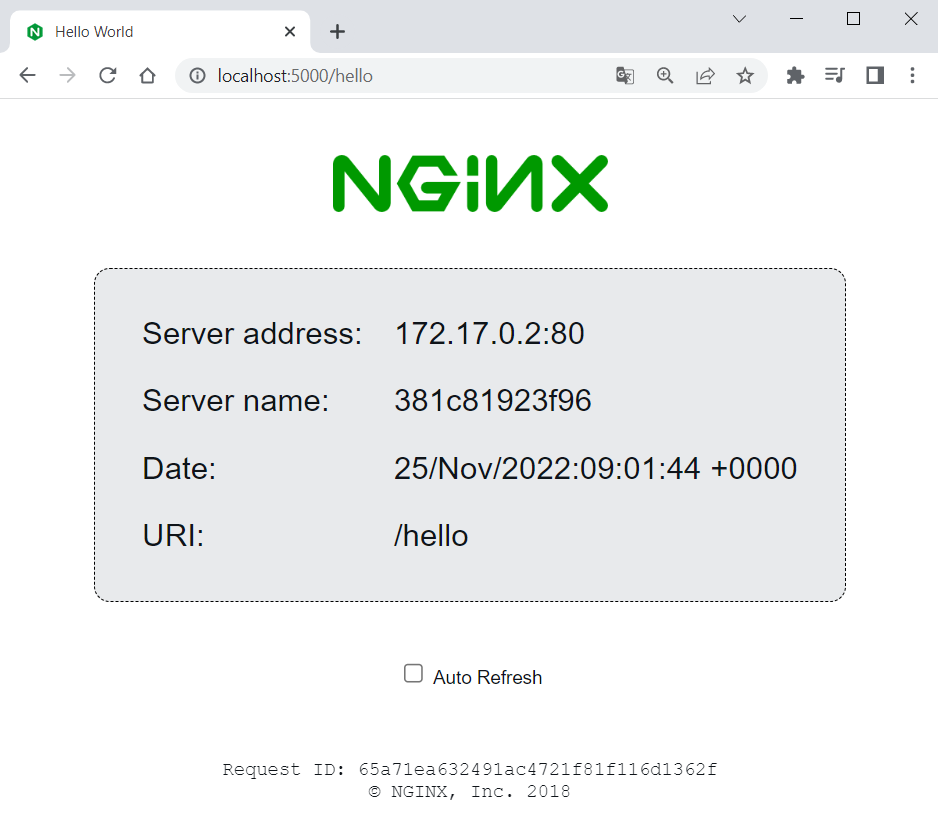
# Lab: Containers and Docker

Lab problems for the ["Back-End Technologies Basics"](https://softuni.bg/trainings/4726/back-end-technologies-basics-september-2024) Course @ SoftUni.

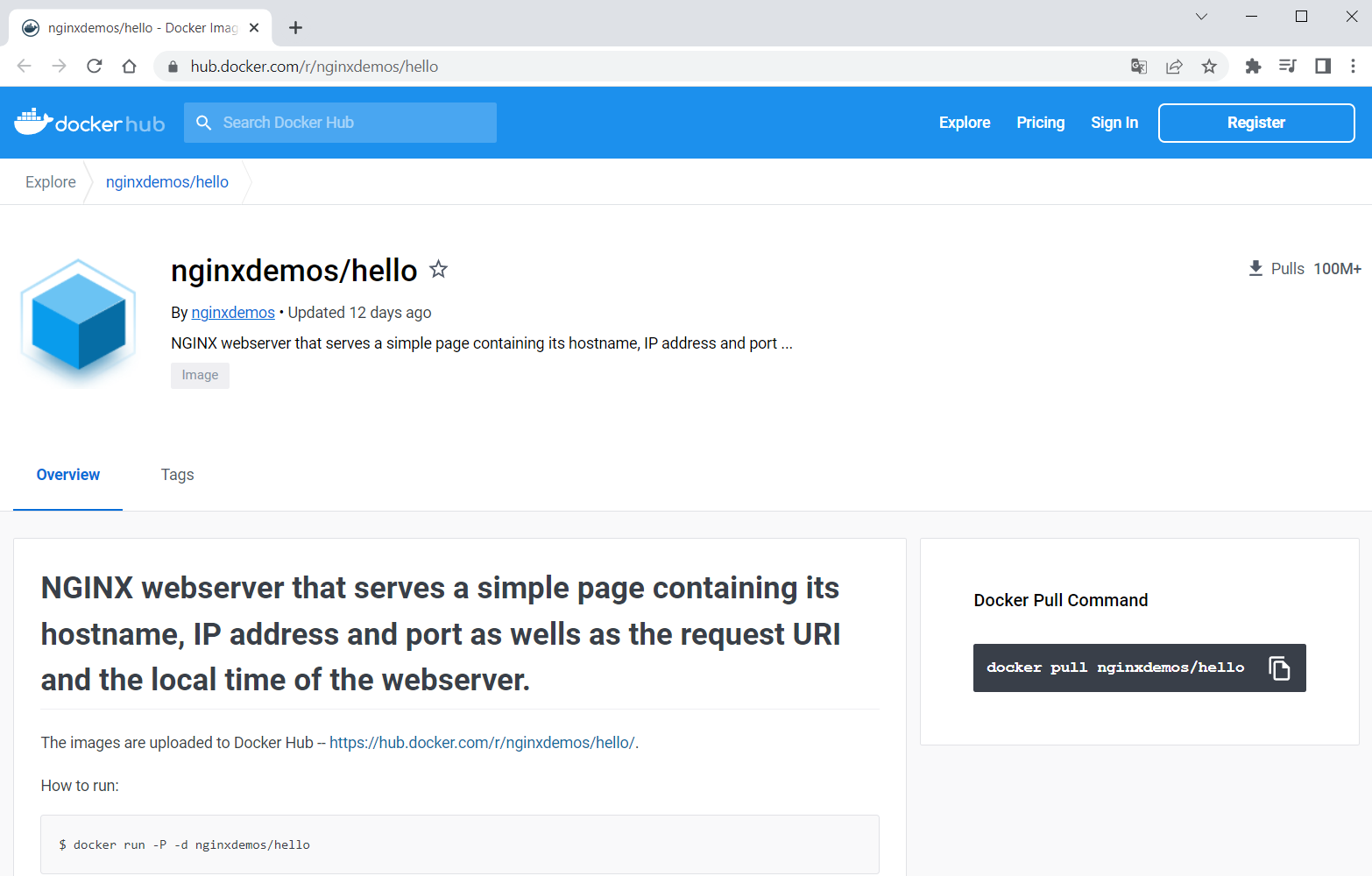
## NGINX Server Container

In this task, we will run a **simple NGINX server** in a **Docker** **container**. The **server** only returns **some server information**:

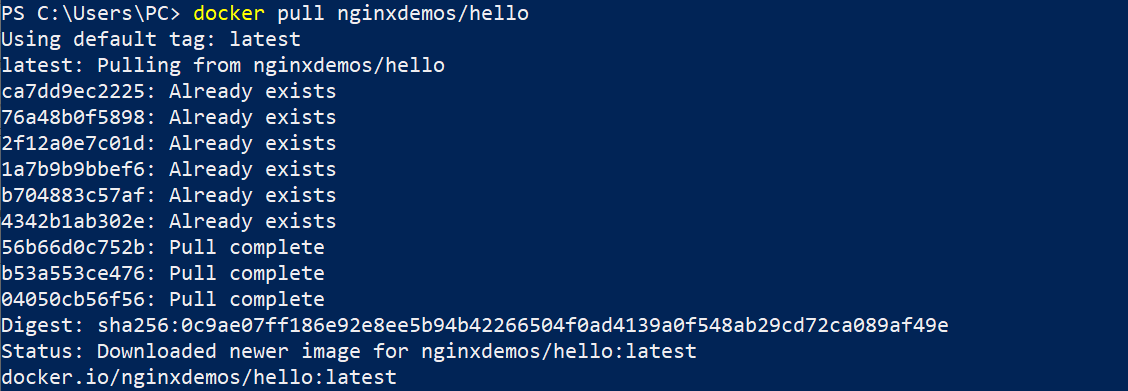
 

### Download Image

To create a **NGINX** **server** **container**, we shall first **pull the NGINX server image** from Docker Hub. You can find the image documentation on **Docker** **Hub** **here**: <https://hub.docker.com/r/nginxdemos/hello>. You can always **refer to the documentation** to get instructions on how to pull, build and run the image:

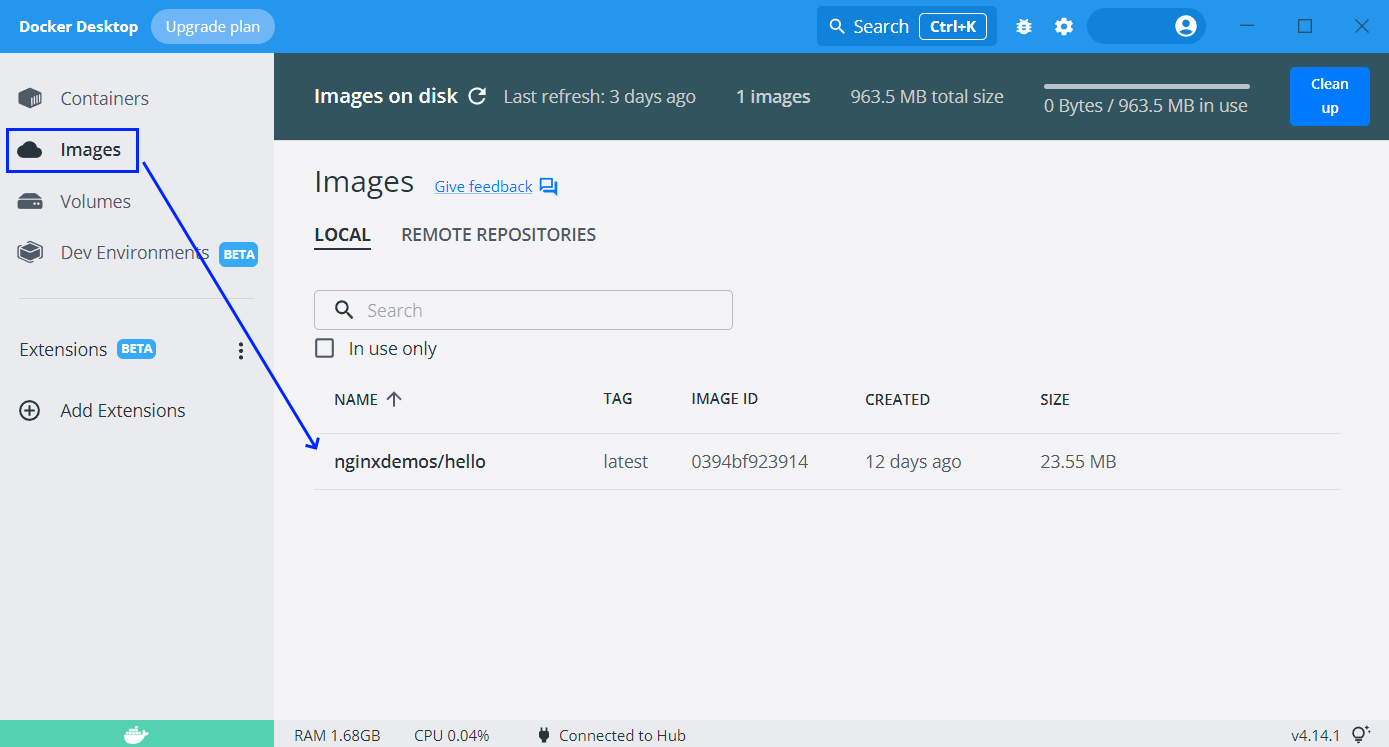


Open a **CLI**, for example, **PowerShell** or **Windows Terminal** or **Command Prompt** (**PowerShell** is recommended), and let's first **pull the Docker image** of the server. You should just use the **pull** **command** from the documentation:

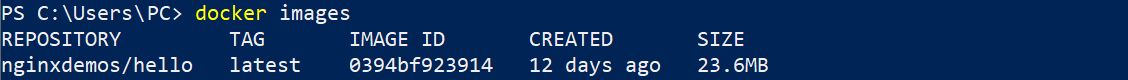


You can see that the **latest image is downloaded by default**. In addition, some of the **image layers** existed from before (from other installations), so they **were not downloaded again** – this is the advantage of Docker image layers.

You can look at your **downloaded** **images** in **Docker** **Desktop**, in the **[Images]** **tab**:

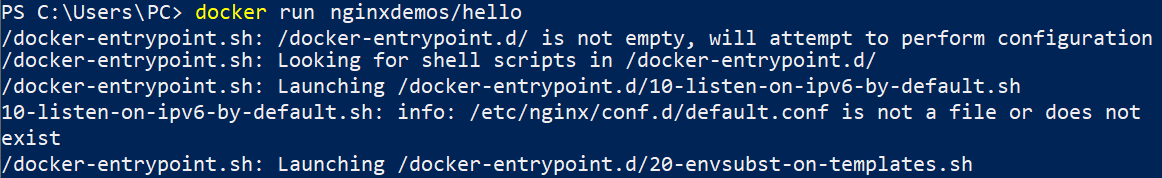


You can also use the command below to **display a list of all images** you have:

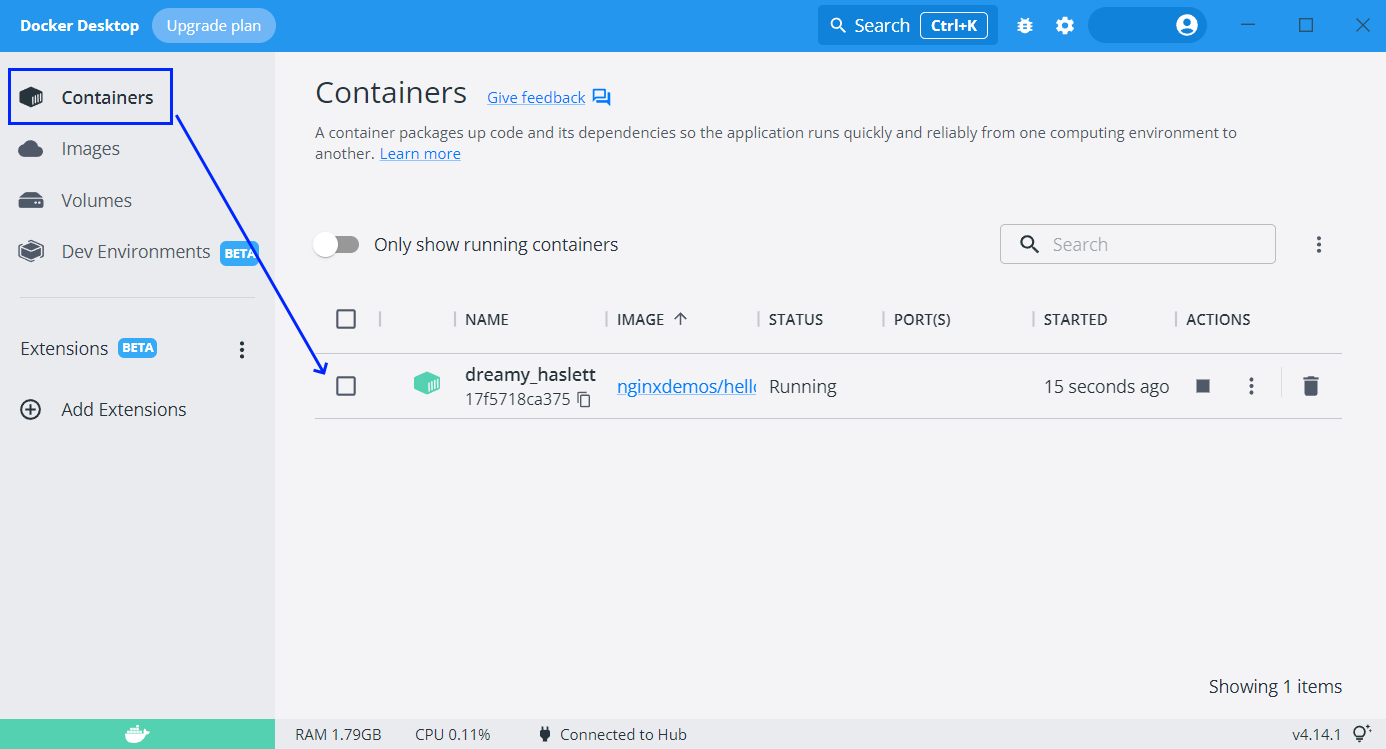


### Run a Container

Now we want to **run a container** with the **NGINX server image**, which is already downloaded from Docker Hub to our local machine. Use the **docker** **run** command and try this way:

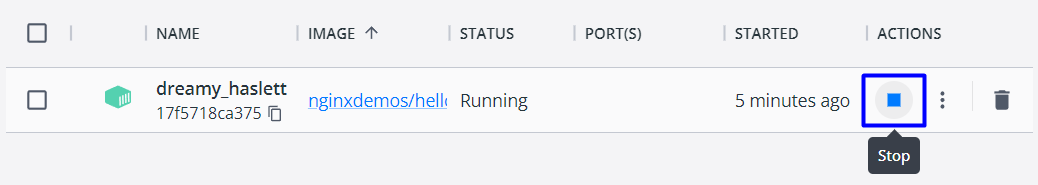
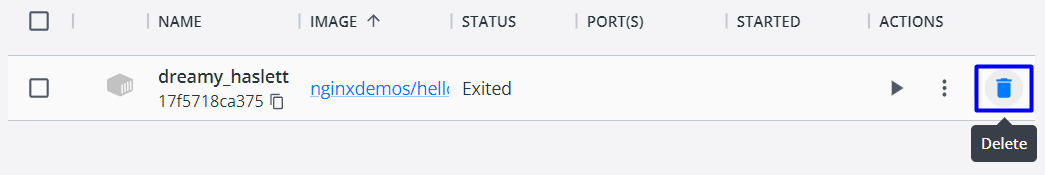


Now we have a **running container** with a **random name**, as we did not set it explicitly. We can see it on **Docker** **Desktop** **[Containers]**:

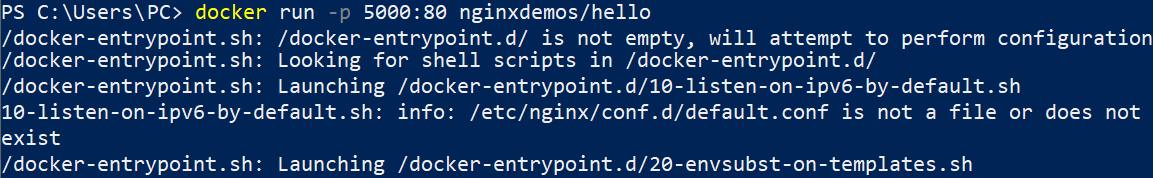


However, you can see that the "**PORT(S)**" **column** **is empty**, which means that **our container cannot be accessed** through the Internet, as it is isolated.

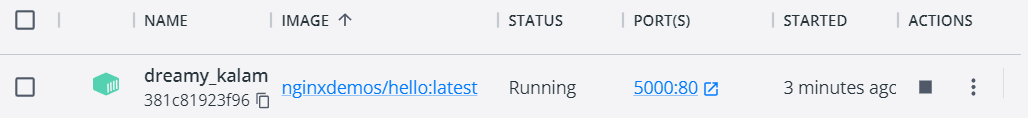
To fix this, we should **expose a port from the container**. But first, let's **stop** **and** **delete the container** we already have by **clicking on the buttons** in **Docker** **Desktop**:

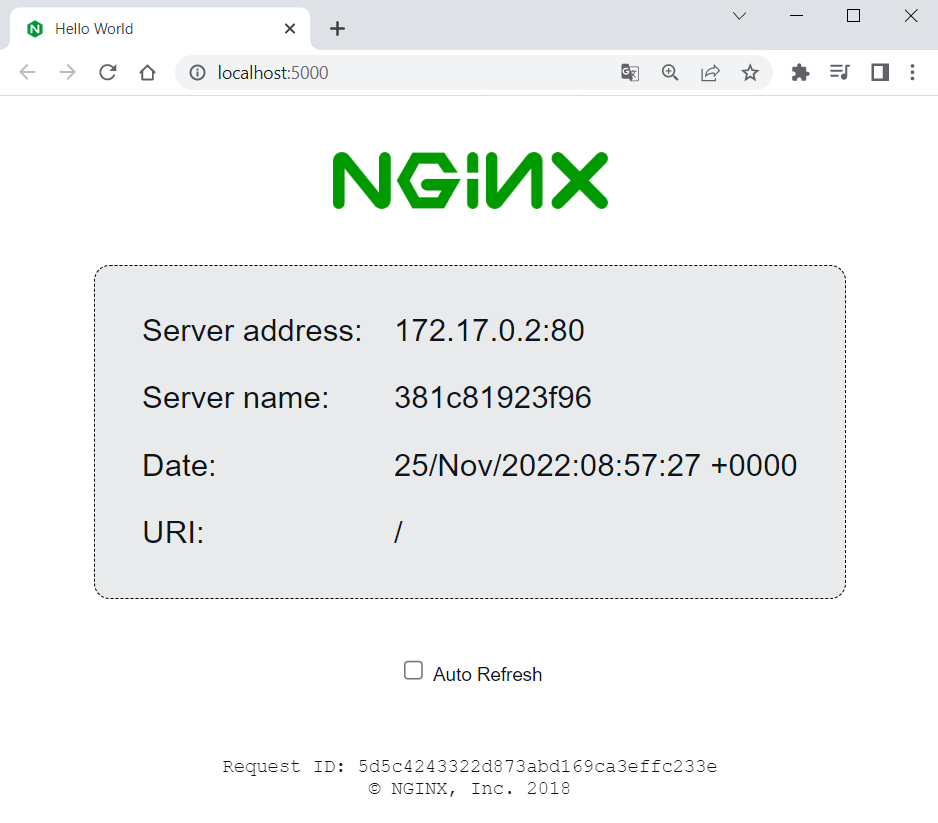
Let's **create another container** and **publish its port to the host**. This is done with the **-p** **option**. After it, we shall add the **port we want the server to be on our machine** (it can be any free port, but let's use **5000**) and the **internal** **port the server uses** – in our case **80**. Do it like this:



Now the **container is exposed**:



So, you can go to **http://localhost:5000** and **access the working server**:



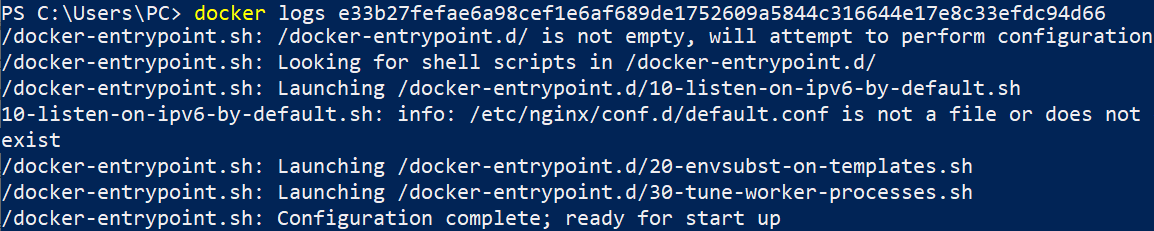
**Stop and delete the container** again and let's do one more thing.

### Run a Named Container in Detached Mode

This time, we want to **run a container** with a **name** and **in detached mode** (the container will be running in the background). To do this, use the **--name** **option** with the **container name you want**, and the **-d** **option** for **detached mode**:



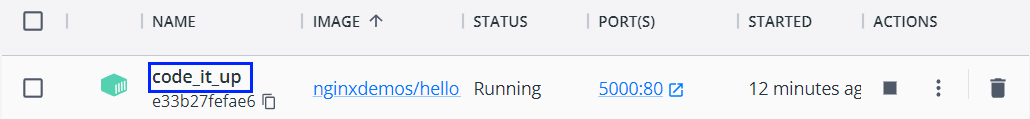
As you can see, now we have only the **container id** returned in the terminal and the **container logs are now shown** (because of the **detached mode**). However, you can **see the logs** with **docker** **logs** and the **container id or name**:

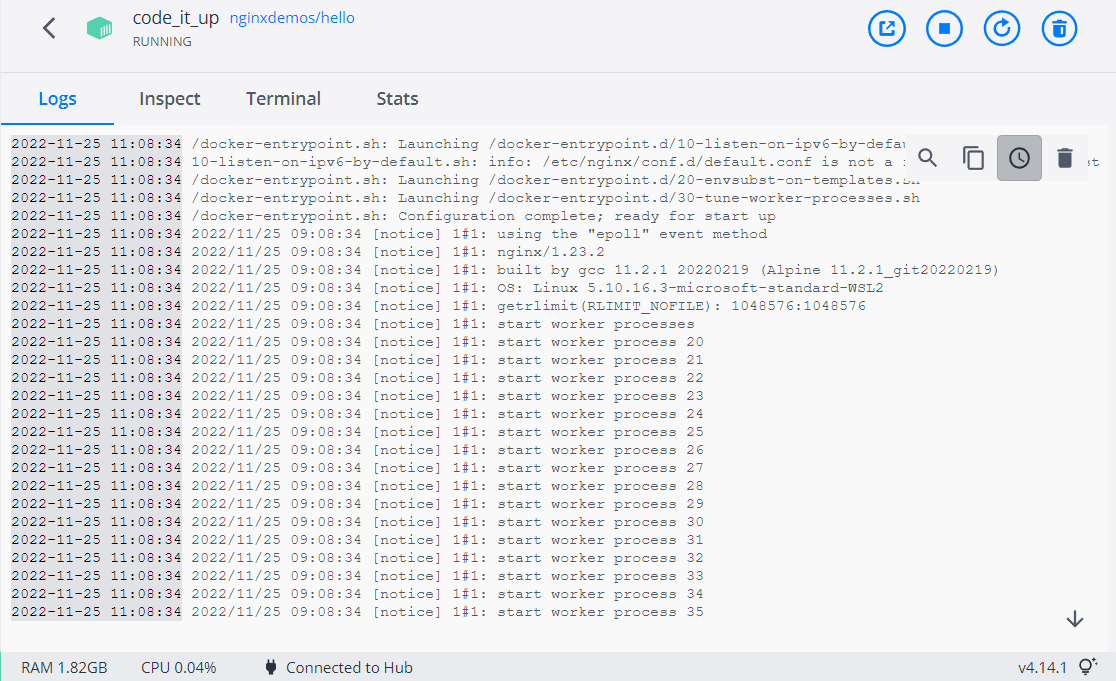


You can also use only the **first two symbols of the container's id**, not the whole one when they are unique (we have no other container with the same first symbols):



Or you can see them directly from **Docker** **Desktop** when you **click on the container's name**:

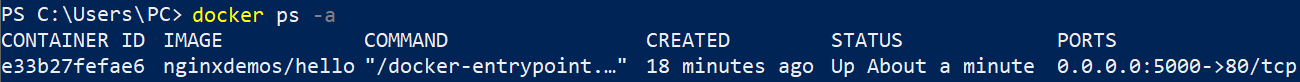




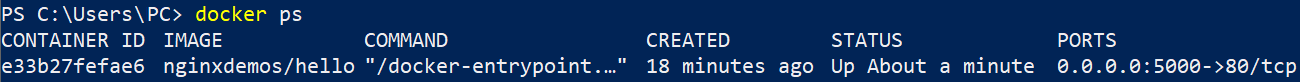
Notice that the **name of our container** is how we explicitly set it to be.

### Examine and Delete Container and Image

We can **see all containers** we have like this:



To **see** **all running containers**, use:



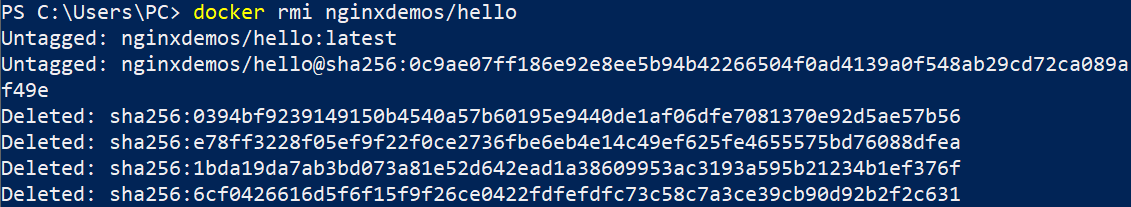
In our case, we have a **single running container** and we have the **same output** from the two commands.

Now let's use the terminal to **stop and delete our container**. Use the following commands with the **container id** or **container name**:

Both commands return the **id or the name of the container**.

Finally, we can also **delete the NGINX server image**:



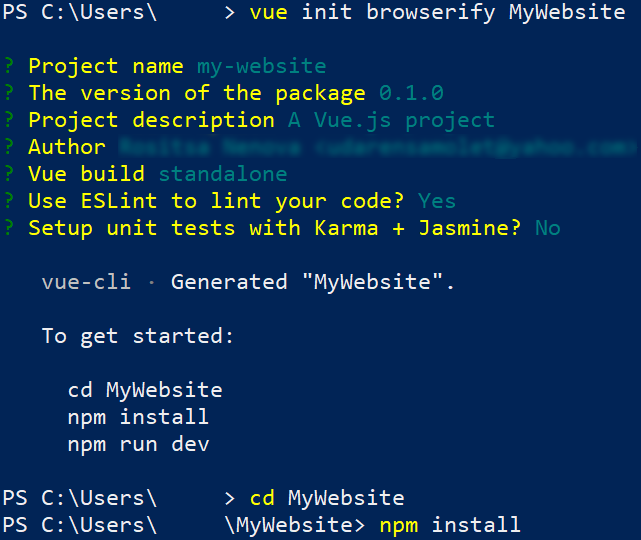
That's how we are supposed to work with **Docker images and containers** at a basic level.

## Vue.js App in Container

Now we will see how to run a **Vue.js app** in a **Docker** **container**. We will **create an app just for the demo**.

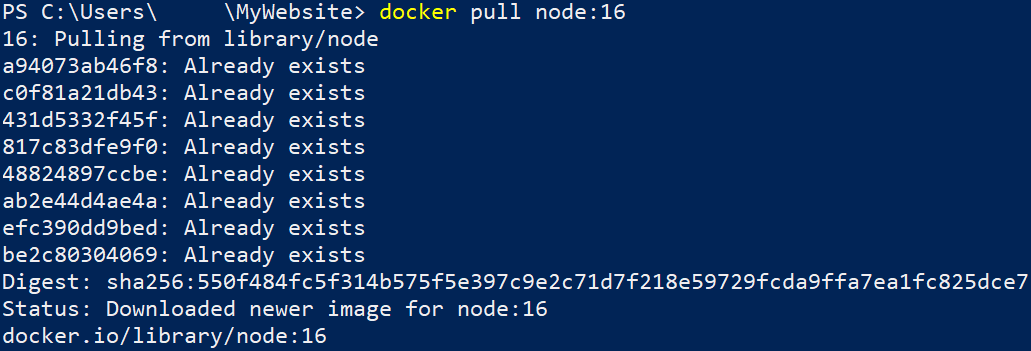
### Create and Set Up a Vue.js App

Let's use Terminal to create an app called **MyWebsite**, in a folder you choose:

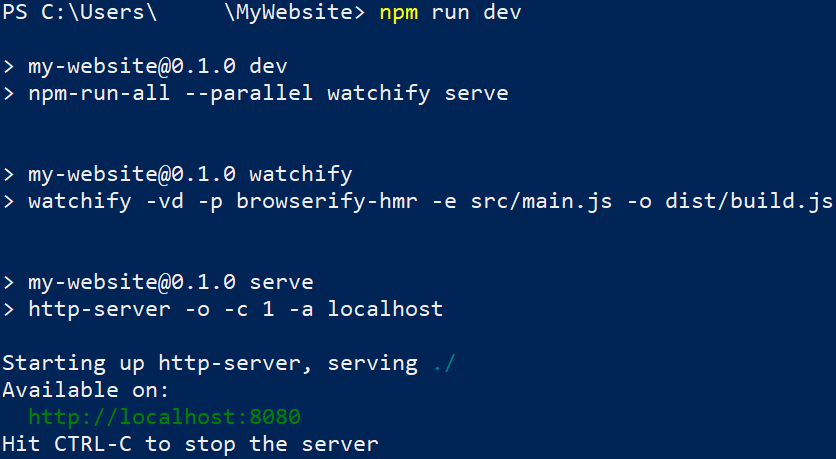


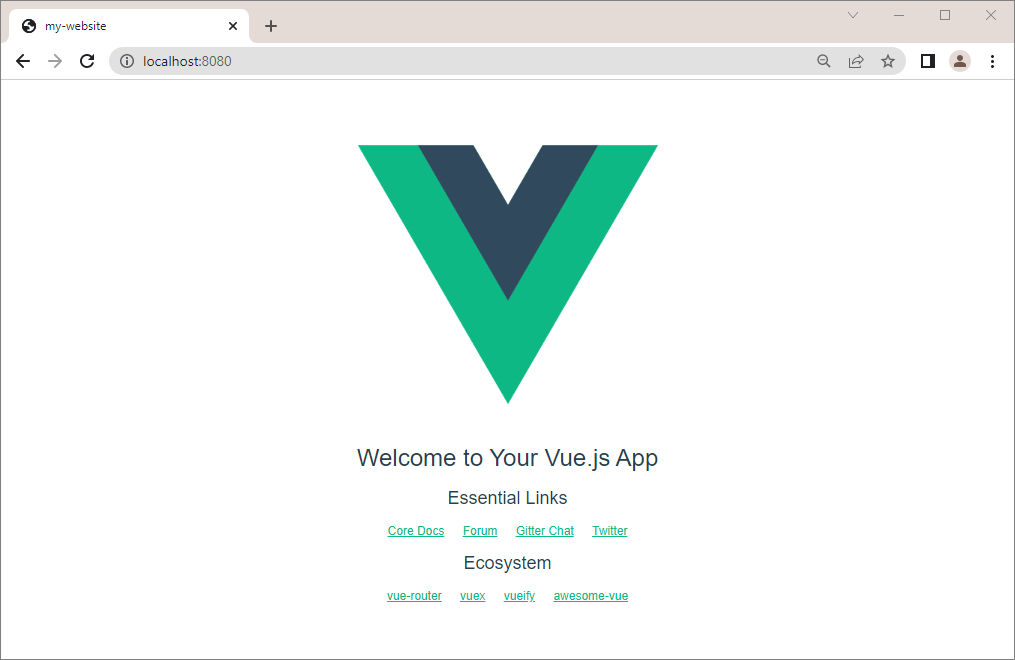
Let's now **pull the image** we will need to run the app – **NodeJs**.

(<https://hub.docker.com/_/node>). Note that the **image version** should be **the** **same** as the **app's Node version**:

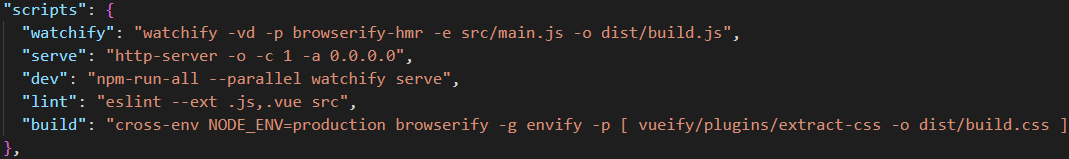


Now let's **run the application locally** in the standard way to check if **everything works as expected**:





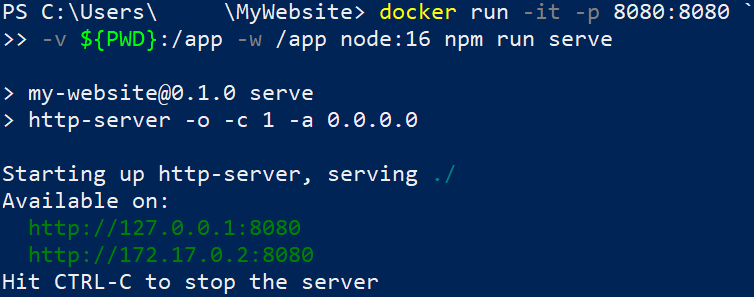
Now we should **modify the app's** **package.json** **file**, so that the **app runs on the IP we want**. **Open the file** in any editor and **change the scripts section** **settings** like this:



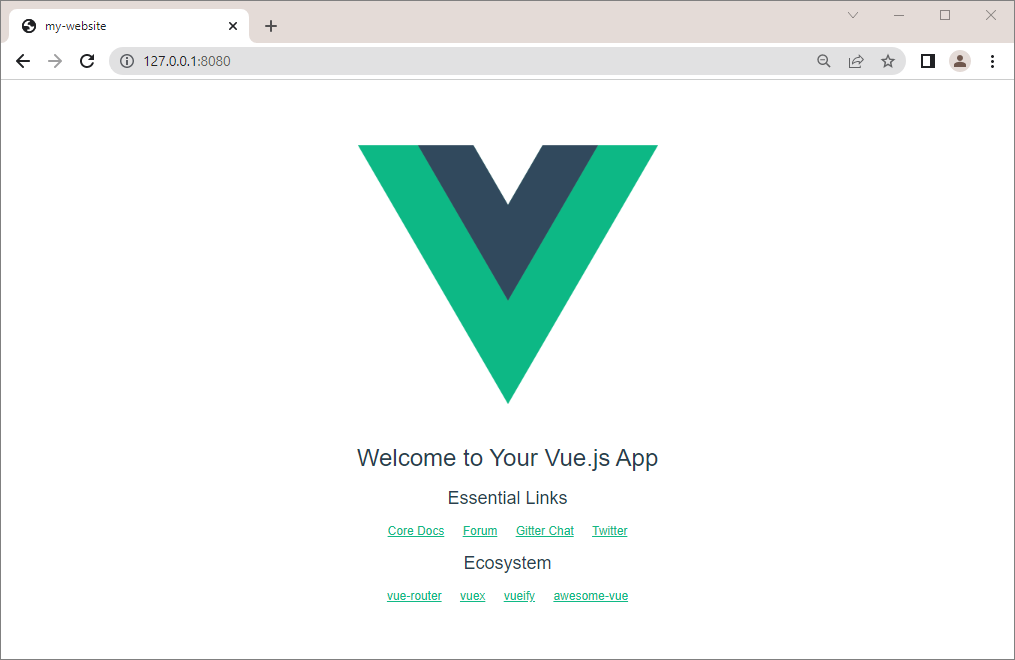
### Run the App in a Container

To run the Vue.js **app in a container**, we will need to **create a container** with an **exposed port**, a **volume** and an **interactive shell**, so that we can **run the app inside the container** with the **docker** **run** command.

To do so, execute the following command:



**Access** the app at [**http://127.0.0.1:8080**](http://127.0.0.1:8080) to validate that the app is running:



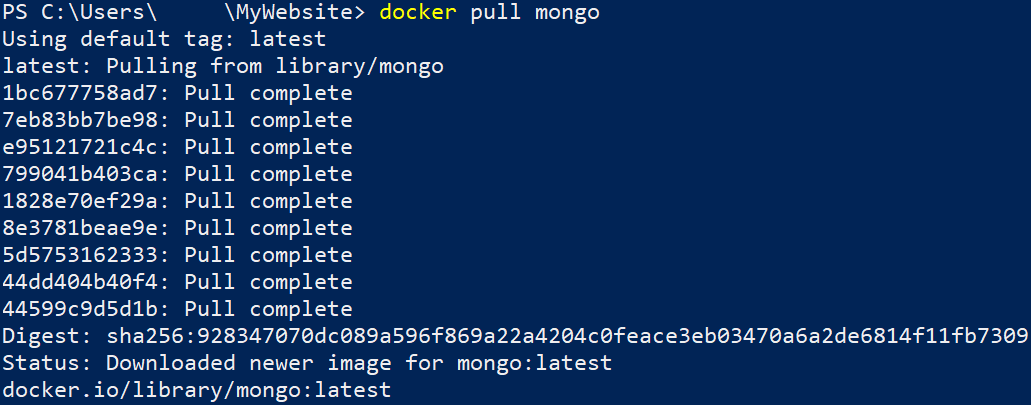
## MongoDB in Container

Our next task is to **run a container** with a **Mongo** **database** in it. To do this, we will need the **following image** from **Docker** **Hub**: <https://hub.docker.com/_/mongo>

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** the process step by step.

### Create the Container

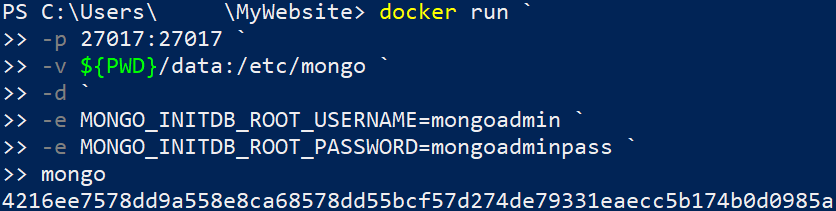
First, pull the latest MongoDB image with the **docker** **pull** **mongo** command:



You can examine the **documentation** on how to use the image: <https://hub.docker.com/_/mongo>

### Run a Database Container

Our next step is to **run** the container, using the following command:



* **docker run** → starts a new Docker container;
* **-p 27017:27017** → sets the external and internal ports to **27017**, so that we can access the MongoDB from outside the container;
* **-v ${PWD}/data:/etc/mongo**
  + ${PWD} → the host directory;
  + **/etc/mongo** → the container directory;
* **-d** → runs the container in detached mode (it will run in the background);
* **-e MONGO\_INITDB\_ROOT\_USERNAME=mongoadmin** → sets the admin username;
* **-e MONGO\_INITDB\_ROOT\_PASSWORD=mongoadminpass** → sets the admin password;
* **mongo** → specifies the image.

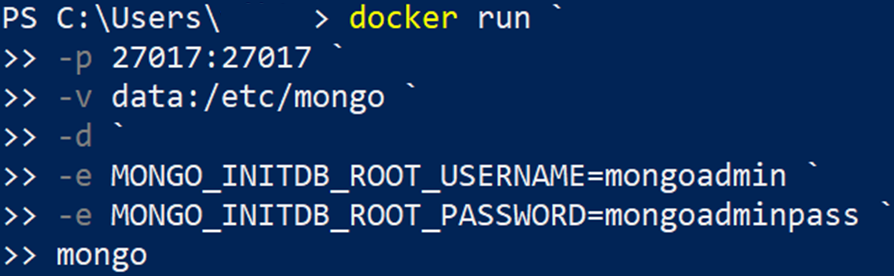
You should disable host's MongoDB Server instances or use another port!

Admin password should always follow the rules from the documentation.

When MongoDB Server container is started, other apps can log in to it and use the database.

**Run a Database Container with Volume**

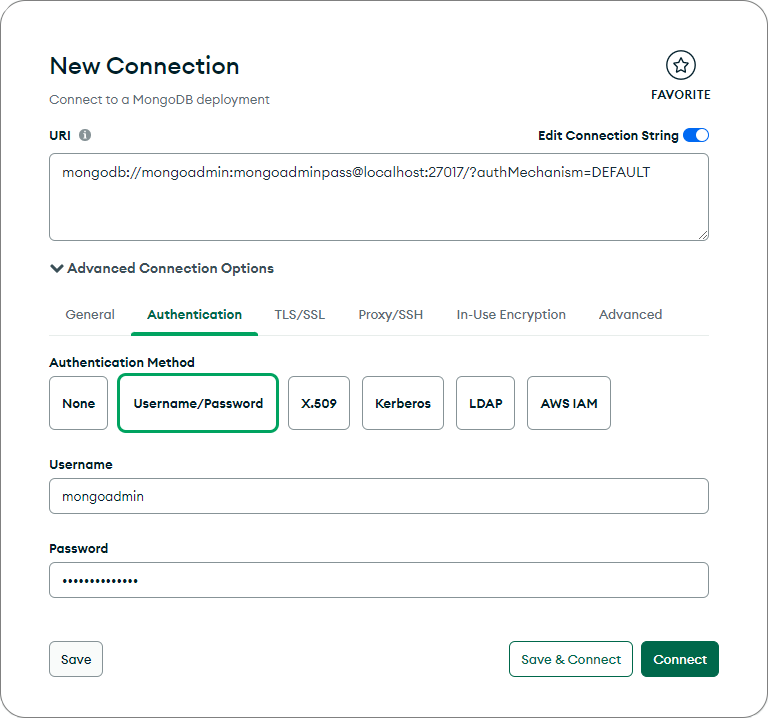
To persist data after container is stopped, **create a volume, using the following command:**

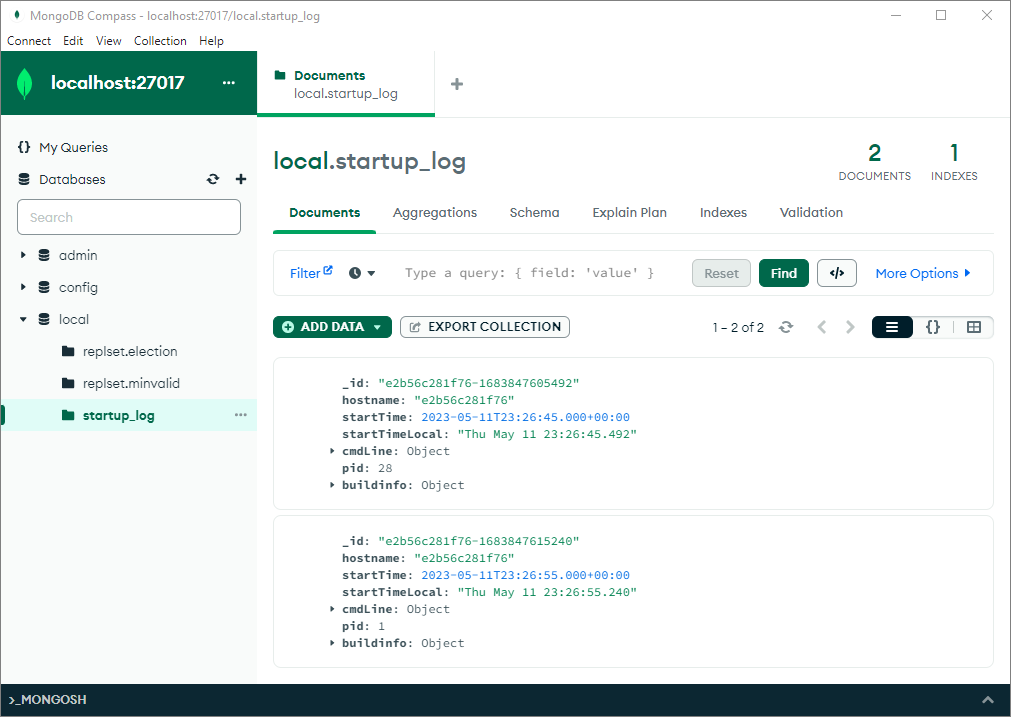


You can then easily **backup** or **restore** the **data** from the volume.

### Connect to the Database Container through MongoDB Compass

You can connect to the container database in **MongoDB Compass,** using the **username** andpassword that we created in the **previous** step**:**

****

****

## MyWebsite App: Building a Custom Image

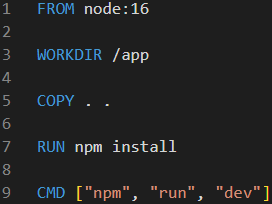
### Step 1: Create a Dockerfile

Our first task is to **create a Dockerfile for a Vue.js app**, which will allow us to **run it in a Docker container**.

First, we have to go to the **root** folder of the **Vue.js** app that we created and ran in our previous session.

Our next step is **creating a Dockerfile** in this **directory**. The **Dockerfile** **contains instructions** on how an **image for the app should be created**. As we know, **Dockerfiles** are **just text files**, so we can create our own and open it with a text editor of our choice. Note that the name of the file should be "**Dockerfile**" without any extensions.

The content should be as shown below:



Each Dokerfile starts with "**FROM**", so we start creating an image, based on the existing image **node:16.**

After that, we will **set** the **"app" folder** as the **current working directory** and we'll **copy** all of the **project files and folders to it.** This will add a layer.

Then, we run the **npm install** command in order to **install the necessary dependencies**, so that our app can run. This will form another layer, too.

Finally, we execute the **npm run dev** commands, in order to run the scripts that defined inside our app's **package.json** file.

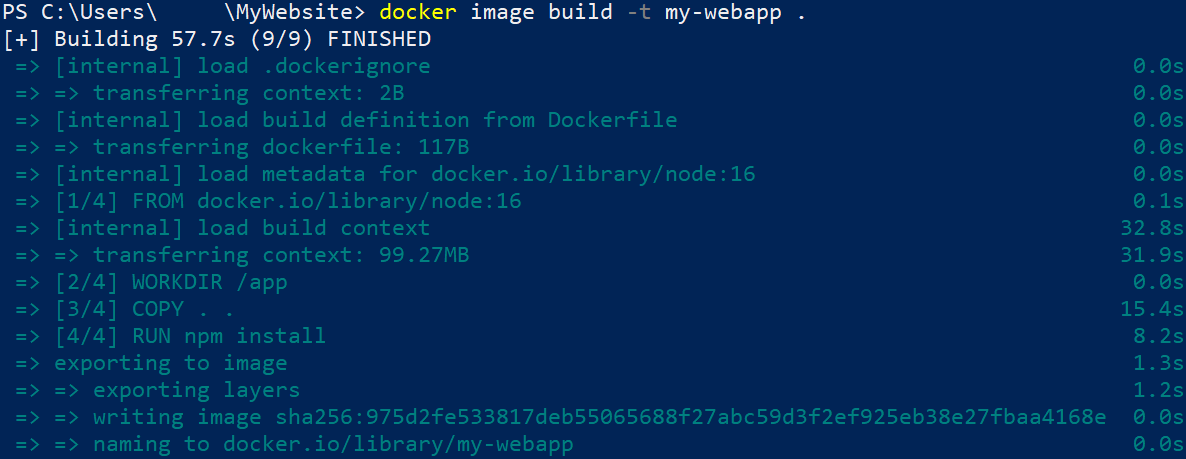
### Step 2: Build and Publish the Image to Docker Hub

#### Build the Image

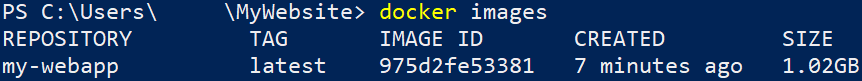
We can now **build a custom image** with this **Dockerfile**. Open a CLI, for example **Powershell**, and fulfill the **following steps** to do it:

* Navigate to the **MyWebsite** **directory**
* Use the **docker image** **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 like this:



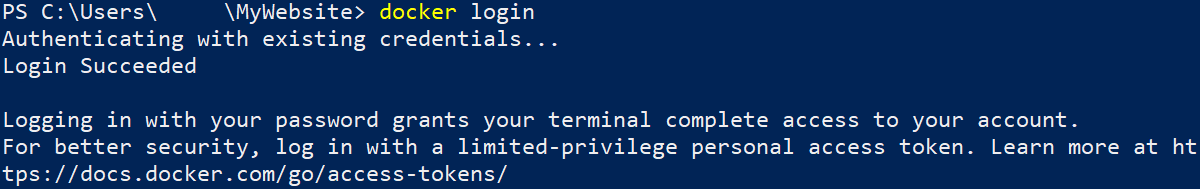
Note that we can examine how the **instructions from the Dockerfile** are executed to **build the image**.   
We can check the **ready image** with the **docker** **images** command.



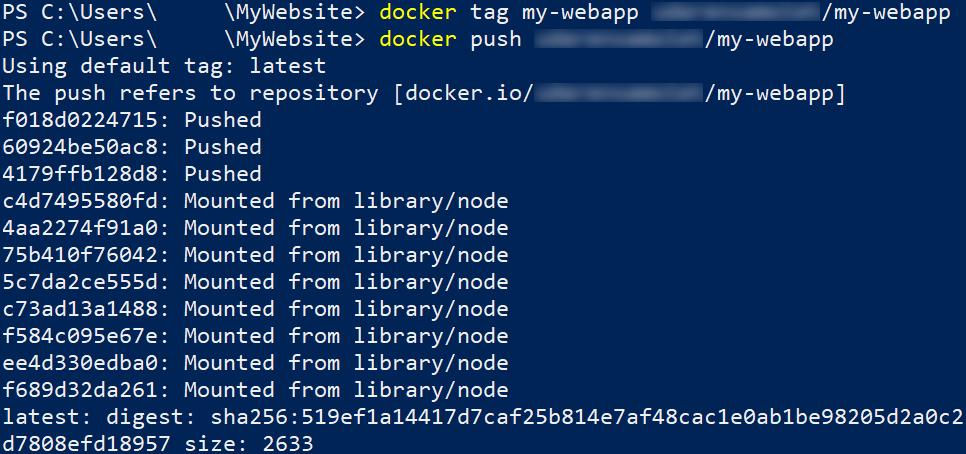
#### Publish the image

Now let's see how to **push our custom image** **to** **Docker** **Hub**. Note 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**.

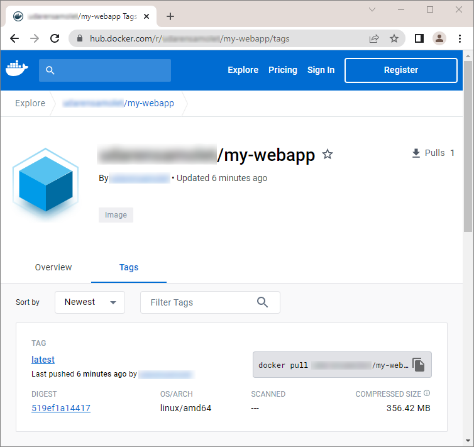
To **push our image to Docker** **Hub**, we should first **log-in to Docker Hub** with the command below. If this is the **first time** you log in, you should **enter your credentials**. Make sure that **login is successful**:



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



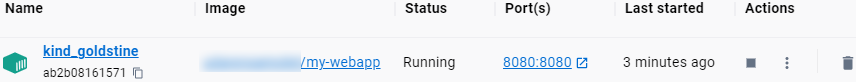
And it now is **available at Docker** **Hub** as a **public image**:



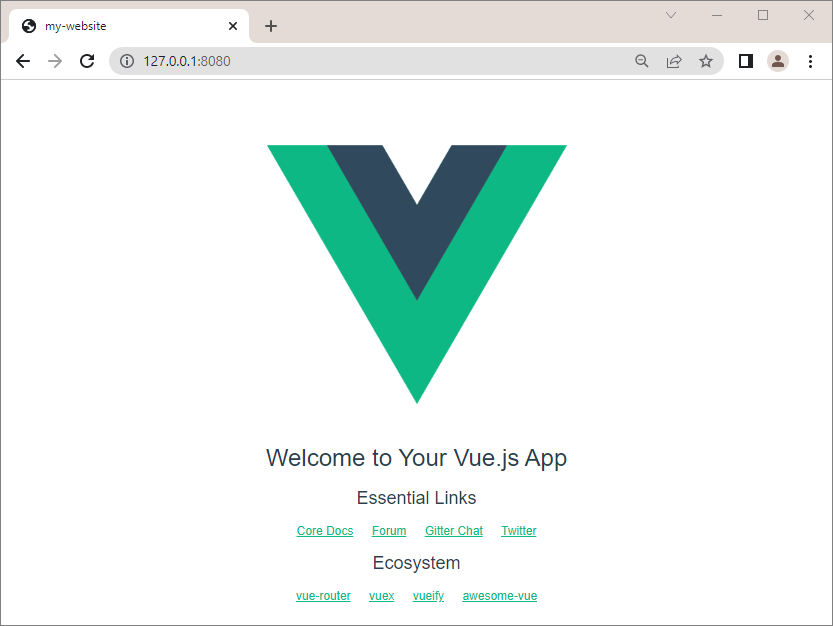
#### Run the Image as a Container

Finally, let's run the newly-created **image** as a **container** on the **right port**, using the command below:





Open your web browser and go to **127.0.0.1:8080**. You should be able to see the **running Vue.js app**:



## WordPress App with MySQL Database: Connecting Containers in a Network

In this exercise, our task is to **set up and run a WordPress container** in **Docker** with a **MySQL** **database** by connecting them in a network.

### Step 1: Create a Network

First, we have to create a network. Open a CLI and first create a new folder, which will contain the files for our app. Then, create a new network with the **docker network create {network\_name}** command.

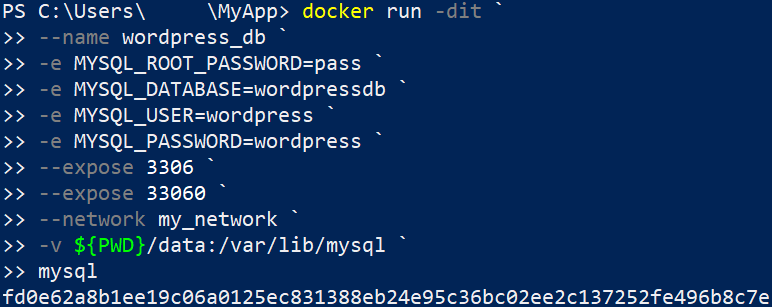


### Step 2: Add MySQL Container to Network

Our next step is **adding** the **MySQL** container to the network that we just created.

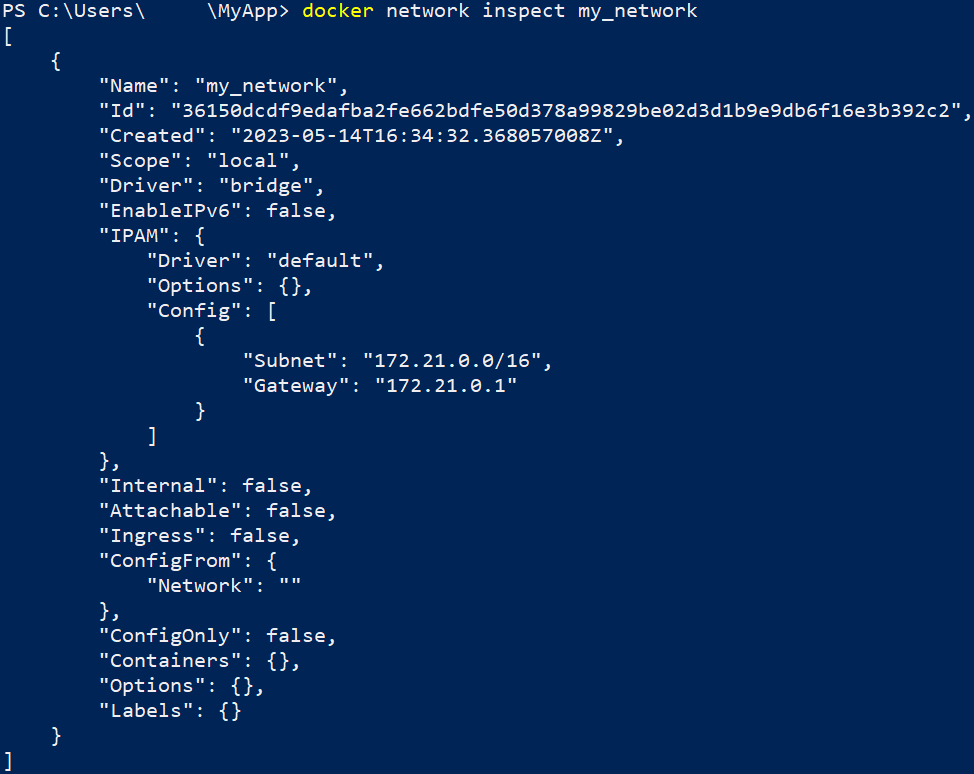
The **commands from the resources** are the following:

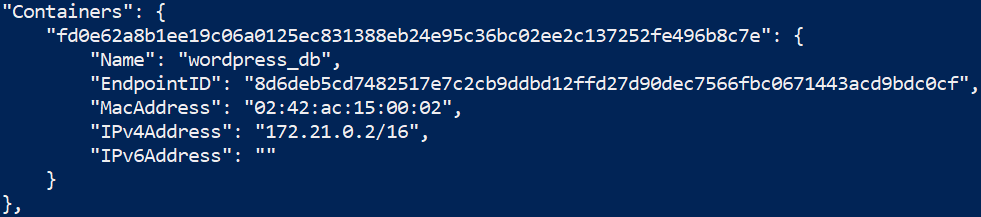
* **docker run -dit** → runs the image detached and in interactive mode;
* **--name wordpress\_db** → names the container **wordpress\_db**;
* **-e MYSQL\_ROOT\_PASSWORD=pass** → sets the password for the root MySQL user;
* **-e MYSQL\_DATABASE=wordpressdb** → sets the name of the MySQL database that we'll use for the WordPress installation;
* **-e MYSQL\_USER=wordpress** → sets the MySQL user that we'll use for the WordPress installation;
* **-e MYSQL\_PASSWORD=wordpress** → sets the password for that user;
* **--expose 3306** → sets the port of the container;
* **--expose 33060** → sets the SSL port of the container;
* **--network my\_network** → sets the network that we want to attach our container to;
* **-v ${PWD}/data:/var/lib/mysql** → maps the directory on our local machine to the directory of the container, so that we can store data outside of it;
* **mysql** → the name of the image.



### Step 3: Inspect Network

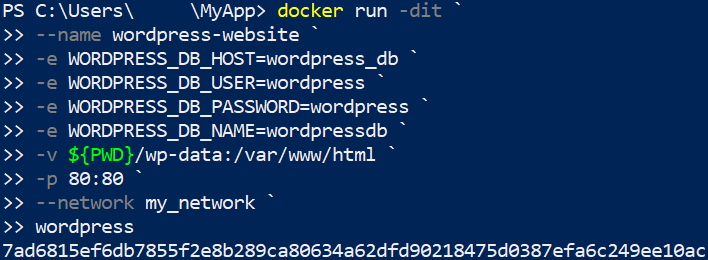
Now let's inspect our network in order to check if our **wordpress\_db** container is attached to it:





### Step 4: Add WordPress Container to Network

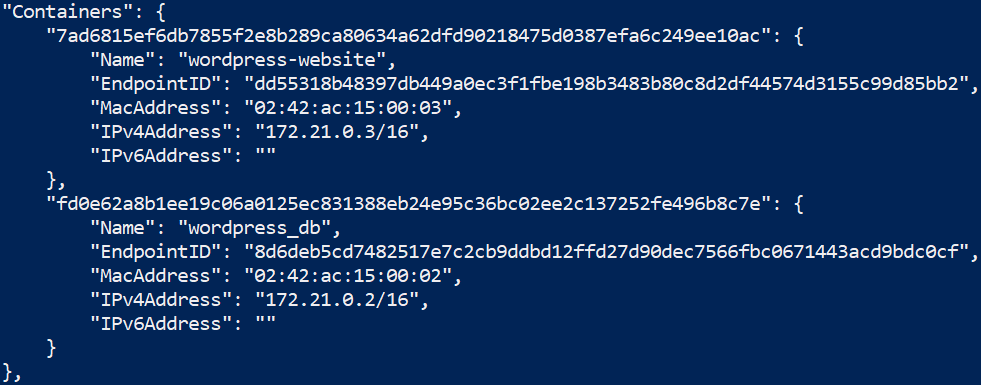
Our next step is adding the WordPress to our network. You can do it with the following command:



* **-e WORDPRESS\_DB\_HOST=wordpress\_db** → sets the WordPress database host, which matches the name of our MySQL container that we set up in Step 2;
* **-e WORDPRESS\_DB\_USER=wordpress** → sets the WordPress user that we previously set up;
* **-e WORDPRESS\_DB\_PASSWORD=wordpress** → sets the password for the user;
* **-e WORDPRESS\_DB\_NAME=wordpressdb** → sets the name of the WordPress database, that we created in Step 2.

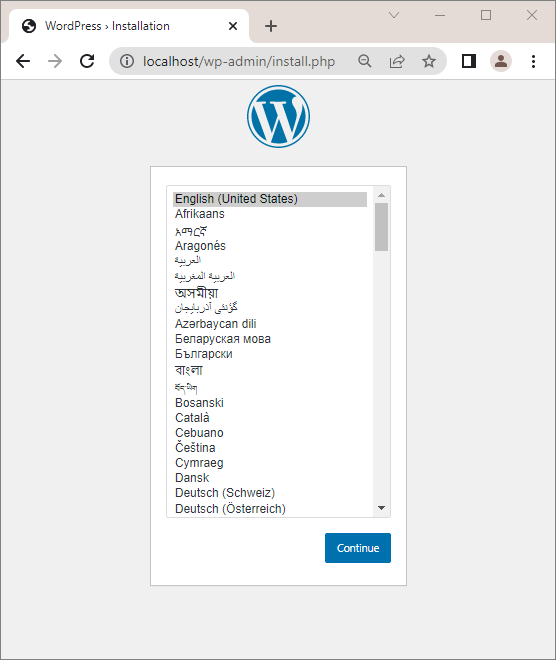
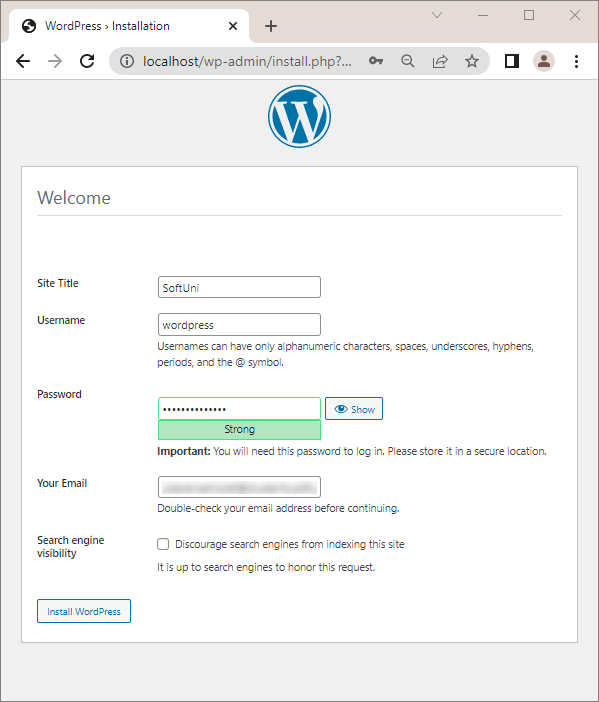
### Step 5: Inspect Network

Now, if we execute the command for inspecting our network, we should see that the two containers are attached to it:

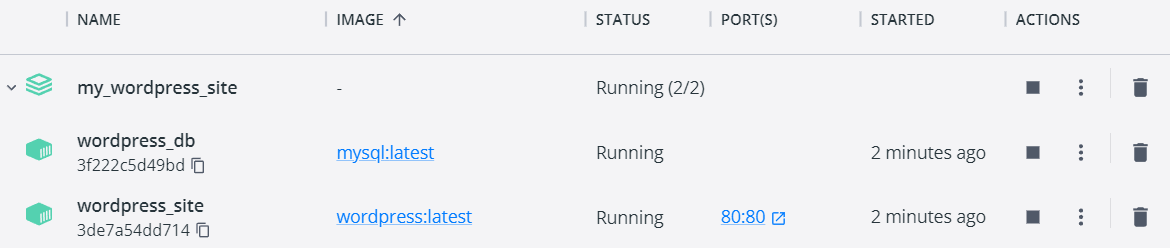


### Step 6: Run the App

You can **access the WordPress site** on **http://localhost:80** and you will see the **WordPress setup page:**

****

When you check **Docker Desktop**, you should be able to see the **two containers** combined in a **single network**:



## WordPress App with MySQL Database: Docker Compose YAML file

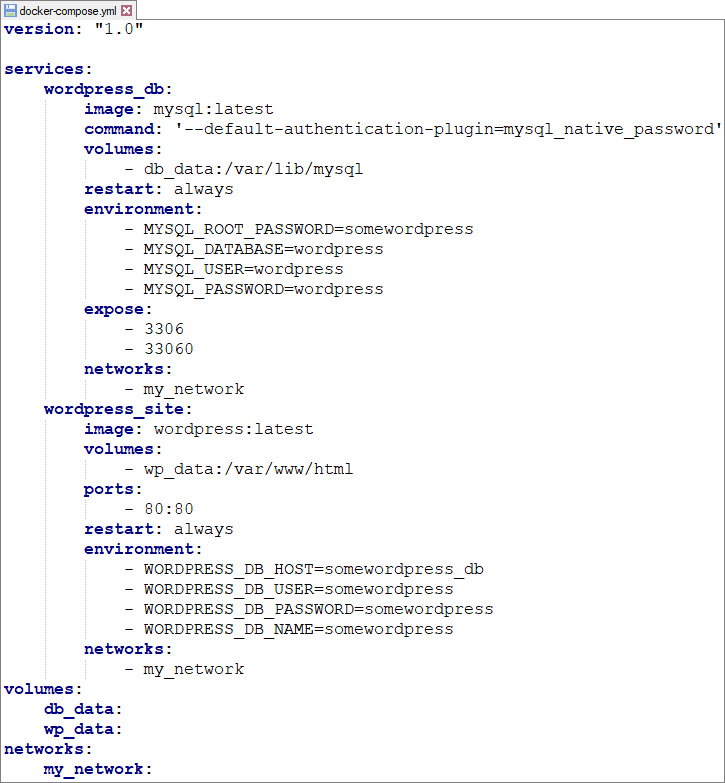
Our next task is, using **Docker Composer**, to **create a docker-compose.yaml file** with instructions for **creating the MySQL and WordPress containers** from the previous task**, together** in a single network, called **my\_network**.

### Step 1: Create a Network

Using the **docker network create**, create a new network, called **my\_network.**

### Step 2: Create the docker-compose.yml file

First, we have to create a **docker-compose.yml** file in the folder of our app. The docker file should look like this:

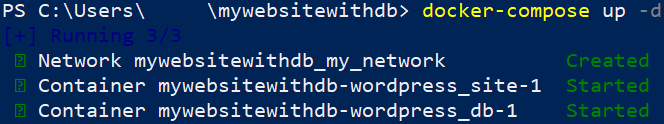


### Step 3: Build and Run the Multi-Container App

Next step is building and running our multi-container app. First, build all of the images with the **docker-compose** **build** command:

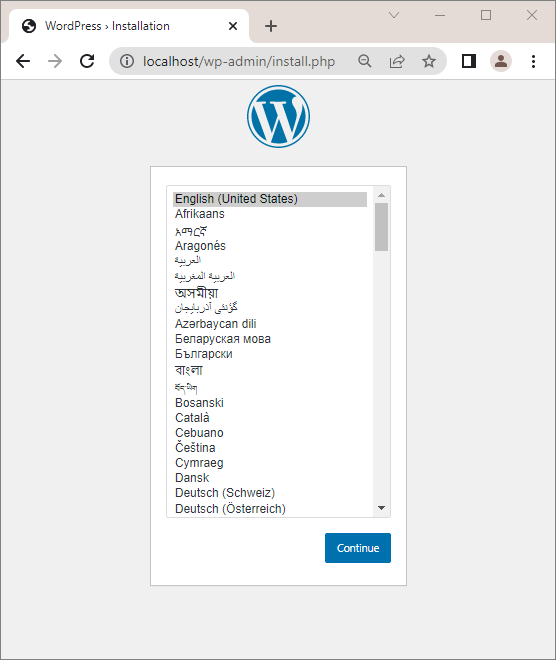


Then, run the containers with the **docker-compose** **up** or **docker-compose up -d command**.



### Step 4: Run the App

You can **access the WordPress site** on **http://localhost:80** and you will see the **WordPress setup page.**



You should be able to **configure and create your website**, following the guide:

