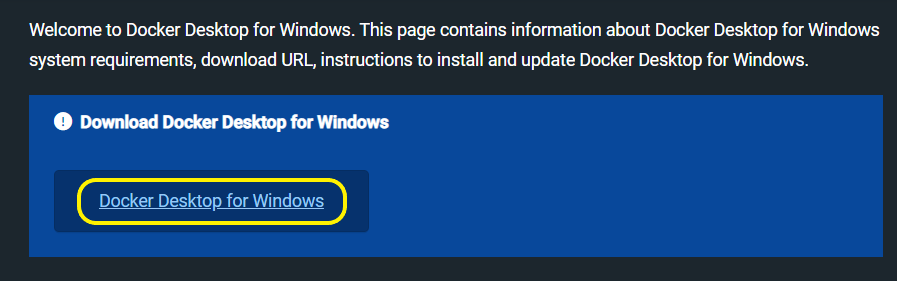
**Docker**

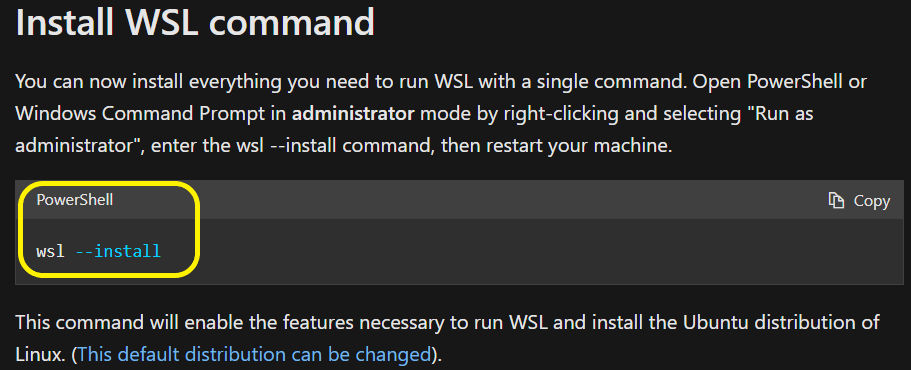
1. [Download and install Docker Desktop on Windows x64 and Docker Engine on Ubuntu and Amazon Linux 2.](#Task_1)
2. [Run first container “hello-world”.](#Task_2)
3. [Create first image(based on Ubuntu 20.04).](#Task_3)
4. [Create first image(based on Centos 7).](#Task_4)
5. [Create a Docker image which will run a Flask app.](#Task_5)
6. [Frequently used Docker commands.](#Task_6)
7. [Docker Compose.](#Task_7)
8. [Docker Compose project example.](#Task_8)
9. [Docker Compose. Use volumes.](#Task_9)
10. [Docker Compose. Use networks.](#Task_10)
11. Download and install [Docker Desktop on Windows x64](https://docs.docker.com/desktop/install/windows-install/) and Docker Engine on [Ubuntu](https://docs.docker.com/engine/install/ubuntu/) and [Amazon Linux 2](https://www.cyberciti.biz/faq/how-to-install-docker-on-amazon-linux-2/#google_vignette).

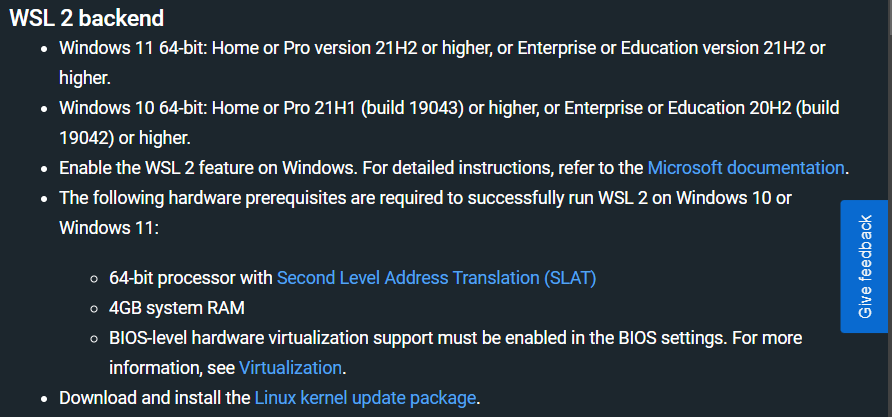
* **Windows**

1. Download newest version Docker Desktop from [website](https://desktop.docker.com/win/main/amd64/Docker%20Desktop%20Installer.exe)

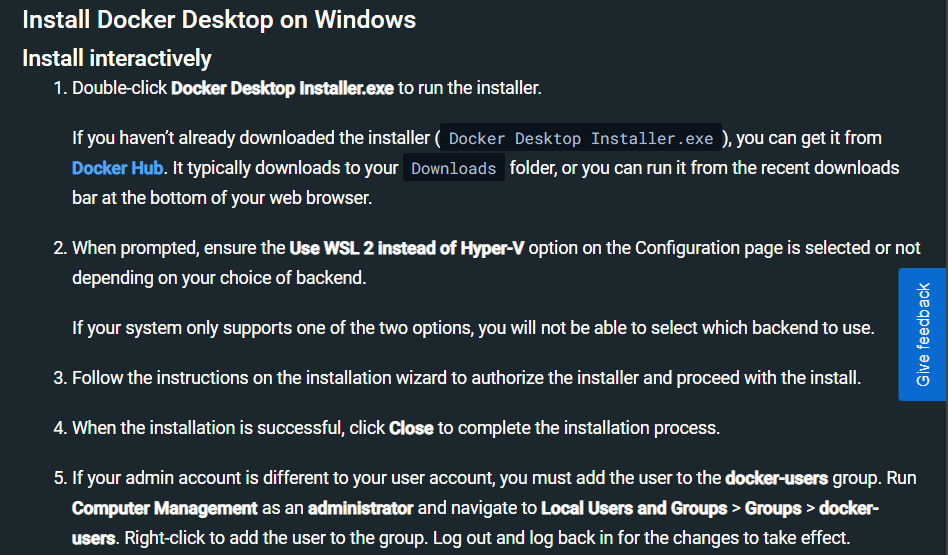


1. Before installing Docker Desktop you have to [install WSL 2 backend](https://learn.microsoft.com/en-us/windows/wsl/install)(Windows Subsystem for Linux version 2)

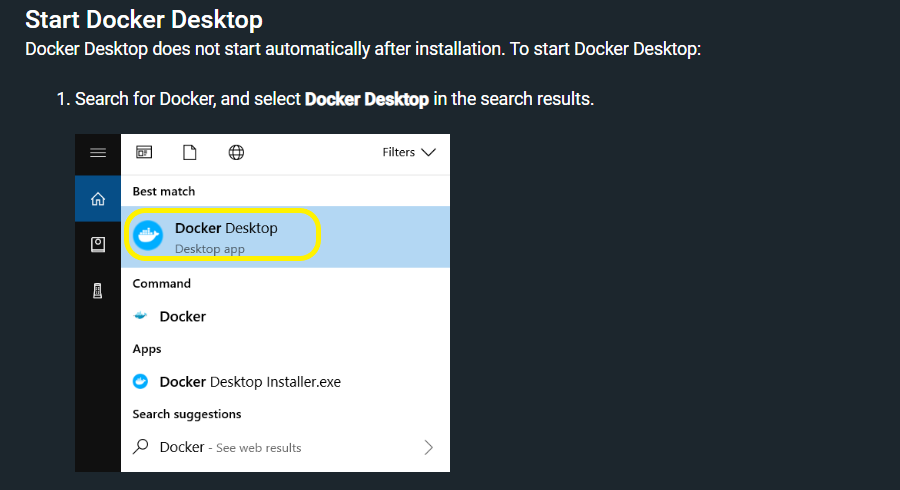




1. Install Docker Desktop on Windows



1. Start Docker Desktop



* [**Ubuntu**](https://docs.docker.com/engine/install/ubuntu/)

Install using the repository.

Set up the repository

1. Update the apt package index and install packages to allow apt to use a repository over HTTPS:

$ sudo apt-get update

$ sudo apt-get install \

ca-certificates \

curl \

gnupg \

lsb-release

1. Add Docker’s official GPG key:

$ sudo mkdir -p /etc/apt/keyrings

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

1. Use the following command to set up the repository:

$ echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

Install Docker Engine

1. Update the apt package index:

$ sudo apt-get update

1. Install Docker Engine, containerd, and Docker Compose.

$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin

1. Verify that the Docker installation is successful by running the hello-world image

$ sudo docker run hello-world

* [**Amazon Linux**](https://www.cyberciti.biz/faq/how-to-install-docker-on-amazon-linux-2/#google_vignette)

Install Docker.

1. Install Docker with the command’s below:

$ sudo yum update

$ sudo yum search docker

$ sudo yum install docker

1. Add group membership for the default ec2-user so you can run docker commands.

$ sudo usermod -a -G docker ec2-user

$ id ec2-user

1. Enable and start docker service.

$ sudo systemctl enable docker.service

$ sudo systemctl start docker.service

1. Check docker service status, run and version:

$ sudo systemctl status docker.service

$ docker version

Install docker-compose.

1. Install using pip (recommended):

$ sudo pip3 install docker-compose

1. Manually install

$ wget https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)

$ sudo mv docker-compose-$(uname -s)-$(uname -m) /usr/local/bin/docker-compose

$ sudo chmod -v +x /usr/local/bin/docker-compose

1. To verify that the installation was successful, you can use the command

$ docker-compose version

Bonus: install ctop.

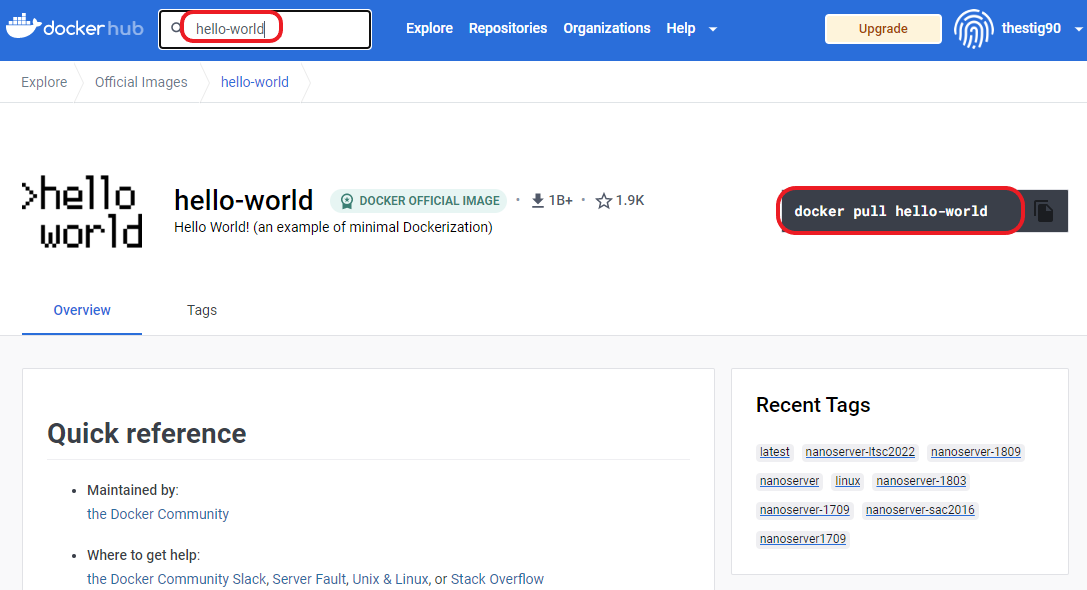
**ctop** provides a concise and condensed overview of real-time metrics for multiple containers

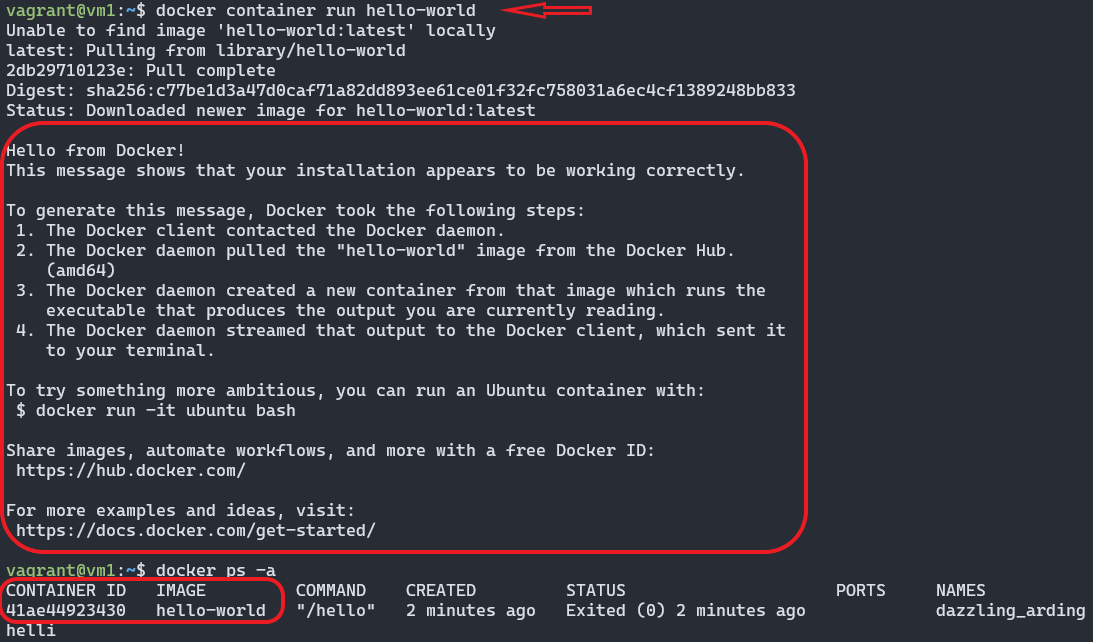
$ sudo wget https://github.com/bcicen/ctop/releases/download/v0.7.7/ctop-0.7.7-linux-amd64 -O /usr/local/bin/ctop

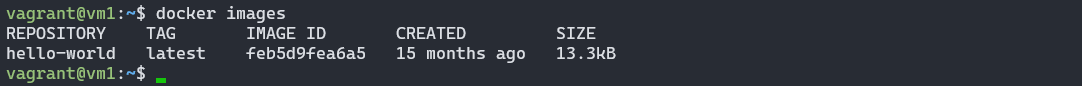
$ sudo chmod +x /usr/local/bin/ctop

1. Run first container “hello-world”.

Command $ sudo docker run hello-world pulls image “hello-world” from Docker Hub Registry and run container with console output.





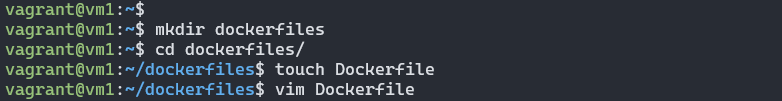


If you would like to use Docker as a **non-root user** you should now consider adding your user to the **“docker” group** with command:

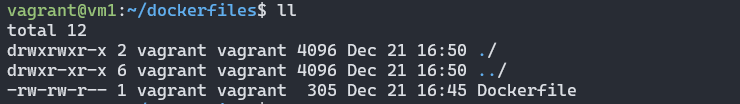
$ sudo usermod -aG docker “user\_name

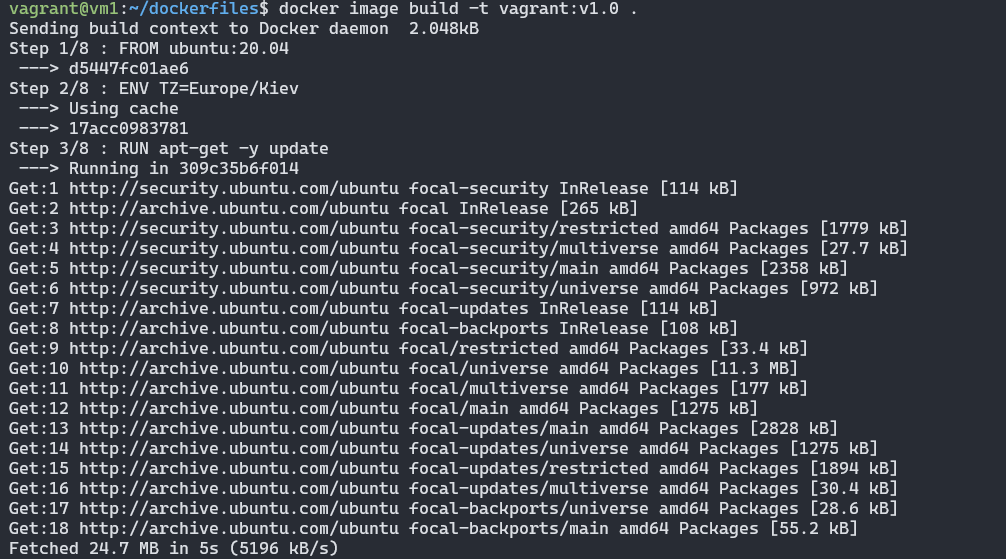
1. Create first image(based on Ubuntu 20.04).

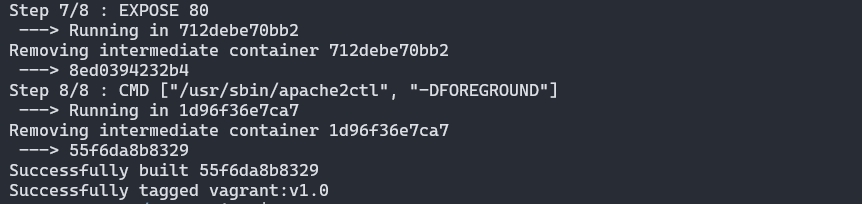
* **Create image**

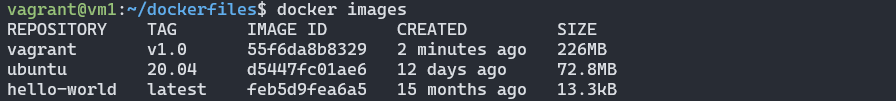




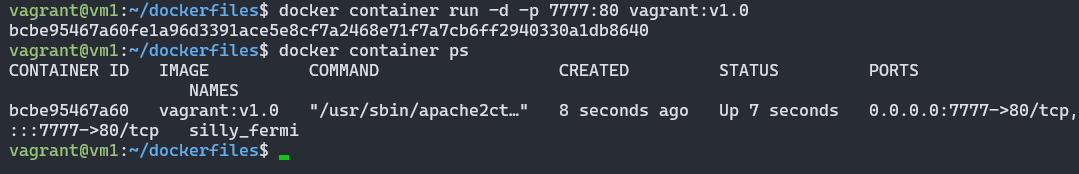


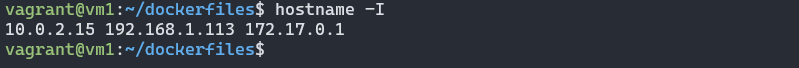


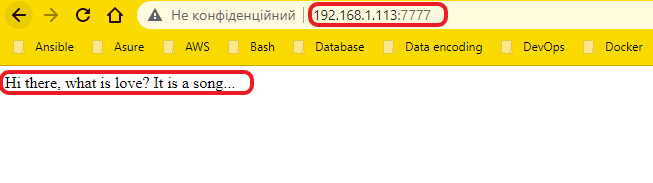


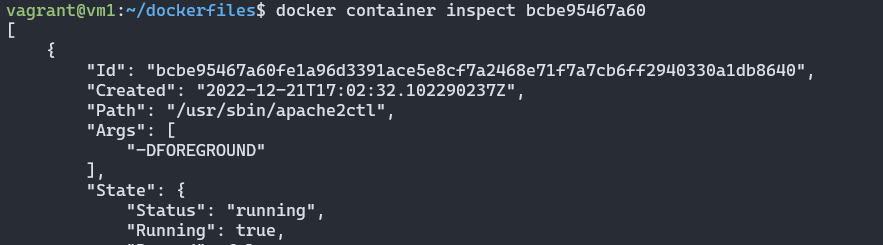


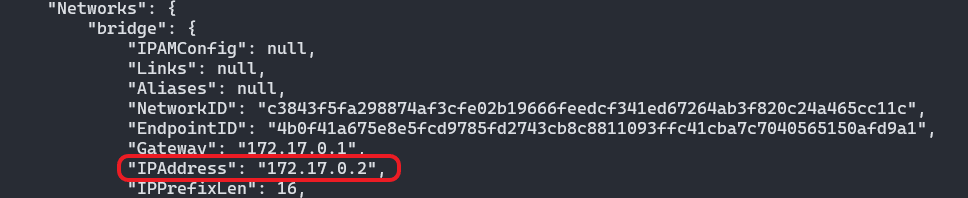
* **Run container**

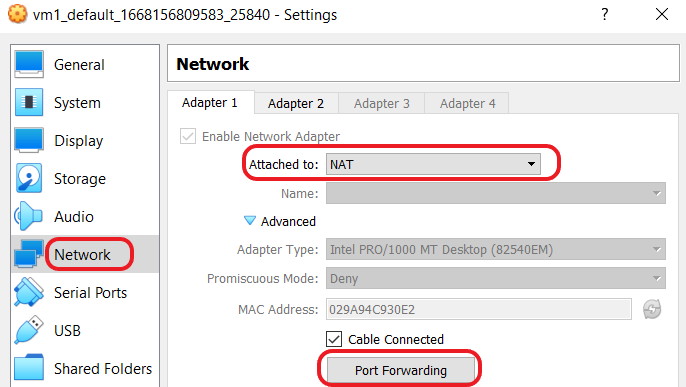


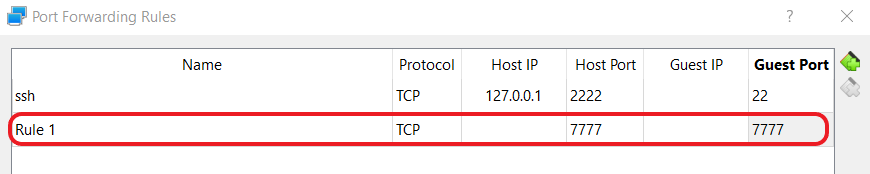


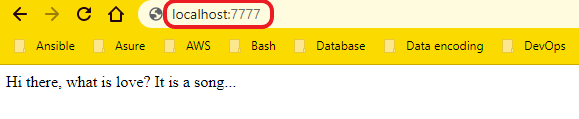
****



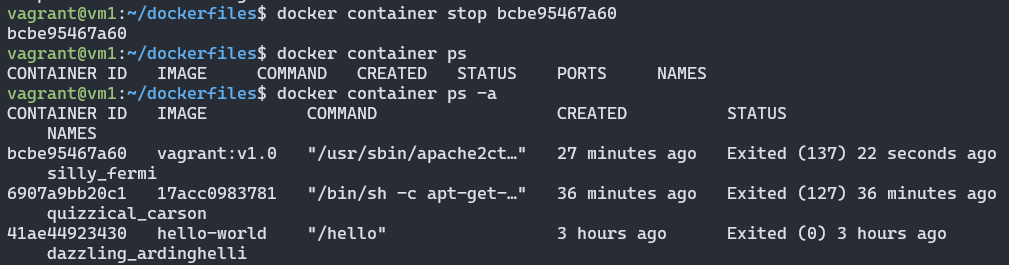


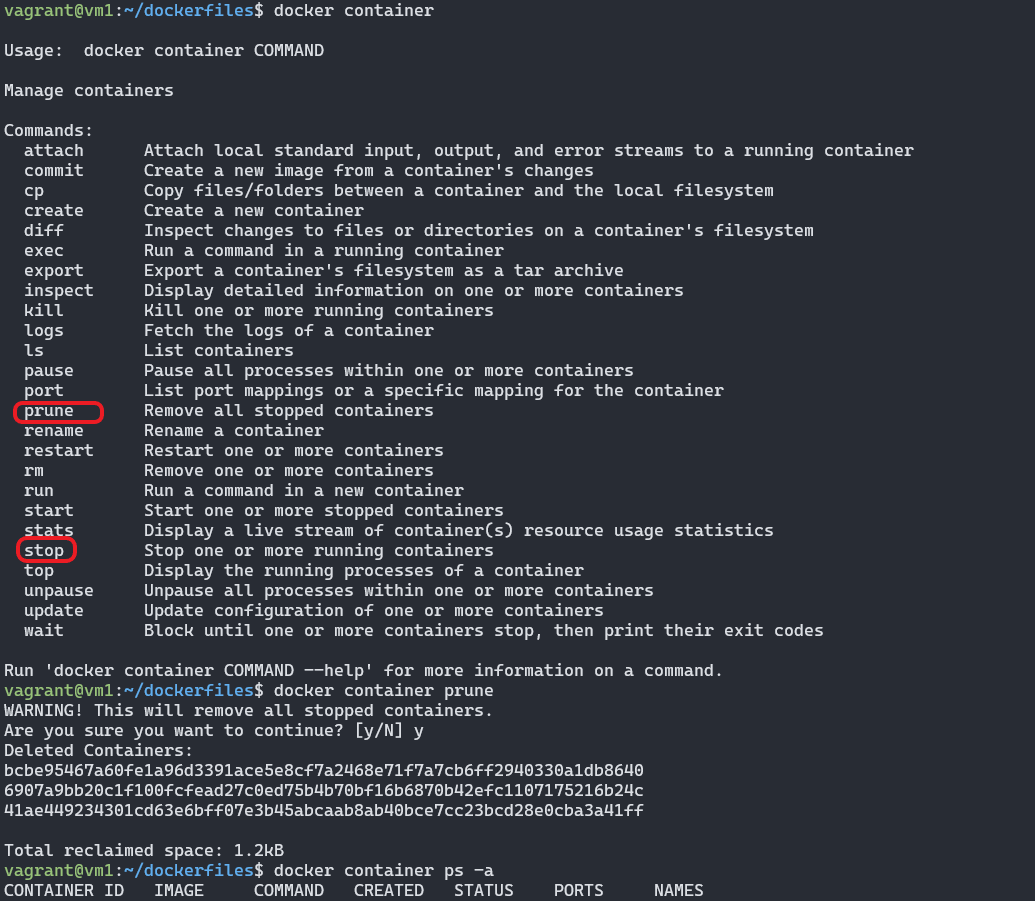






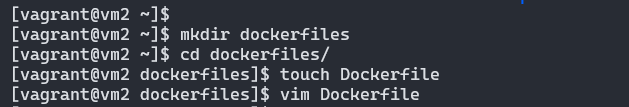
* **Stop and delete all stoped container**

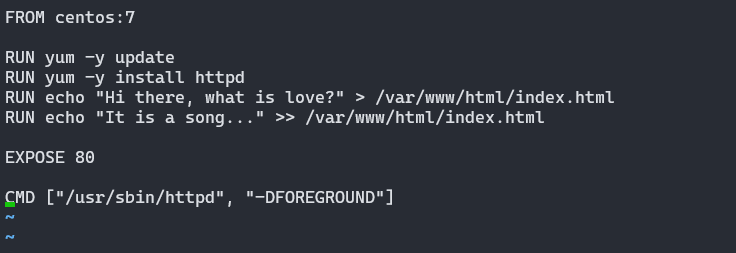


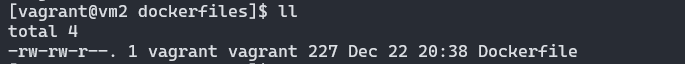


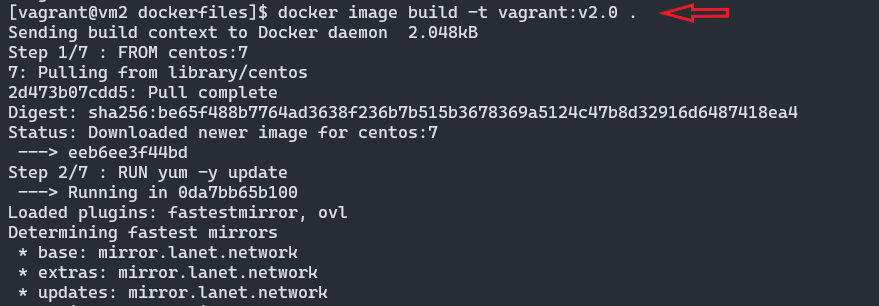
1. Create first image(based on Centos 7).

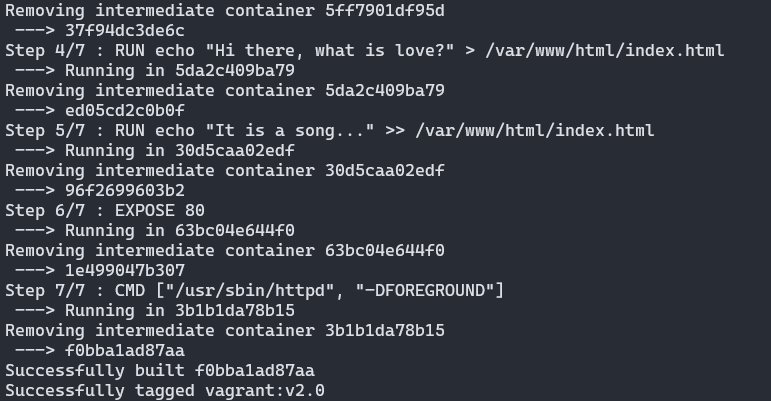
* **Create image**

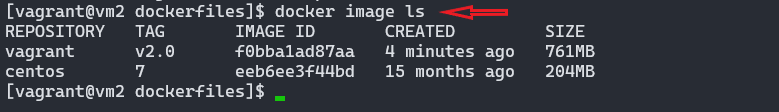




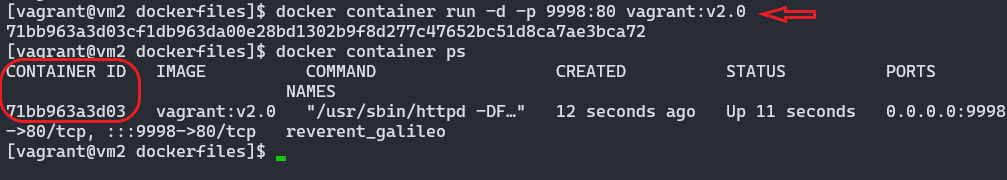


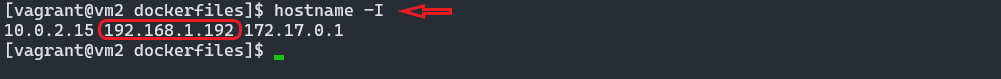


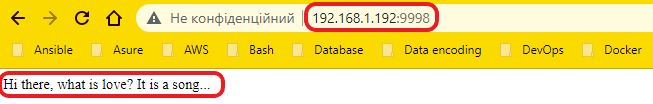




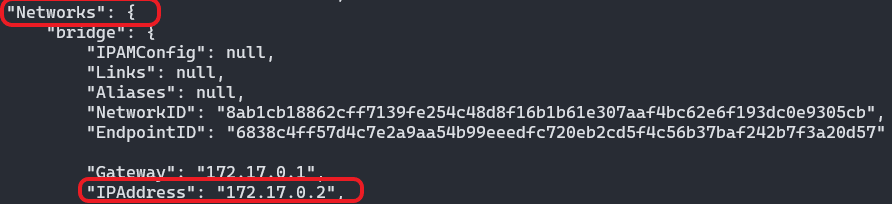
* **Run container**

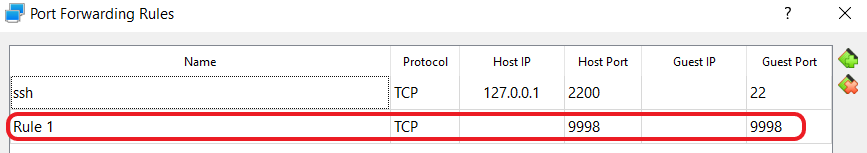
****

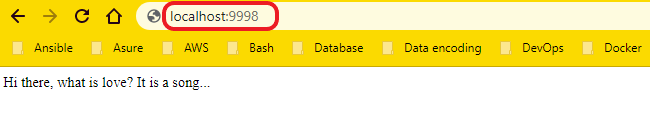
****

****

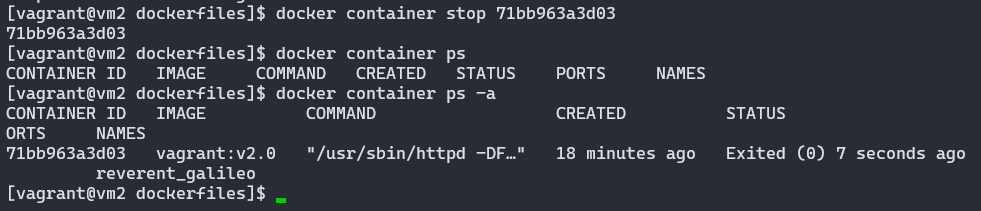
****

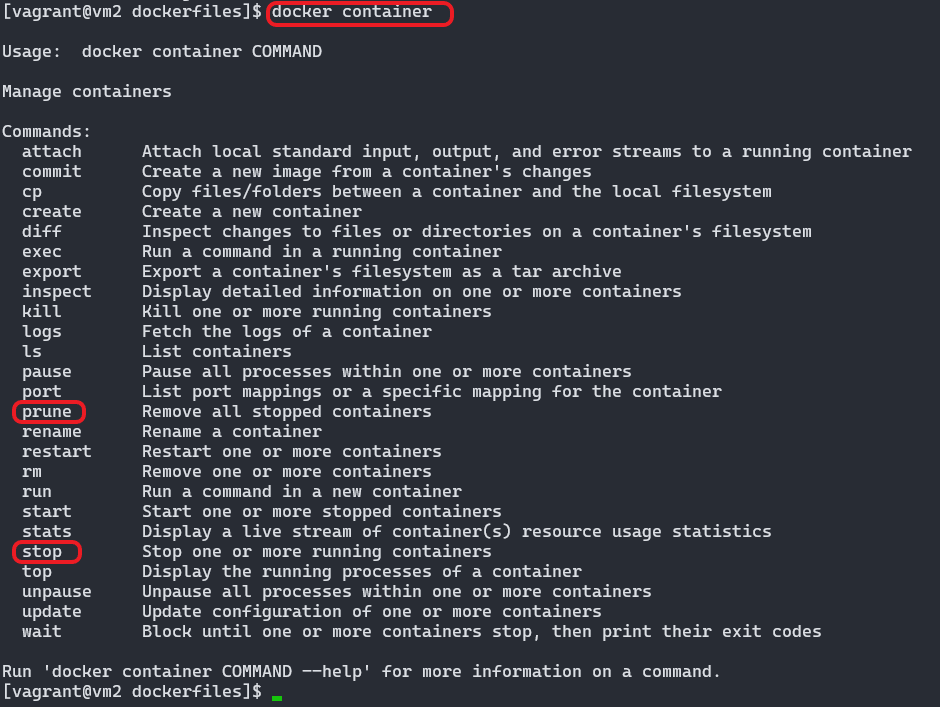
****

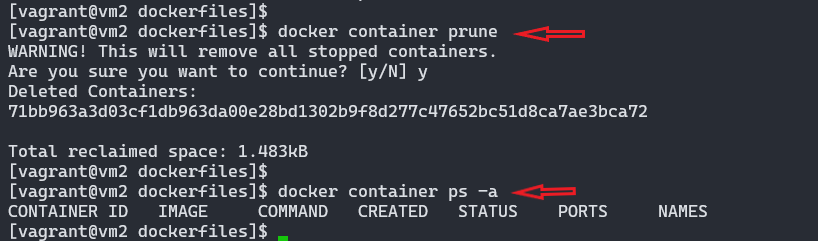
****

****

* **Stop and delete all stoped container**

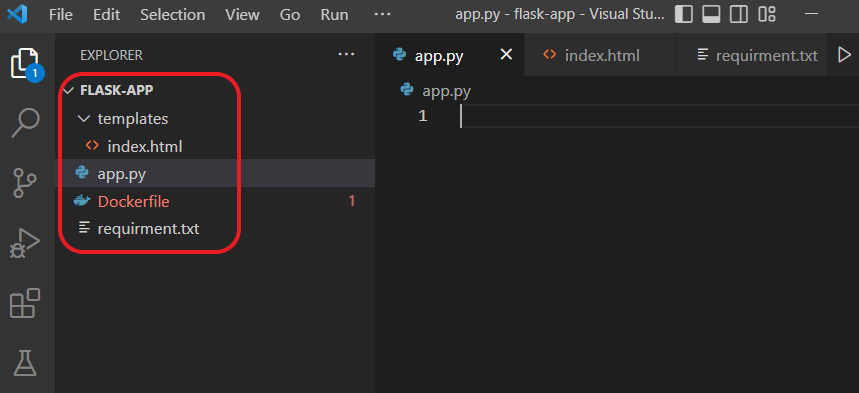




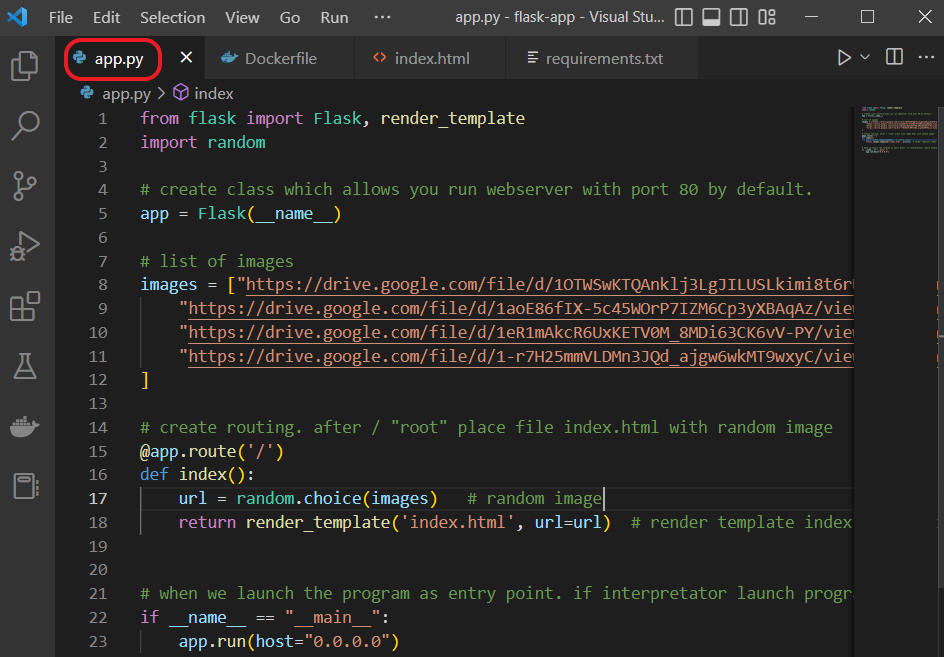


1. Create a Docker image which will run a Flask app.

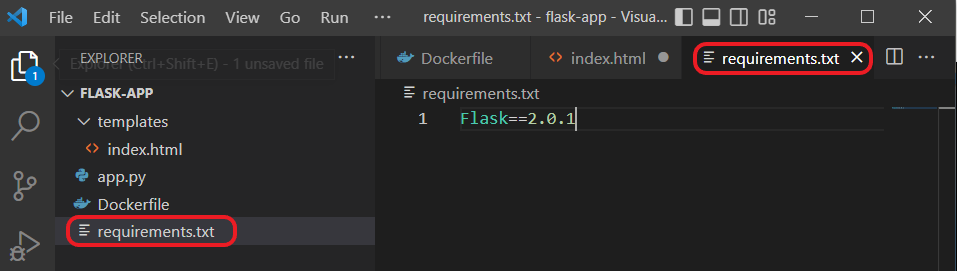
* Creating a directory called **flask-app** with following files:



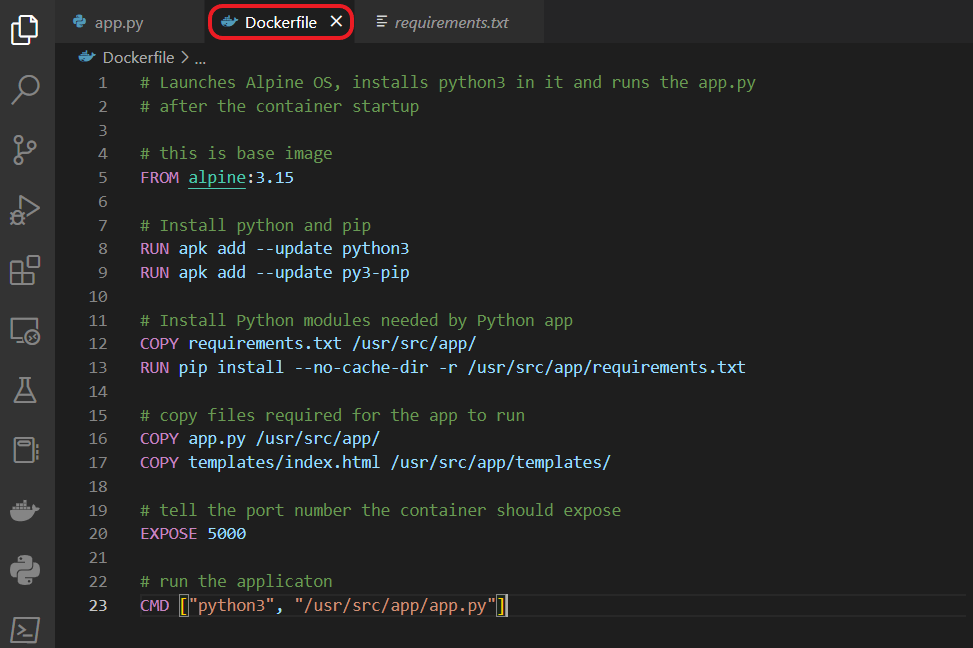
* Content of **app.py**:



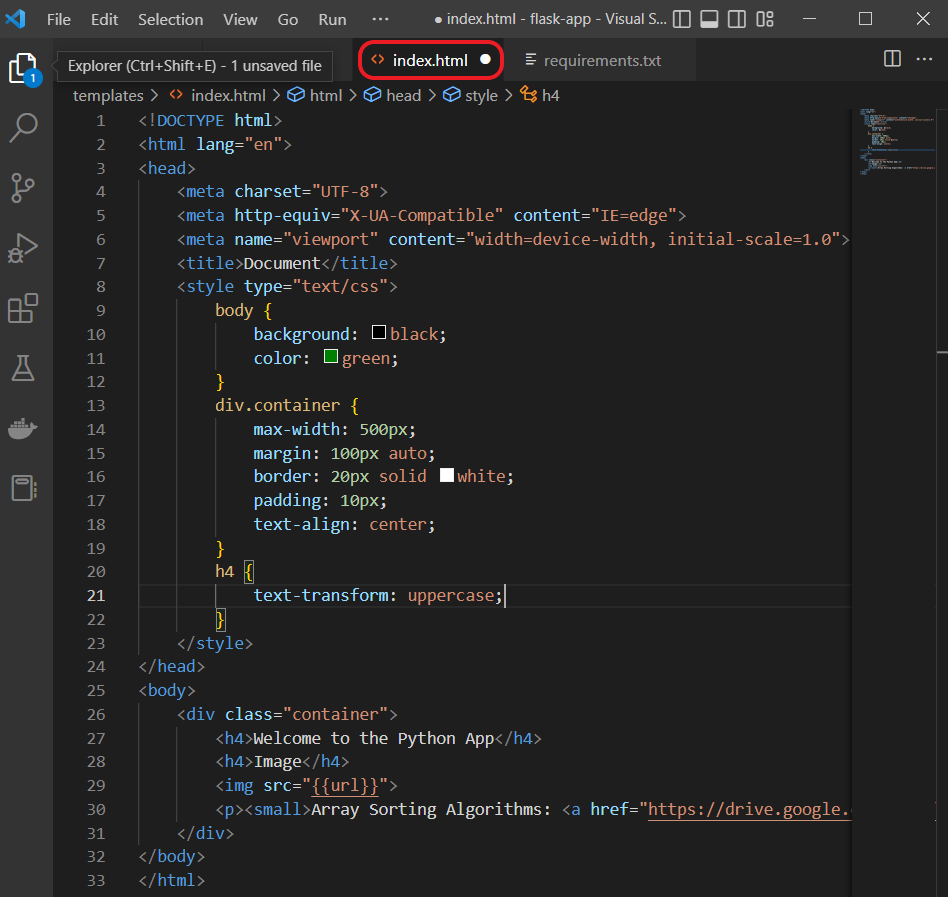
* Content of **requirements.txt**;



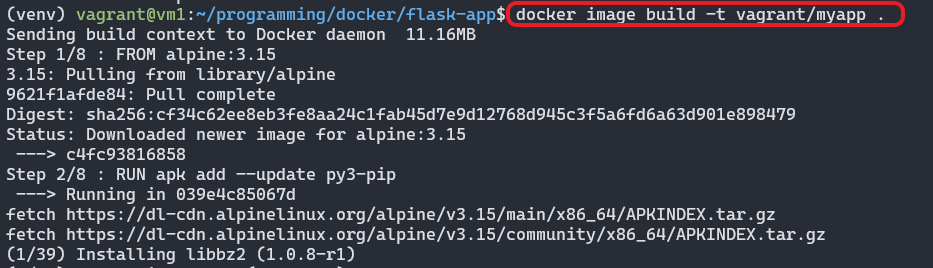
* Content of **Dockerfile**:

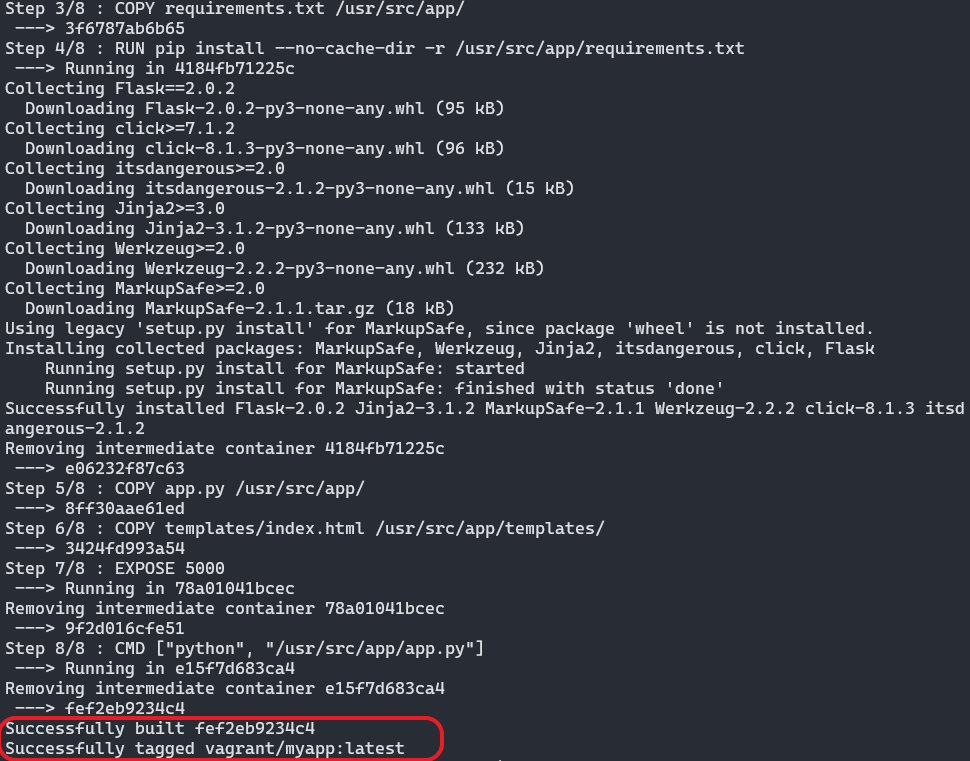


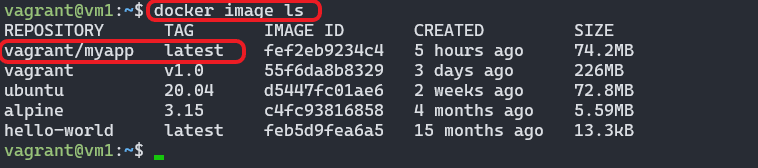
* Content of **index.html**:



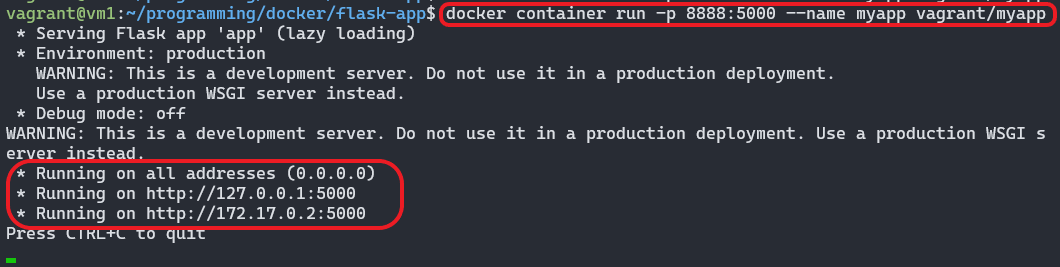
* Docker image building:

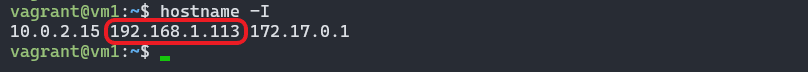


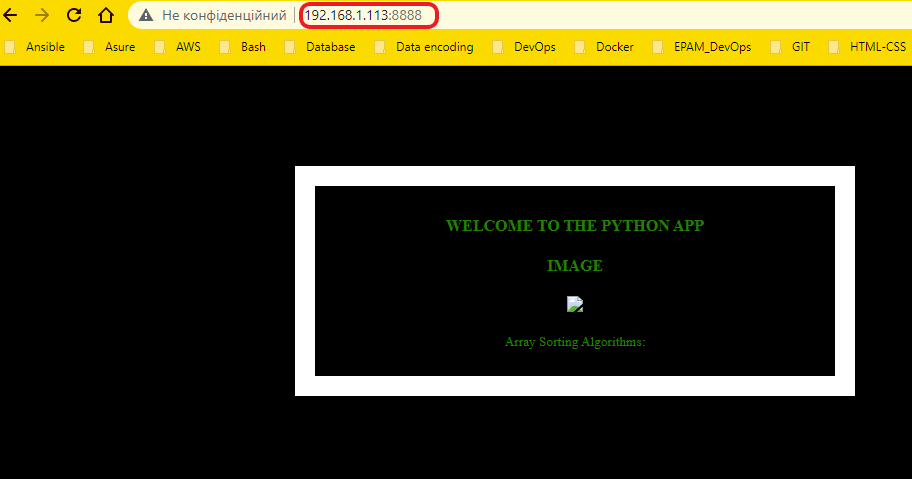




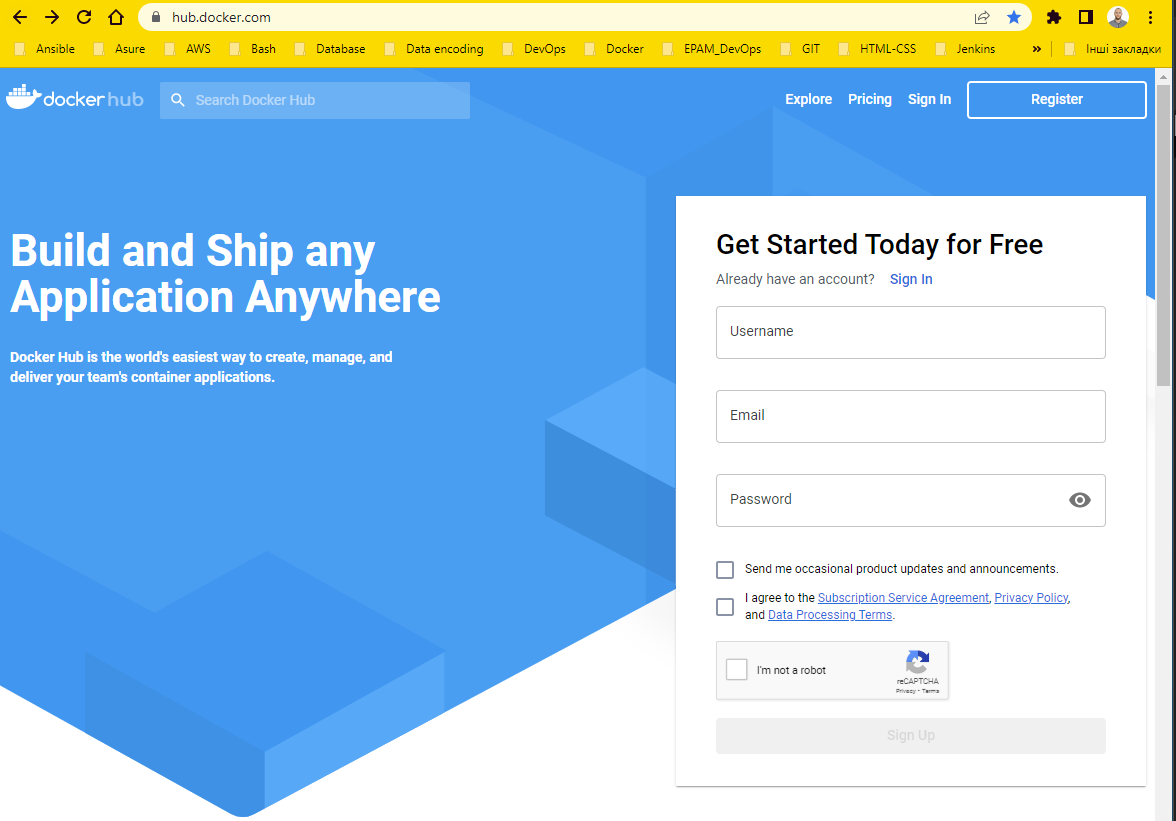
* Running container with our web application:



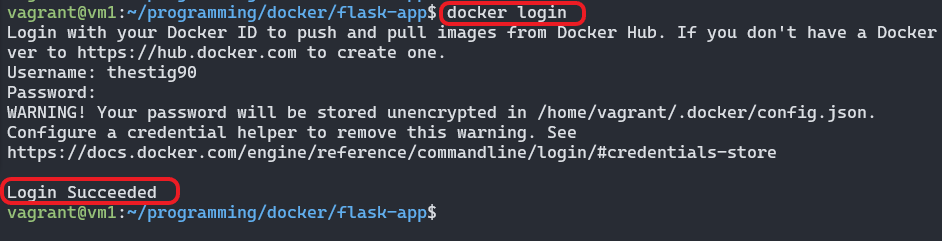




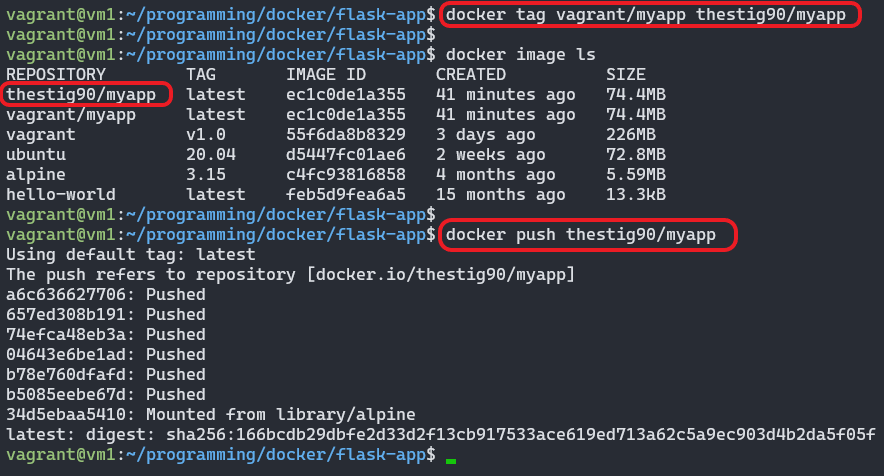
* Creating an account in DockerHub :

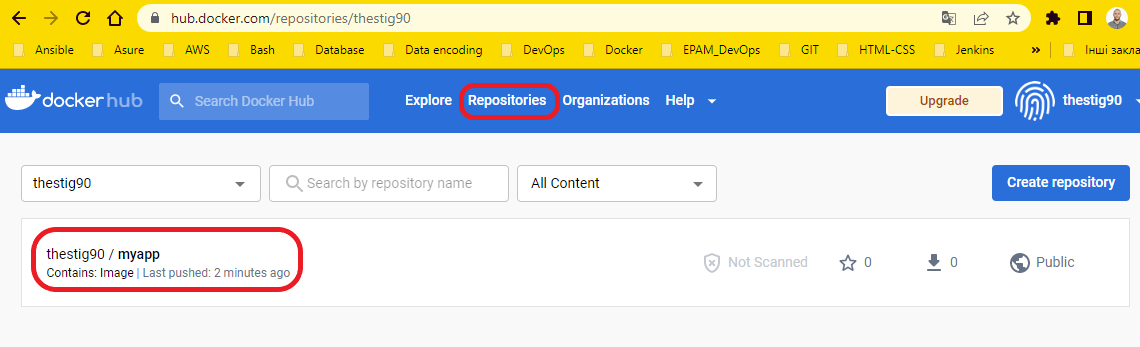


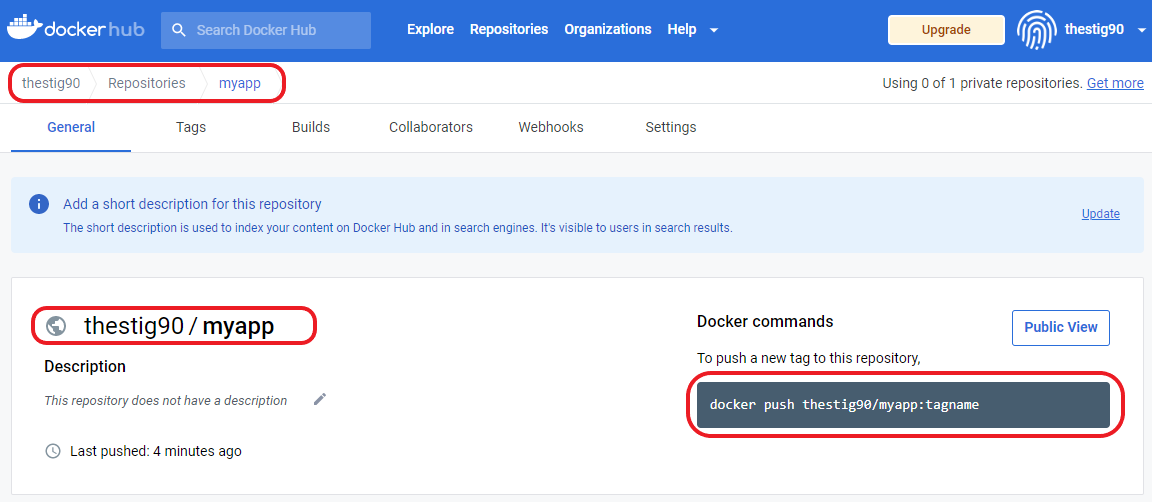
* Login to Docker registry:



* Pushing to Docker registry:



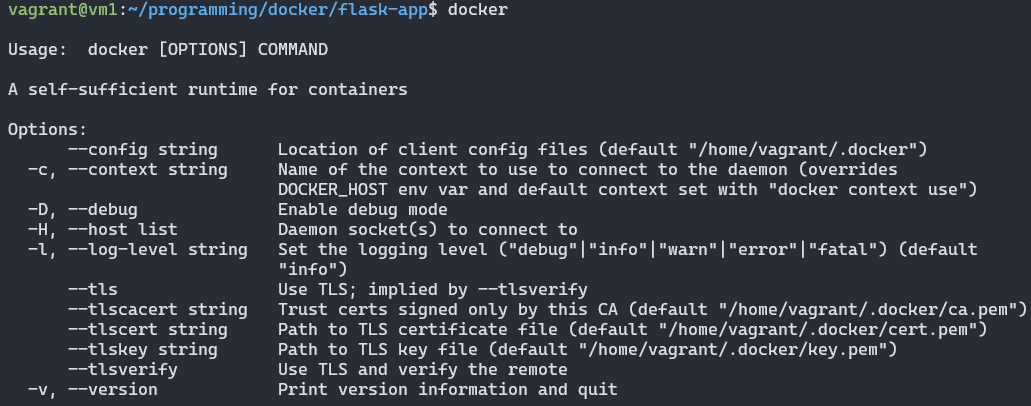


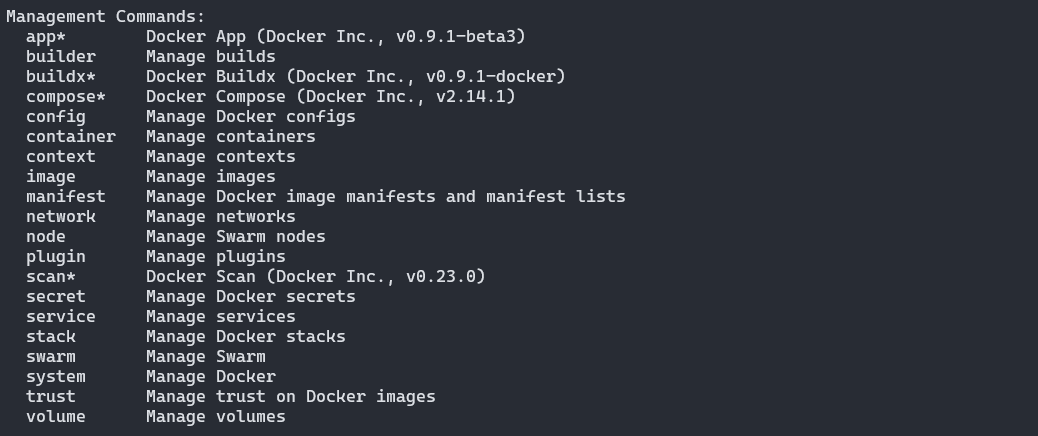


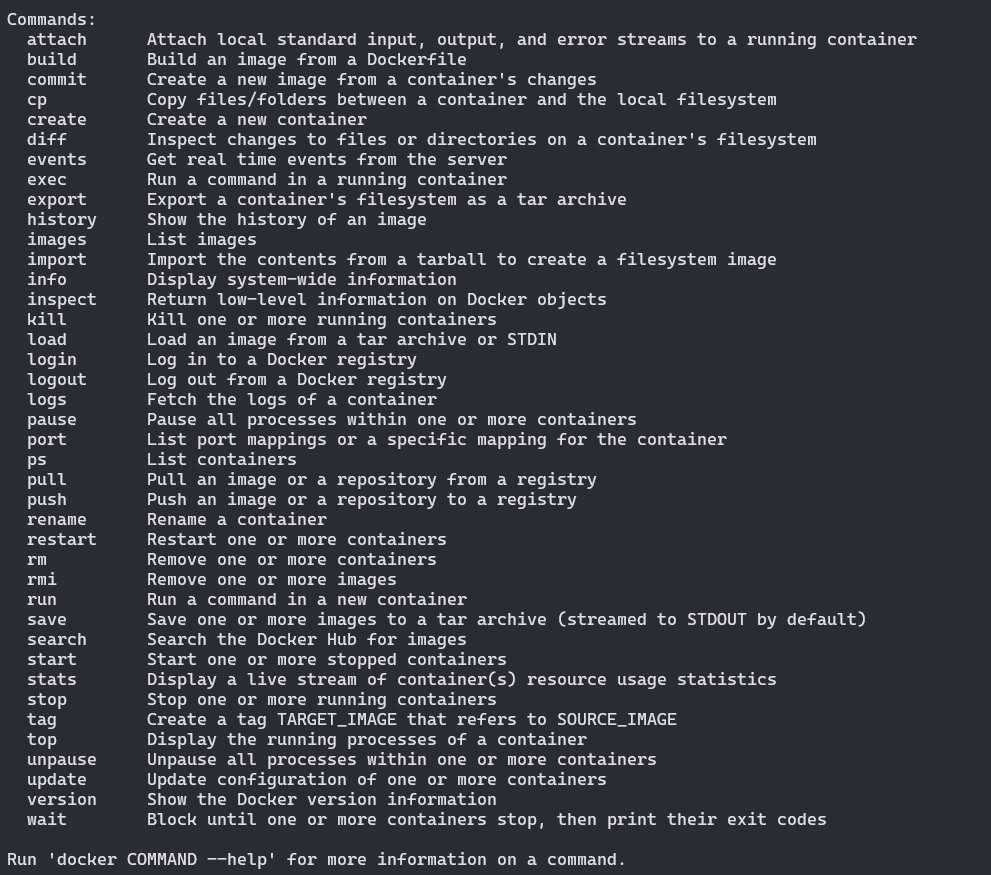
1. Frequently used Docker commands.

$ docker stop $(docker ps -a -q) #stop all containers [you need stop before delete]

$ docker rmi $(docker images-a -q) # remove all images







1. Docker Compose.

* Install **Docker Compose**

You can run Compose on macOS, Windows, and 64-bit Linux.

* Prerequisites

**Docker Compose** relies on **Docker Engine** for any meaningful work, so make sure you have **Docker Engine** installed either locally or remote, depending on your setup.

On desktop systems like **Docker Desktop** for Mac and Windows, **Docker Compose** is included as part of those desktop installs.

On Linux systems, first install the **Docker Engine** for your OS as described earlier.

To run Compose as a non-root user:

$ sudo usermod -aG docker “user\_name”

* Installation

In the latest version **Docker Compose** has been included into a Docker Engine, so you don’t need to install them separately.

To download and install Compose standalone, run:

$ curl -SL https://github.com/docker/compose/releases/download/v2.14.2/docker-compose-linux-x86\_64 -o /usr/local/bin/docker-compose

Apply executable permissions to the standalone binary in the target path for the installation:

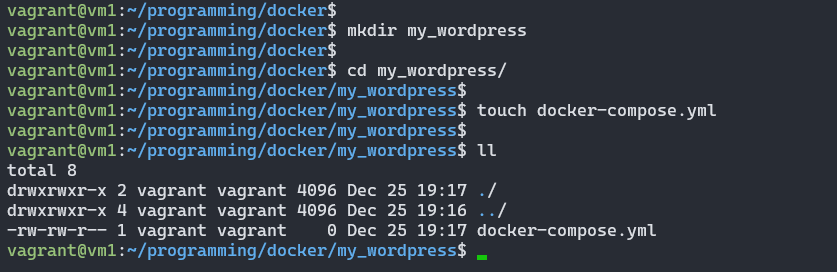
$ sudo chmod +x /usr/local/bin/docker-compose

Test and execute compose commands using:

$ docker compose

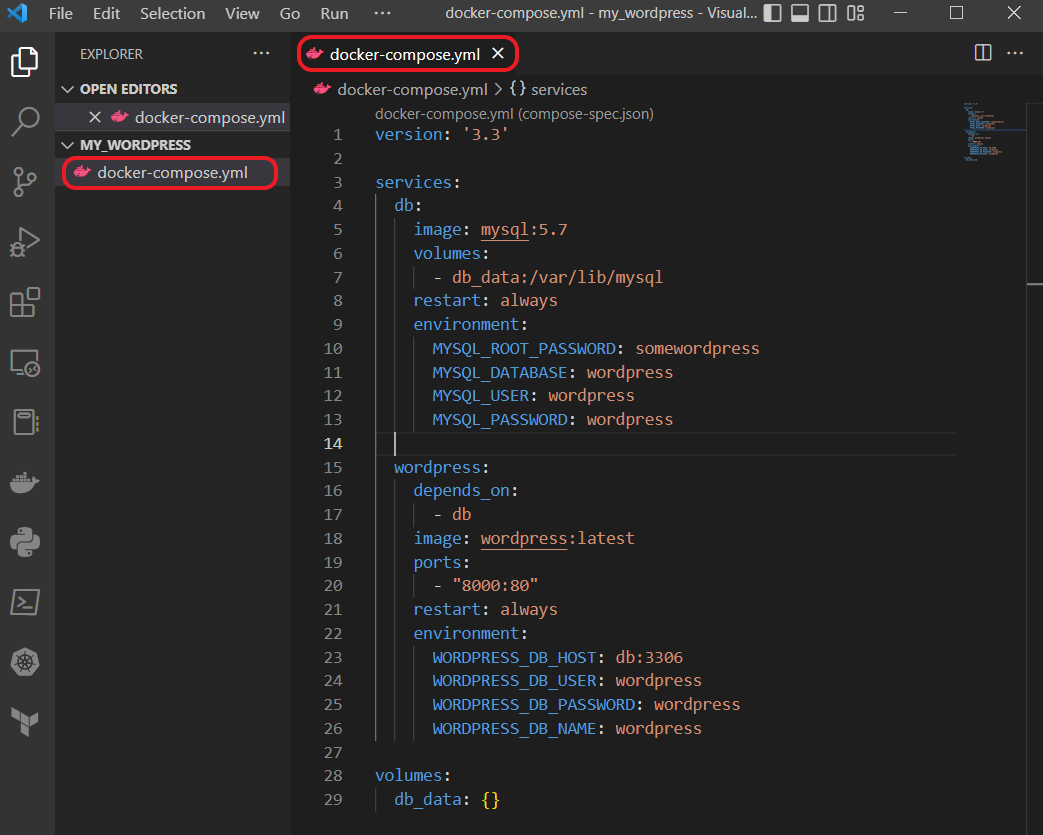
1. Docker Compose project examle.

* Create an empty project directory with **docker-compose.yml** and change into.



* Create **docker-compose.yml** file.

This **docker-compose.ym** file starts your WordPress blog and a separate MySQL instance with a volume mount for data persistence:

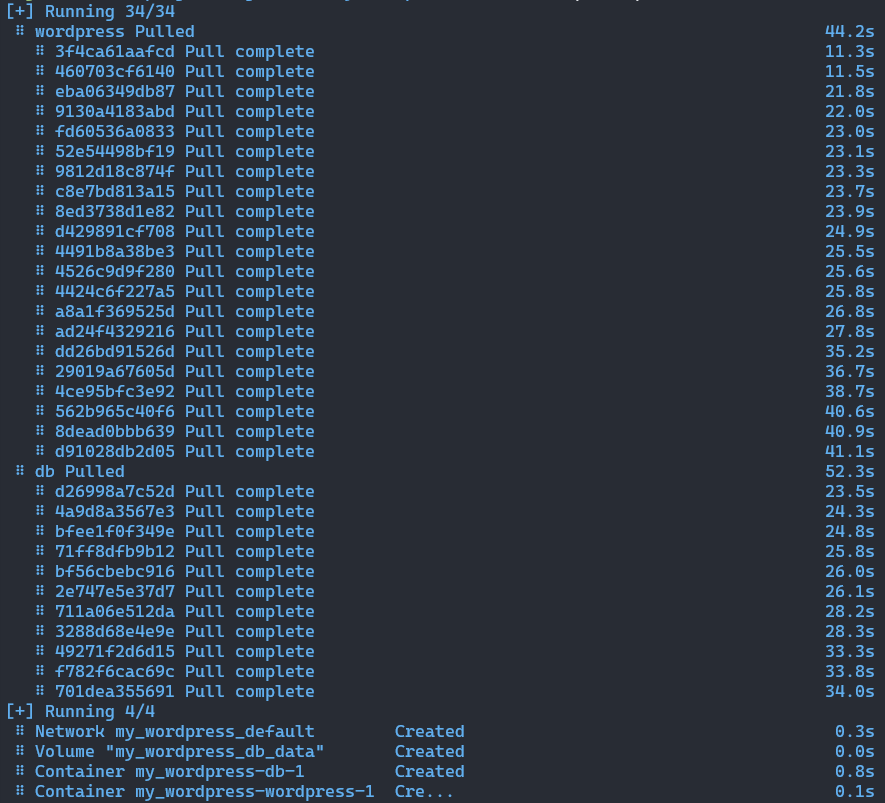


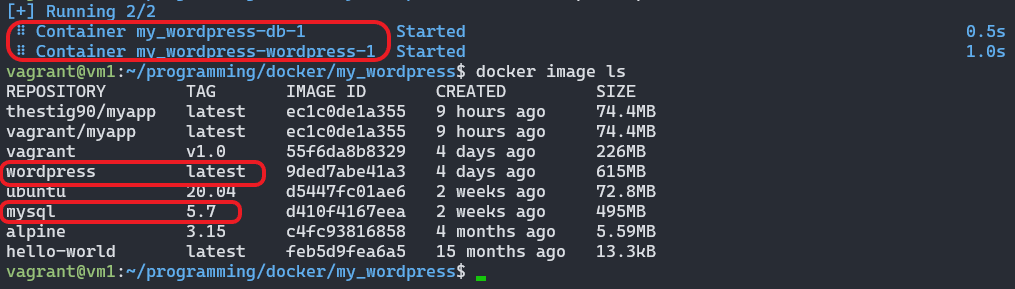
* Build the project.

$ docker compose up -d

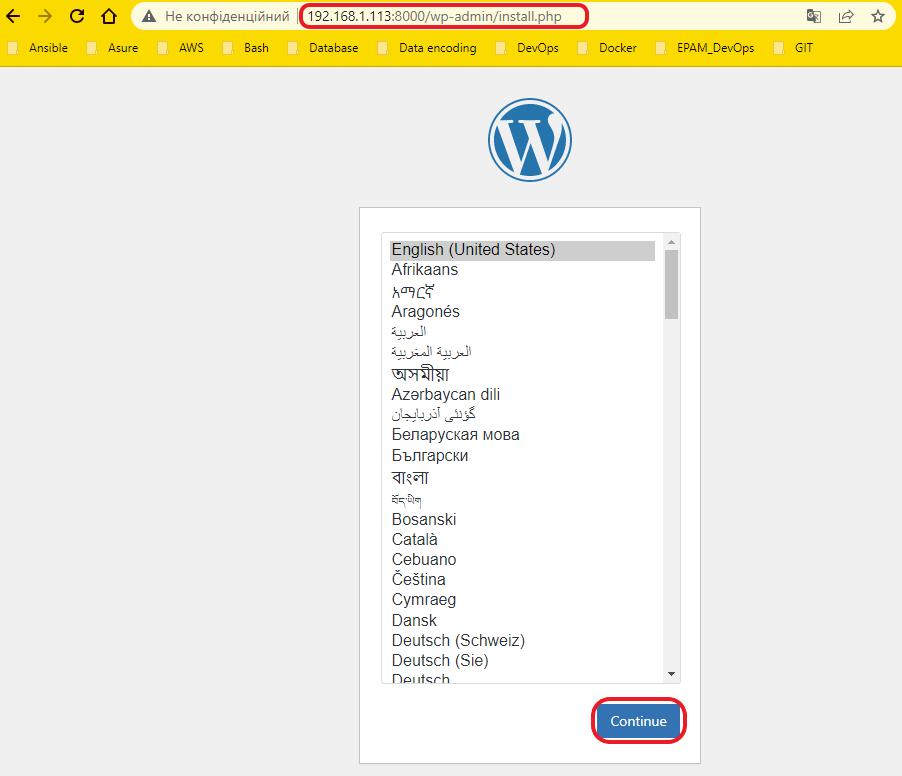
This runs **docker-compose up** in detached mode, pulls the needed **Docker images**, and starts the **wordpress** and **database containers**, as shown in the example below:

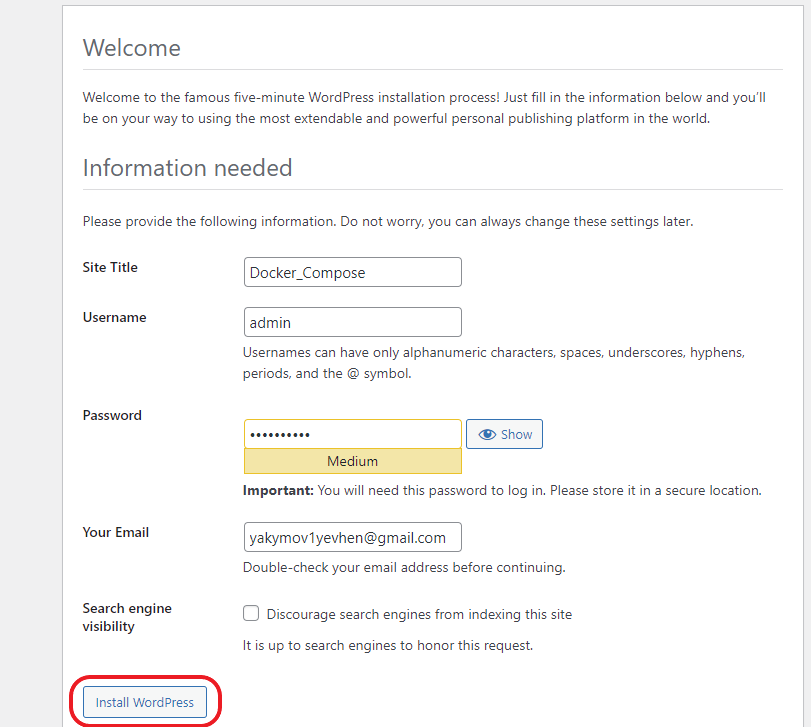


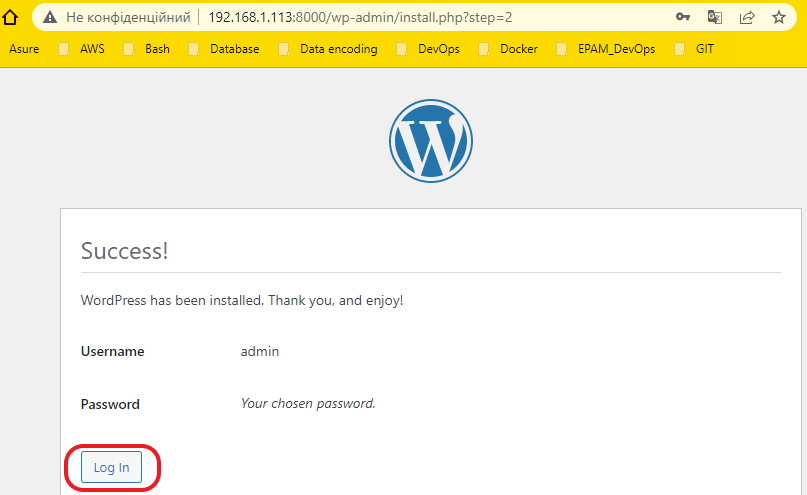


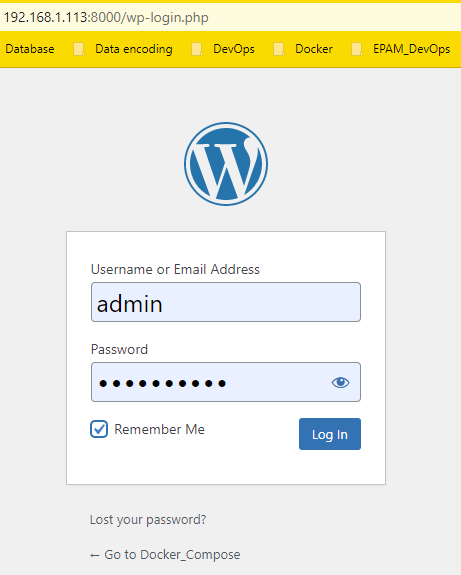


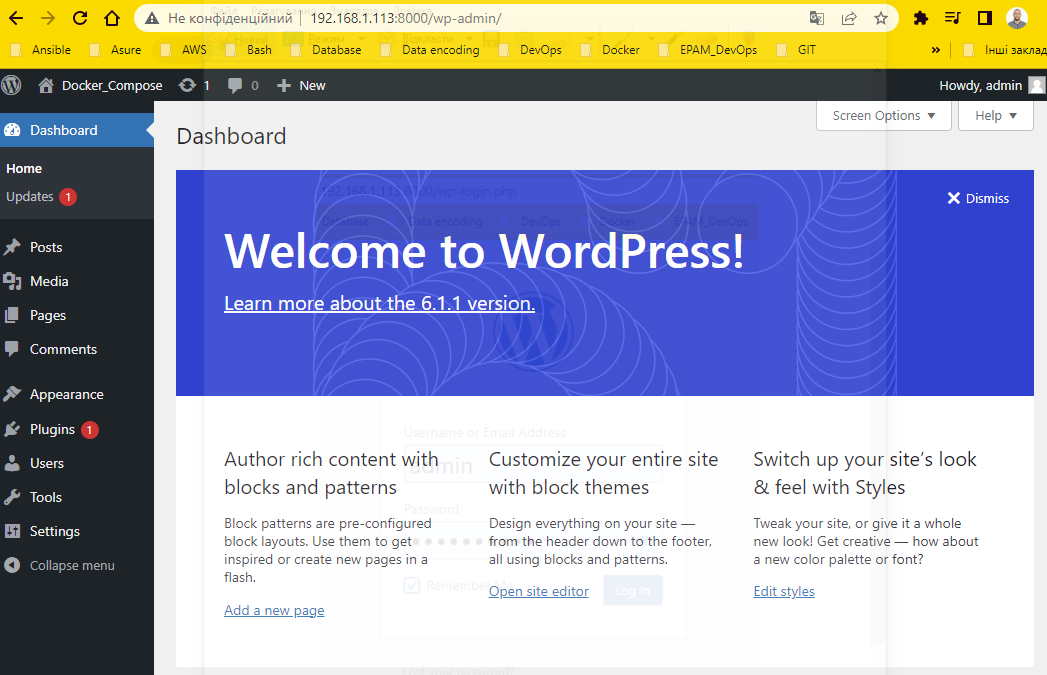
* Wordpress realization.

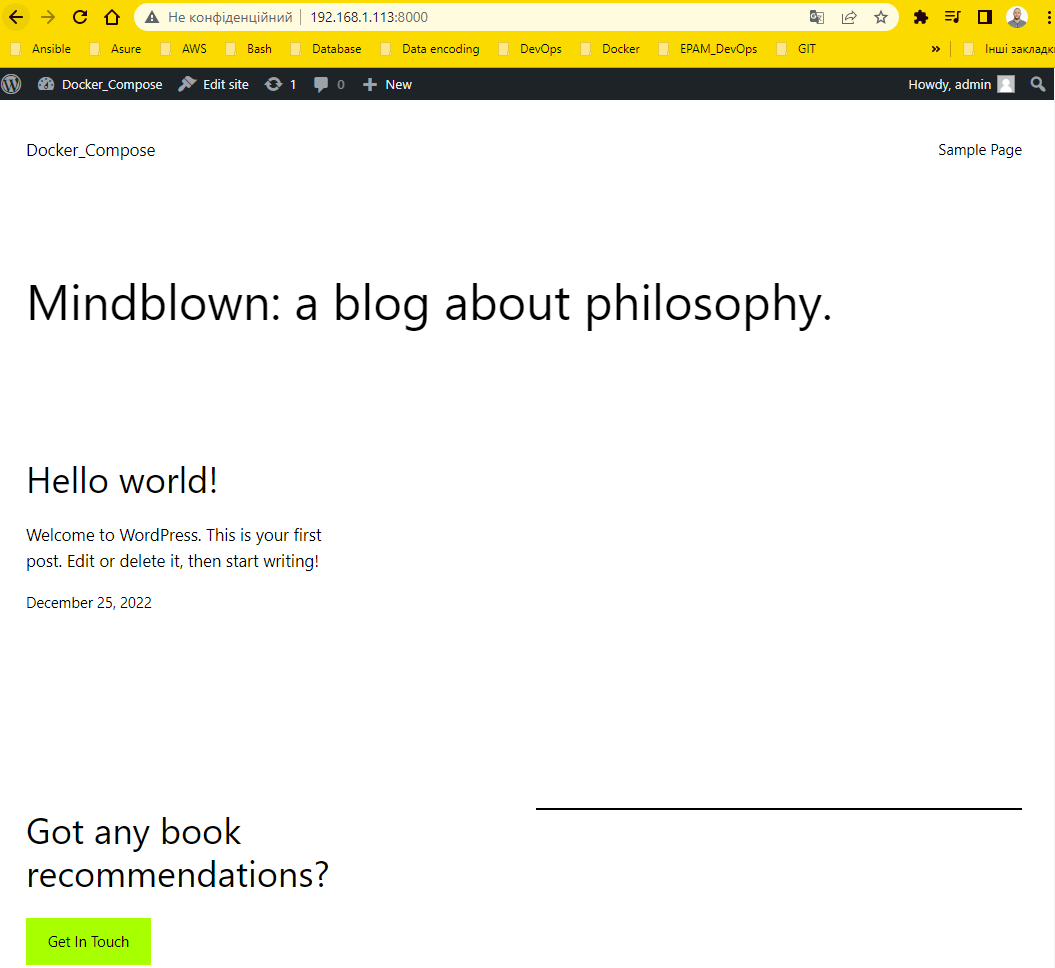












1. Docker Compose. [Use volumes](https://docs.docker.com/storage/volumes/).

* Use volumes.

**Volumes** are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

-Volumes are easier to back up or migrate than bind mounts.

- You can manage volumes using Docker CLI commands or the Docker API.

- Volumes work on both Linux and Windows containers.

- Volumes can be more safely shared among multiple containers.

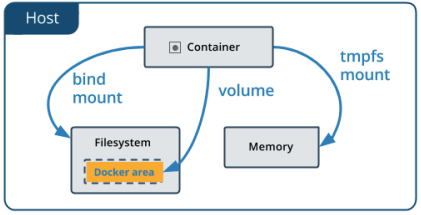
-Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.

- New volumes can have their content pre-populated by a container.

- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

-Volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.

If your container generates non-persistent state data, consider using a **tmpfs mount** to avoid storing the data anywhere permanently, and to increase the container’s performance by avoiding writing into the container’s writable layer.



* Create and manage volumes.

You can create and manage volumes outside the scope of any container.

Create a volume:

$ docker volume create my\_vol

List volumes:

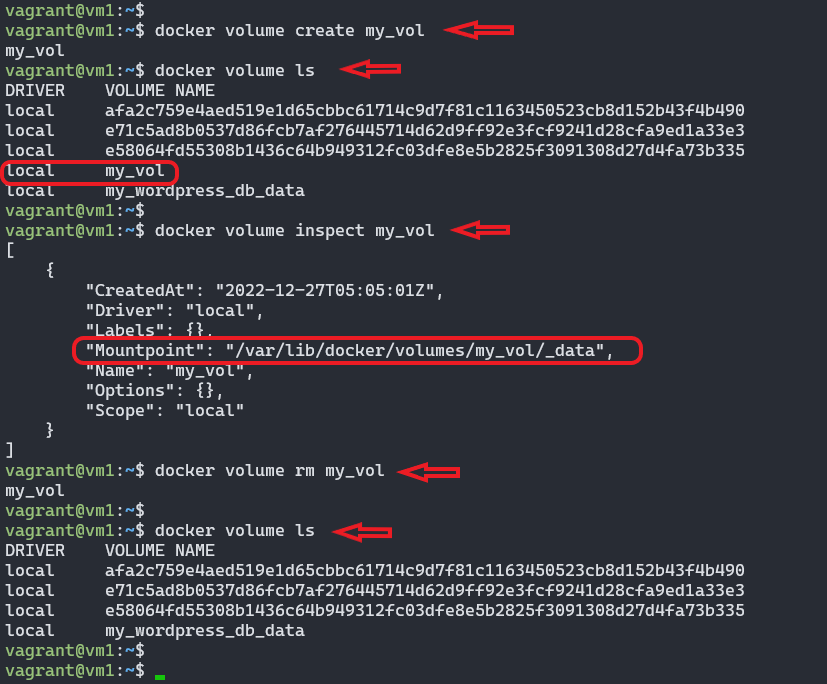
$ docker volume ls

Inspect a volume:

$ docker volume inspect my\_vol

Remove a volume:

$ docker volume rm my\_vol



1. Docker Compose. [Use networks](https://docs.docker.com/network/).

One of the reasons **Docker containers** and services are so powerful is that you can connect them together, or connect them to non-Docker workloads.

* Network drivers.

**bridge**: The default network driver. If you don’t specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate. See bridge networks.

**host**: For standalone containers, remove network isolation between the container and the Docker host, and use the host’s networking directly. See use the host network.

**overlay**: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See overlay networks.

**ipvlan**: IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration. See IPvlan networks.

**macvlan**: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host’s network stack. See Macvlan networks.

**none**: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See disable container networking.

**Network plugins**: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor’s documentation for installing and using a given network plugin.

* Create and manage bridge networks.

In terms of networking, a **bridge network** is a Link Layer device which forwards traffic between network segments. A bridge can be a hardware device or a software device running within a host machine’s kernel.

**Bridge networks** apply to containers running on the same Docker daemon host. For communication among containers running on different Docker daemon hosts, you can either manage routing at the OS level, or you can use an **overlay network**.

**User-defined bridges** provide automatic **DNS resolution** between containers:

Containers on the default **bridge network** can only access each other by **IP addresses**, unless you use the --link option, which is considered legacy. On a **user-defined bridge** network, containers can **resolve each other by name or alias**. Imagine an application with a web front-end and a database back-end. If you call your containers web and db, the web container can connect to the db container at db, no matter which Docker host the application stack is running on.

Use the **docker network create** command to create a user-defined bridge network.:

$ docker network create my\_net

List networks:

$ docker network ls

Inspect a network:

$ docker network inspect my\_net

Remove a network:

$ docker network rm my\_net

