What is Docker?

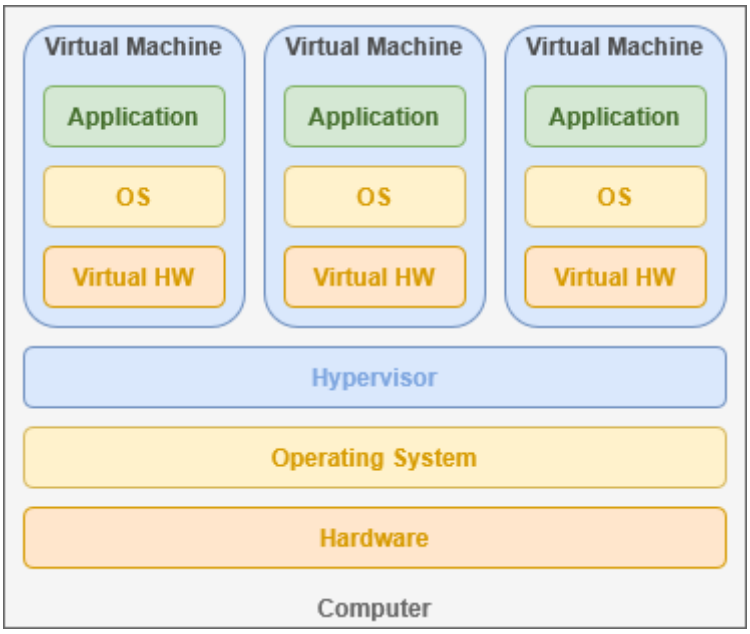
This software service runs on top of the operating system to create "virtual containers", each with their own small operating system running on top of the Linux kernel. While we can access the container's terminal or open ports to transmit data, it otherwise operates independently from the host system – sequestered into its own kernel namespace for security purposes.

> This is same technology that major companies use to host their websites.

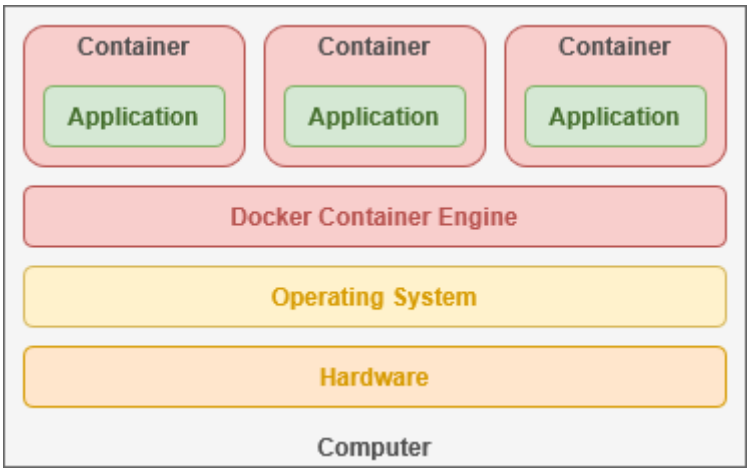
Containers allow administrators to quickly deploy software on nearly any hardware environment from a Raspberry Pi to a full blade server cluster. Docker Engine interfaces directly with the Linux kernel to access the drivers that communicate with your computer's hardware. The mechanism that virtual containers employ is fundamentally different than a virtual machine, an older technology that performs a similar function.

Virtual Systems

Virtual machines use a "hypervisor" to emulate the virtual hardware necessary to "host" its own "guest" kernel and operating system. This happens under the supervision of your "host" operating system and incurs a great deal of computational overhead. You are essentially using your computer to emulate a fully isolated system with its own hardware, firmware and software that must be juggled.

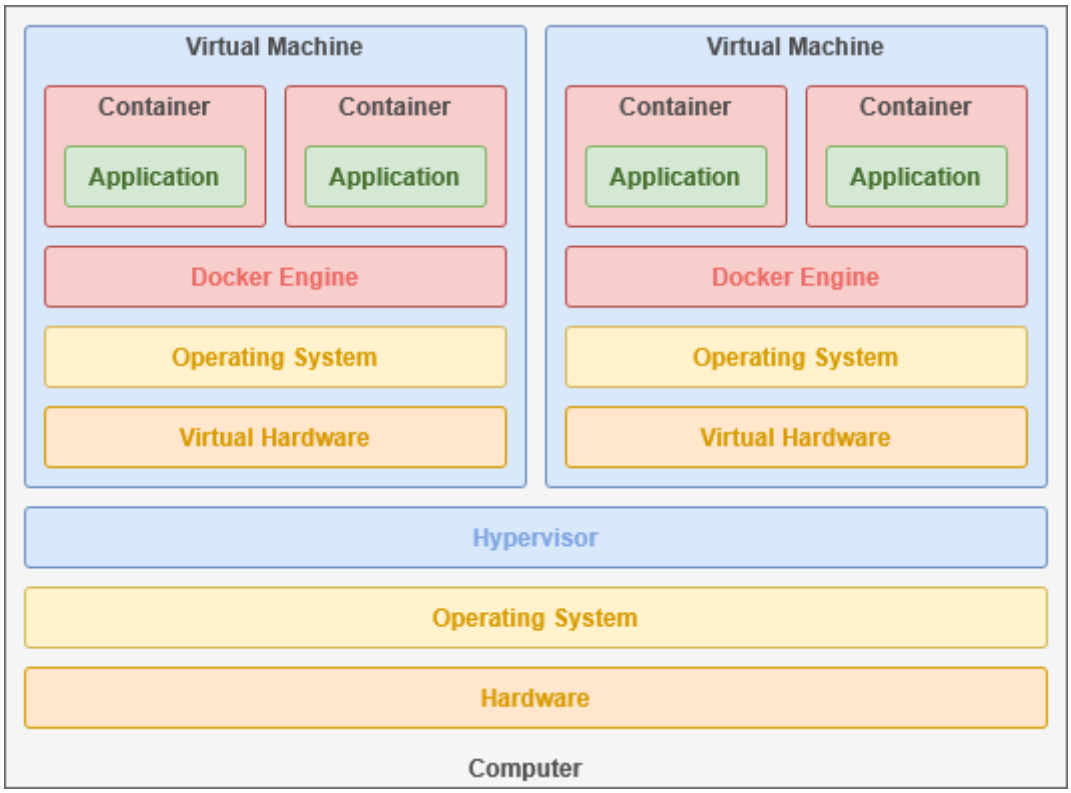


By comparison, Docker containers share their host operating system's kernel and directly utilize the existing hardware infrastructure. This allows containers to emulate the smallest possible operating system required for their software.



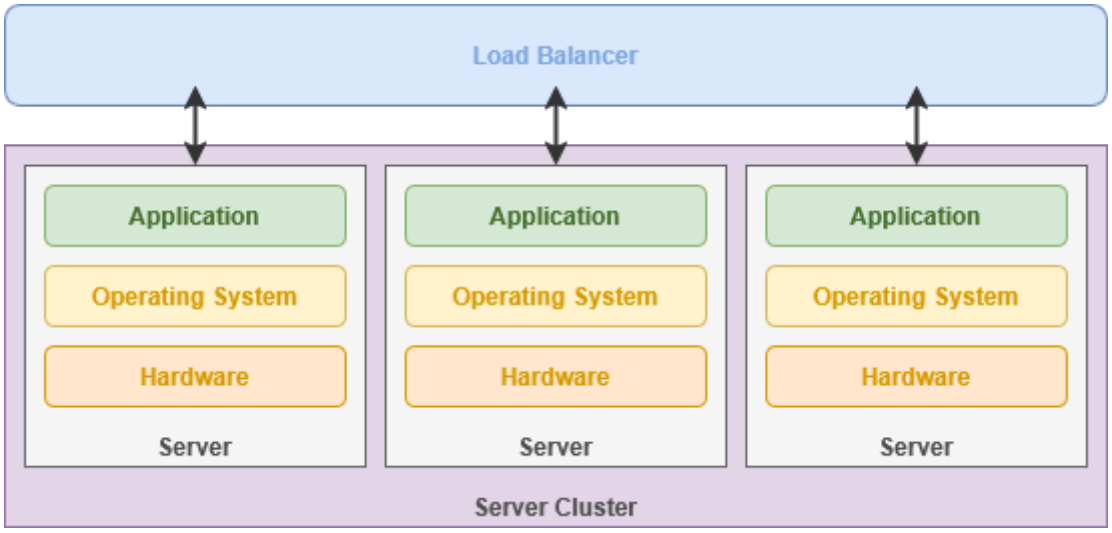
Scaling Up

Data server infrastructure often combines these two technologies to host a virtual machines for clients with each running its own private Docker instance. The exact implementation depends on a balance of efficiency, privacy and performance.



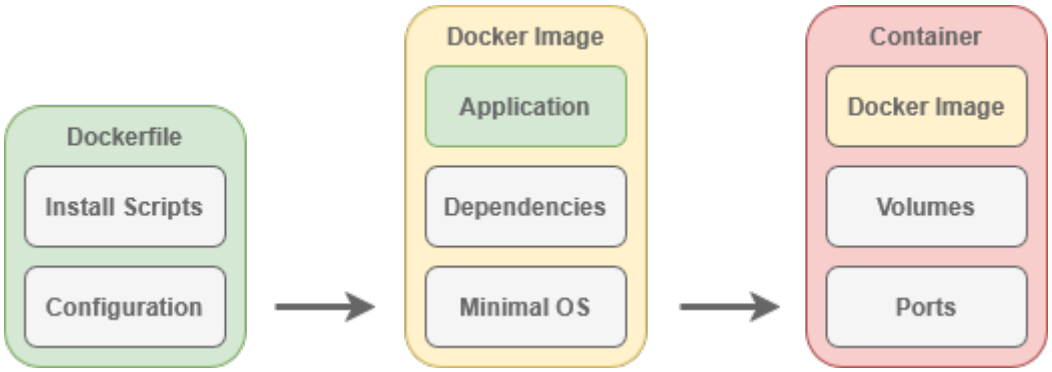
As your user base grows, Docker Swarm makes it simple to deploy across multiple servers – each running the same software – so the load can be balanced between independent systems.

Communicating over the Local Area Network, Docker can automatically create and destroy containers as needed to handle traffic.

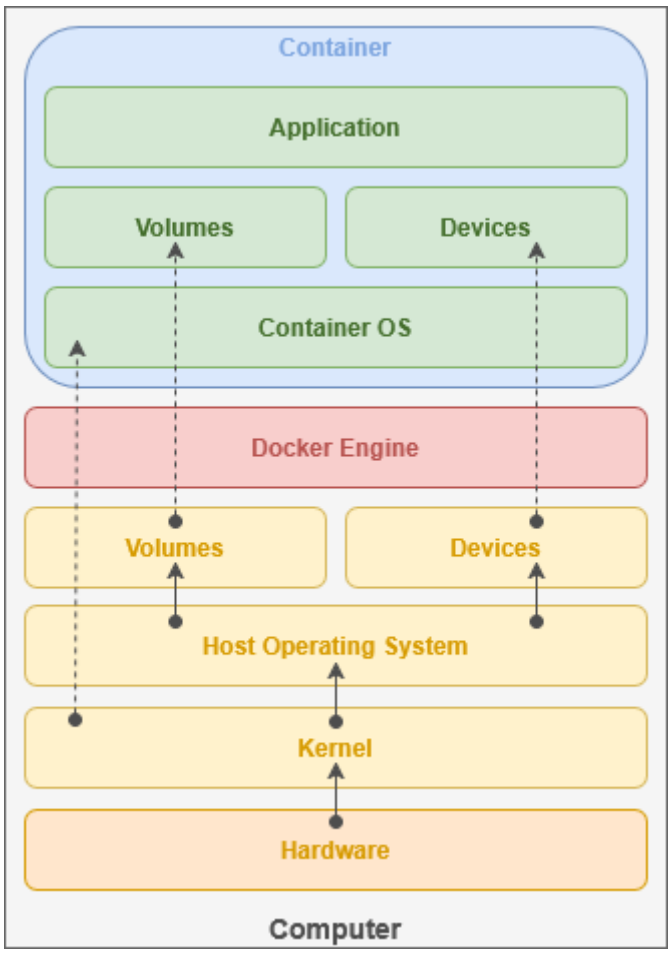


Immutable Images

Developers build a 'container image' that contains the complete operating system required for their application. Alpine Linux is the foundation of many Docker containers requiring only 5mb of storage space and 120mb of RAM.



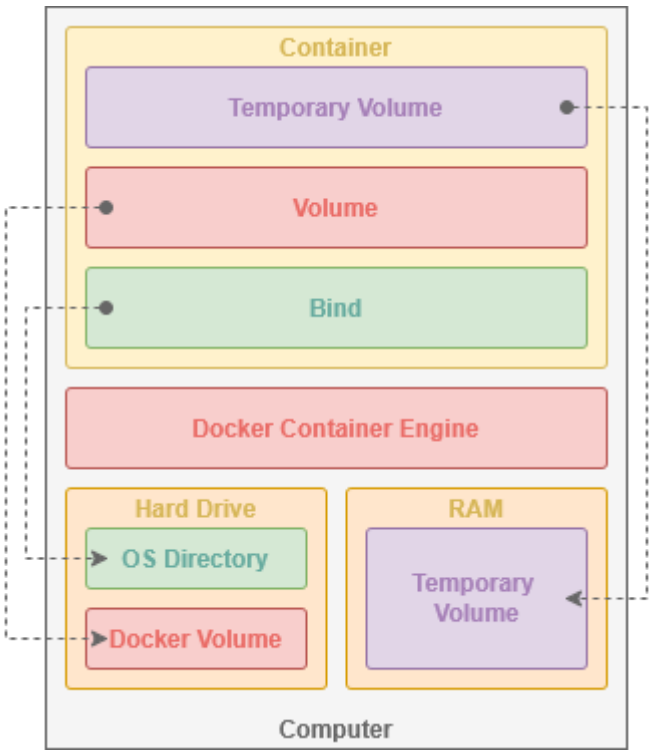
These act as a template for quickly deploying a containerized operating system that can interface directly with your hardware. Since Docker can communicate with the kernel, it has the ability to share device and file access with individual containers.



Container images are read-only and its files cannot be directly changed – a property known as "immutable". Any changes you make will only be stored in memory and will reset once the container is restarted. In order to keep data between power cycles, we need to designate persistent storage space for the container.

Persistent Storage

Docker-optimized applications will often store all of their persistent data within a single directory, commonly called "*/app*" or "*/config*". This makes services easy to update services because all you need to do is download the latest Docker image and re-start the container using it.



Docker Volumes

Docker can automatically create virtual disk drives – called "volumes" – tied to the container that can be deleted alongside it when no longer needed. This is helpful for trying out an image, creating ephemeral web instances or sharing data among multiple containers.

Host Binds

You can also mount a directory or file from the host server inside the container. These files can be read-only or fully controlled by the container. This is how many containerized applications keep track of their persistent data in a single-server solution.

Temporary Volumes

Docker can also create a temporary filesystem in active memory that is deleted when the container is stopped. These are excellent for occasions where information needs to be quickly accessible, but should never written to a hard drive for security purposes – such as private caches.

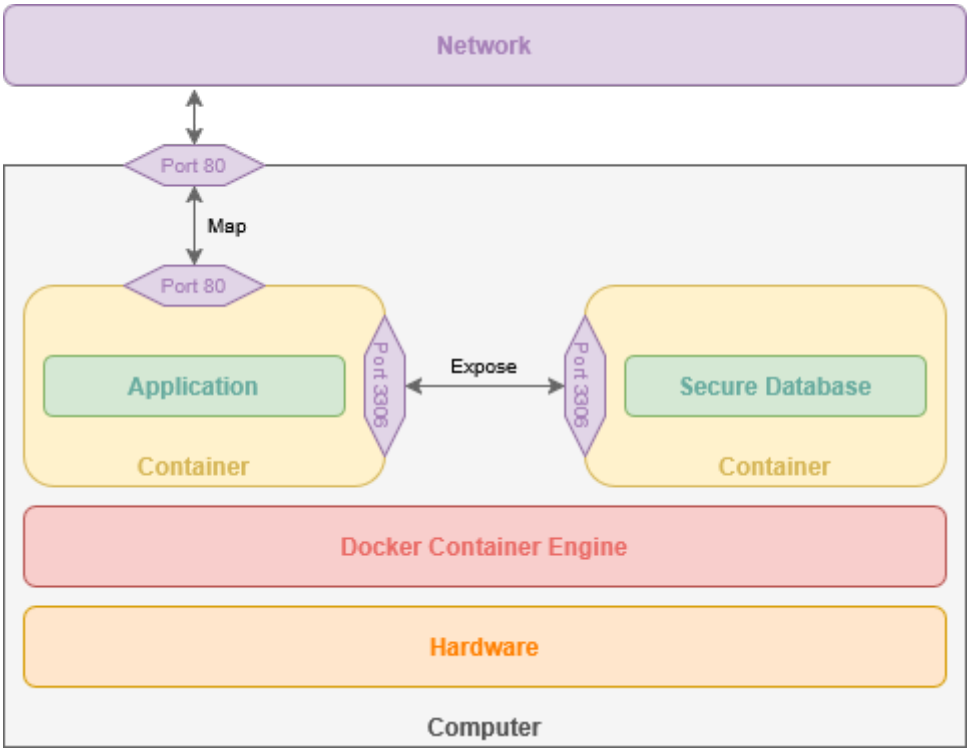
Networks & Neighbors

When creating a container, we can open access to network ports that allow communication with the service you are hosting. For many self-hosted cloud services, this includes access to the browser-based graphical user interface served over HTTP. Some services, like qBittorrent, use ports to coordinate communication with the outside internet through your router. Each service chooses what the functions or purpose for the ports it uses.

These ports can also enable communication between multiple containers – such as an application frontend and it's database. Docker can increase security by allowing your services to communicate behind-the-scenes, inaccessible to access from outside your local computer.

Modern operating systems have 65,535 ports to be individually allocated. A few are reserved system ports (such as port 80 for HTTP), but the majority are freely available for use. As a metaphor, specific telephone numbers are reserved for emergency services while others are available for use as telephone numbers to businesses and consumers.

We can either chose to "expose" a port – making it accessible only to the other containers within the stack along side it – or "map" the ports to make them accessible outside of the stack. When opening a port, we need to use both an "internal" port and an "external" port.

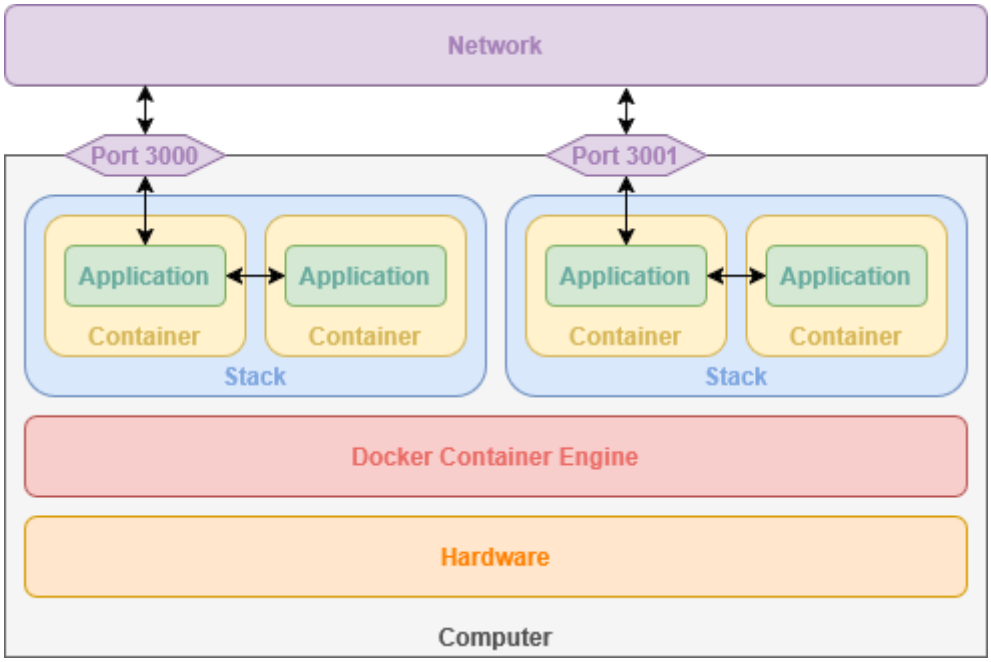


Since we are virtualizing an operating system inside each container, they have their own 65,000 available ports. By defining two numbers – such as 80:80 – we can connect the port used by the container to the port used by our host system. When a container image defaults to a common port – like port 80 – we can assign it to another available port. We use the order "*external:internal*" – or "*host:container*" – when mapping a container port.

> Mapping an nginx container – which uses port 80 – to port 4000 on your host computer would be: "*4000:80*".

Isolated Systems

By leveraging ports, we can access multiple services hosted from the same machine, each running in their own container. This is a common practice – known as a Docker Stack – that allows you to deploy new containers as well as define the virtual private networks connecting them. Conceptually, a stack sits a level higher than containers and can consist of several containers that work in tandem.



As an example, we could create two stacks for hosting two independent websites. Each stack would have an nginx container allocating sequential ports – such as 3000 and 3001. Additionally, each stack would have a MariaDB container for storing web application data. Within each stack, nginx can communicate with the MariaDB container, but is completely unaware of the other MariaDB container within the other stack.

Creating Containers

Docker containers are controlled through the terminal, allowing you to easily start, stop and restart them. Similarly, you can connect to the operating system running inside the container to perform tasks and get information.

You can run a docker container from the terminal with one command.

```
sudo docker run --it -d -p 80:80 --name nginx -v /srv/nginx/:/config scr.io/linuxserver/nginx:latest
```

This is the basic syntax for creating any Docker container. There are several important parameters that define how our container functions.

The above command follows the basic syntax:

```
sudo [[program]] [[command]] [[parameters]]
```

Running 'sudo' tells the shell to run the command as Root – or 'super user do'.

We are telling the 'docker' program to 'run' a container with the following parameters:

--it	Keeps the container's shell accessible through the terminal
-d	Runs container in the background
-p	Opens a port on the container, connecting a port from the container to an external port on our host computer. This allows the service to be accessible by other computers on your network.
--name	Name to use for the container
-v	Links a directory or file from our host computer to the container so it can access it.
scr.io/linuxserver/nginx:latest	The Docker image to use for creating the container

We can check the status of running Docker containers by entering the command:

```
sudo docker ps
```

Compose

This Docker Engine add-on makes it very easy to quickly pop-up containers using an easy-to-read syntax. Compose uses YAML, a data serialization language commonly used for human-readable software configuration files.

This example shows how YAML might be used for storing information about people:

```
people:
  sally:
    name: Sally
    age: 32
    interests:
      - "Watching movies"
      - Linux
  john:
    name: John
    age: 46
    interests:
```

```
    - Music
    - "Eating out"
```

Using the Docker Compose, we can quickly define a stack with one or more containers. This makes it simple to automatically define private networks for each stack running on the system. By default, apps within the same stack can communicate with each other but restricts external connections unless a port is configured.

```
services:
  nginx:
    image: lscr.io/linuxserver/nginx:latest
    container_name: nginx
    volumes:
      - /srv/nginx/:/config
    ports:
      - 80:80
```

This Docker Compose snippet creates the same container as our Docker Engine example above using the command line.

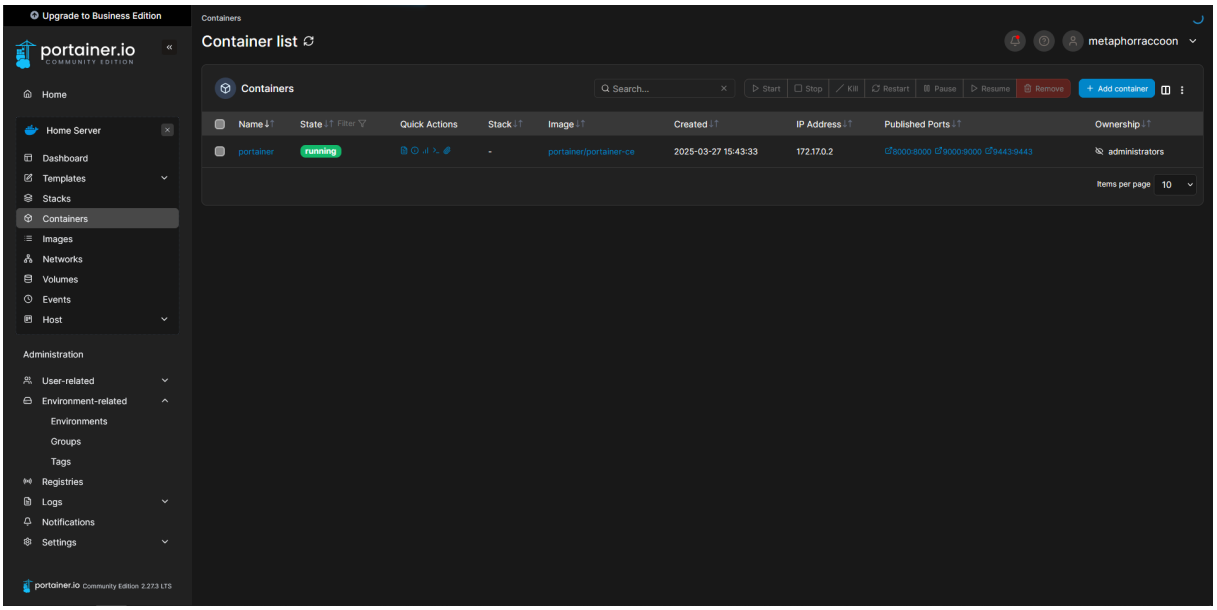
We can also create a Stack with multiple connected containers, such as WordPress which uses a web server and a database.

```
---
services:
  db:
    image: mariadb:10.6.4-focal
    command: '--default-authentication-plugin=mysql_native_password'
    volumes:
      - /srv/wordpress/db:/var/lib/mysql
    restart: always
    expose:
      - 3306

  wordpress:
    image: wordpress:latest
    ports:
      - 8080:80
    restart: always
    depends_on:
      - db
```

```
volumes:
 - /srv/wordpress/html:/var/www/html
environment:
  - WORDPRESS_DB_HOST=db
```

For convenience, we will be using Portainer which offers fully browser-based access for managing Docker containers and stacks.



Revision #59
Created 17 February 2025 09:11:56 by metaphorraccoon
Updated 11 May 2025 05:32:25 by metaphorraccoon