Publishing and exposing ports

Explanation

If you've been following the guides so far, you understand that containers provide isolated processes for each component of your application. Each component - a React frontend, a Python API, and a Postgres database - runs in its own sandbox environment, completely isolated from everything else on your host machine. This isolation is great for security and managing dependencies, but it also means you can’t access them directly. For example, you can’t access the web app in your browser.

That’s where port publishing comes in.

Publishing ports

Publishing a port provides the ability to break through a little bit of networking isolation by setting up a forwarding rule. As an example, you can indicate that requests on your host’s port `8080` should be forwarded to the container’s port `80`. Publishing ports happens during container creation using the `-p` (or `--publish`) flag with `docker run`. The syntax is:

```
$ docker run -d -p HOST_PORT:CONTAINER_PORT nginx
```

- `HOST_PORT`: The port number on your host machine where you want to receive traffic
- `CONTAINER_PORT`: The port number within the container that's listening for connections

For example, to publish the container's port `80` to host port `8080`:

```
$ docker run -d -p 8080:80 nginx
```

Now, any traffic sent to port `8080` on your host machine will be forwarded to port `80` within the container.

Important

When a port is published, it's published to all network interfaces by default. This means any traffic that reaches your machine can access the published application. Be mindful of publishing databases or any sensitive information. Learn more about published ports here.

Publishing to ephemeral ports

At times, you may want to simply publish the port but don't care which host port is used. In these cases, you can let Docker pick the port for you. To do so, simply omit the `HOST_PORT` configuration.
For example, the following command will publish the container's port `80` onto an ephemeral port on the host:

```
$ docker run -p 80 nginx
```

Once the container is running, using `docker ps` will show you the port that was chosen:

```
docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED
STATUS          PORTS                    NAMES
a527355c9c53   nginx         "/docker-entrypoint.…"   4 seconds ago
Up 3 seconds    0.0.0.0:54772->80/tcp    romantic_williamson
```

In this example, the app is exposed on the host at port `54772`.

Publishing all ports

When creating a container image, the `EXPOSE` instruction is used to indicate the packaged application will use the specified port. These ports aren't published by default.
With the `-P` or `--publish-all` flag, you can automatically publish all exposed ports to ephemeral ports. This is quite useful when you're trying to avoid port conflicts in development or testing environments.

For example, the following command will publish all of the exposed ports configured by the image:

```
$ docker run -P nginx
```

Try it out

In this hands-on guide, you'll learn how to publish container ports using both the CLI and Docker Compose for deploying a web application.

Use the Docker CLI

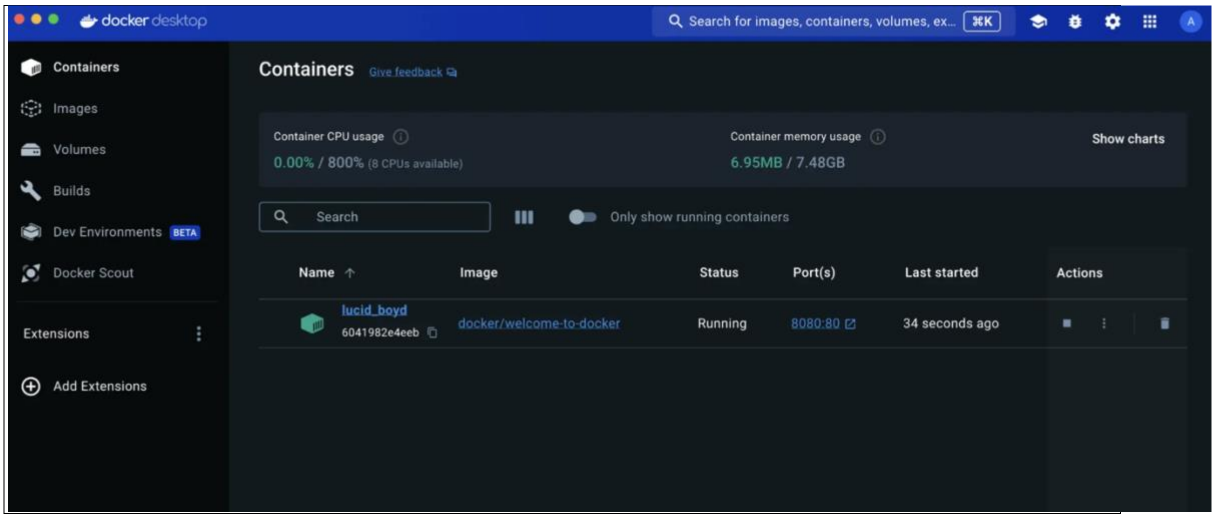
In this step, you will run a container and publish its port using the Docker CLI.

1. Download and install Docker Desktop.
2. In a terminal, run the following command to start a new container:

3.
   ```
   $ docker run -d -p 8080:80 docker/welcome-to-docker
   ```

   The first `8080` refers to the host port. This is the port on your local machine that will be used to access the application running inside the container. The second `80` refers to the container port. This is the port that the application inside the container listens on for incoming connections. Hence, the command binds to port `8080` of the host to port `80` on the container system.

4. Verify the published port by going to the **Containers** view of the Docker Desktop Dashboard.



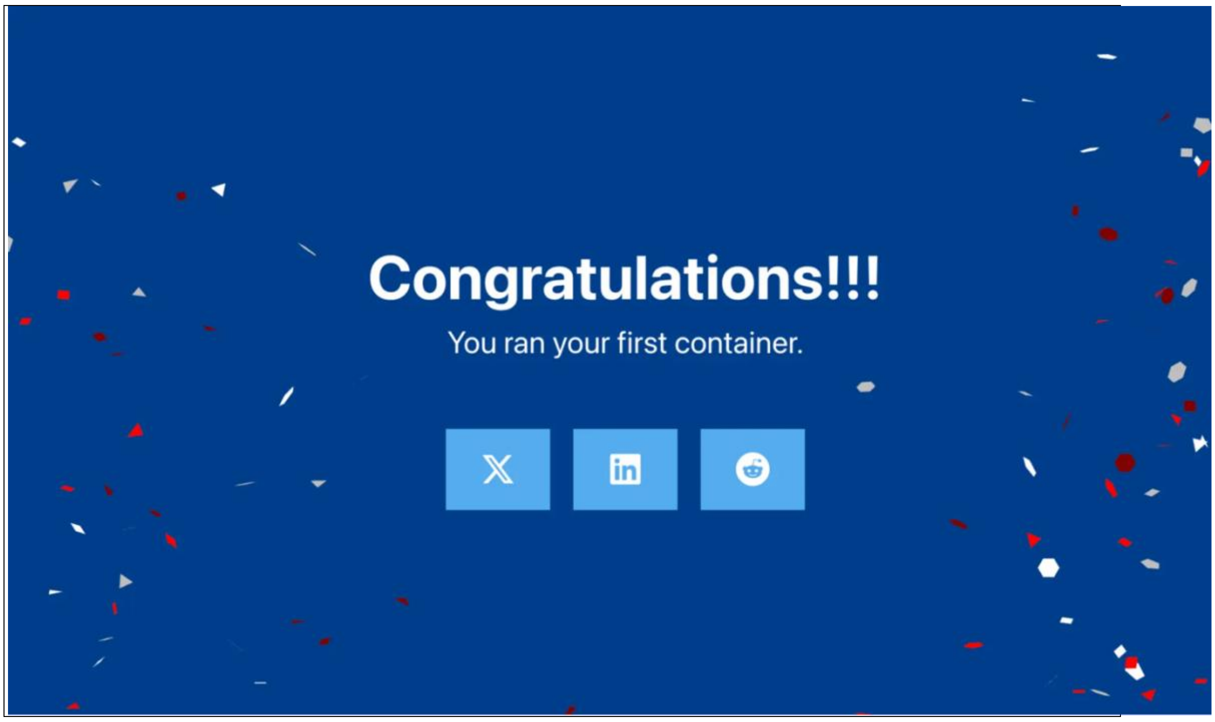
Containers

Container CPU usage 0.00% / 800% (8 CPUs available)

Container memory usage 6.95MB / 7.48GB

Name	Image	Status	Port(s)	Last started	Actions
lucid_boyd 6041982e4eeb	docker/welcome-to-docker	Running	8080:80	34 seconds ago	

5. Open the website by either selecting the link in the **Port(s)** column of your container or visiting http://localhost:8080 in your browser.



Congratulations!!!

You ran your first container.

Use Docker Compose

This example will launch the same application using Docker Compose:

1. Create a new directory and inside that directory, create a `compose.yaml` file with the following contents:

```yaml
services:
  app:
    image: docker/welcome-to-docker
    ports:
      - 8080:80
```

   The `ports` configuration accepts a few different forms of syntax for the port definition. In this case, you're using the same `HOST_PORT:CONTAINER_PORT` used in the `docker run` command.

6. Open a terminal and navigate to the directory you created in the previous step.
7. Use the `docker compose up` command to start the application.
8. Open your browser to http://localhost:8080.

Additional resources

If you'd like to dive in deeper on this topic, be sure to check out the following resources:

- `docker container port` CLI reference
- Published ports

Next steps

Now that you understand how to publish and expose ports, you're ready to learn how to override the container defaults using the `docker run` command.

Overriding container defaults

Explanation

When a Docker container starts, it executes an application or command. The container gets this executable (script or file) from its image’s configuration. Containers come with default settings that usually work well, but you can change them if needed. These adjustments help the container's program run exactly how you want it to.

For example, if you have an existing database container that listens on the standard port and you want to run a new instance of the same database container, then you might want to change the port settings the new container listens on so that it doesn’t conflict with the existing container. Sometimes you might want to increase the memory available to the container if the program needs more resources to handle a heavy workload or set the environment variables to provide specific configuration details the program needs to function properly.

The `docker run` command offers a powerful way to override these defaults and tailor the container's behavior to your liking. The command offers several flags that let you to customize container behavior on the fly.

Here's a few ways you can achieve this.

Overriding the network ports

Sometimes you might want to use separate database instances for development and testing purposes. Running these database instances on the same port might conflict. You can use the `-p` option in `docker run` to map container ports to host ports, allowing you to run the multiple instances of the container without any conflict.

```
$ docker run -d -p HOST_PORT:CONTAINER_PORT postgres
```

Setting environment variables

This option sets an environment variable `foo` inside the container with the value `bar`.

```
$ docker run -e foo=bar postgres env
```

You will see output like the following:

```
HOSTNAME=2042f2e6ebe4
foo=bar
```

Tip

The `.env` file acts as a convenient way to set environment variables for your Docker containers without cluttering your command line with numerous `-e` flags. To use a `.env` file, you can pass `--env-file` option with the `docker run` command.

```
$ docker run --env-file .env postgres env
```

Restricting the container to consume the resources

You can use the `--memory` and `--cpus` flags with the `docker run` command to restrict how much CPU and memory a container can use. For example, you can set a memory limit for the Python API container, preventing it from consuming excessive resources on your host. Here's the command:

```
$ docker run -e POSTGRES_PASSWORD=secret --memory="512m" --cpus="0.5"
postgres
```

This command limits container memory usage to 512 MB and defines the CPU quota of 0.5 for half a core.

Monitor the real-time resource usage

You can use the `docker stats` command to monitor the real-time resource usage of running containers. This helps you understand whether the allocated resources are sufficient or need adjustment.
By effectively using these `docker run` flags, you can tailor your containerized application's behavior to fit your specific requirements.

Try it out

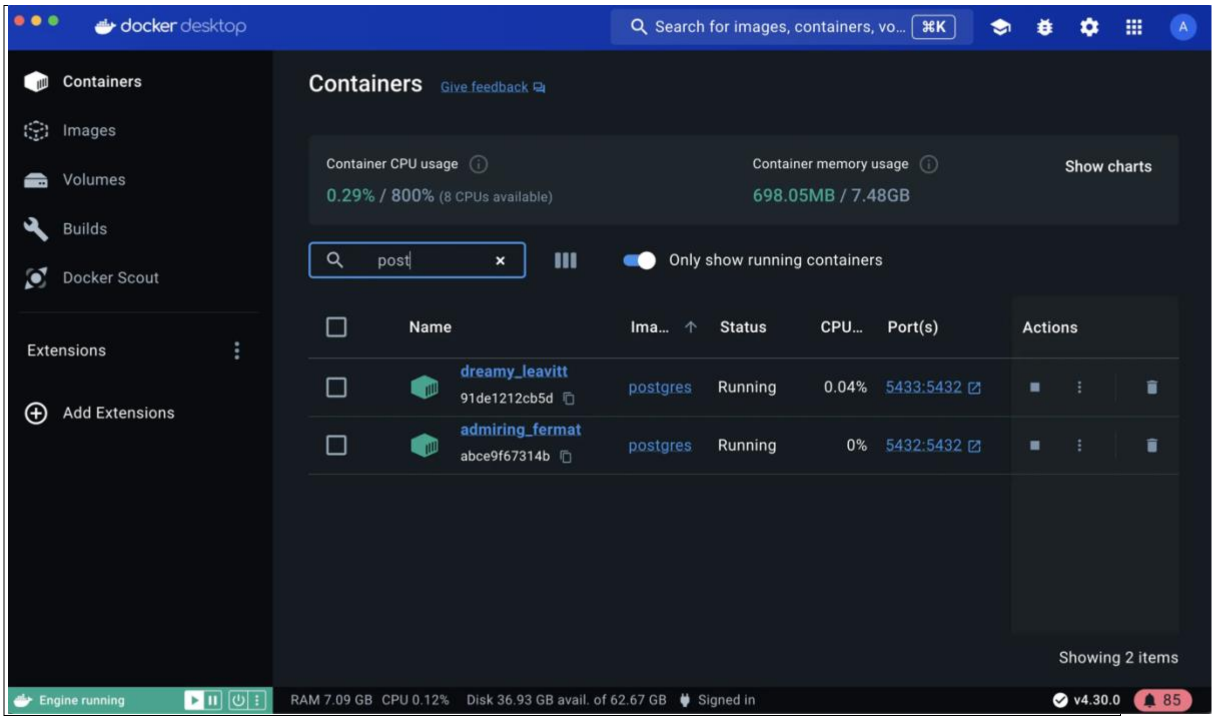
In this hands-on guide, you'll see how to use the `docker run` command to override the container defaults.

1. Download and install Docker Desktop.

Run multiple instance of the Postgres database

1. Start a container using the Postgres image with the following command:
2.
```
$ docker run -d -e POSTGRES_PASSWORD=secret -p 5432:5432 postgres
```
   This will start the Postgres database in the background, listening on the standard container port `5432` and mapped to port `5432` on the host machine.
3. Start a second Postgres container mapped to a different port.
4.
```
$ docker run -d -e POSTGRES_PASSWORD=secret -p 5433:5432 postgres
```
   This will start another Postgres container in the background, listening on the standard postgres port `5432` in the container, but mapped to port `5433` on the host machine. You override the host port just to ensure that this new container doesn't conflict with the existing running container.

5. Verify that both containers are running by going to the **Containers** view in the Docker Desktop Dashboard.



Containers

Container CPU usage 0.29% / 800% (8 CPUs available)

Container memory usage 698.05MB / 7.48GB

post

Name	Ima...	Status	CPU...	Port(s)	Actions
dreamy_leavitt 91de1212cb5d	postgres	Running	0.04%	5433:5432	
admiring_fermat abce9f67314b	postgres	Running	0%	5432:5432	

Showing 2 items

RAM 7.09 GB CPU 0.12% Disk 36.93 GB avail. of 62.67 GB Signed in v4.30.0 85

Run Postgres container in a controlled network

By default, containers automatically connect to a special network called a bridge network when you run them. This bridge network acts like a virtual bridge, allowing containers on the same host to communicate with each other while keeping them isolated from the outside world and other hosts. It's a convenient starting point for most container interactions. However, for specific scenarios, you might want more control over the network configuration.

Here's where the custom network comes in. You create a custom network by passing `--network` flag with the `docker run` command. All containers without a `--network` flag are attached to the default bridge network.

Follow the steps to see how to connect a Postgres container to a custom network.

1. Create a new custom network by using the following command:

```
$ docker network create mynetwork
```

2. Verify the network by running the following command:

```
$ docker network ls
```

This command lists all networks, including the newly created "mynetwork".

3. Connect Postgres to the custom network by using the following command:

```
$ docker run -d -e POSTGRES_PASSWORD=secret -p 5434:5432 --network mynetwork postgres
```

This will start Postgres container in the background, mapped to the host port 5434 and attached to the `mynetwork` network. You passed the `--network` parameter to override the container default by connecting the container to custom Docker network for better isolation and communication with other containers. You can use `docker network inspect` command to see if the container is tied to this new bridge network.

Key difference between default bridge and custom networks

1. DNS resolution: By default, containers connected to the default bridge network can communicate with each other, but only by IP address. (unless you use `--link` option which is considered legacy). It is not recommended for production use due to the various technical shortcomings. On a custom network, containers can resolve each other by name or alias.
2. Isolation: All containers without a `--network` specified are attached to the default bridge network, hence can be a risk, as unrelated containers are then able to communicate. Using a custom network provides a scoped network in which only containers attached to that network are able to communicate, hence providing better isolation.

Manage the resources

By default, containers are not limited in their resource usage. However, on shared systems, it's crucial to manage resources effectively. It's important not to let a running container consume too much of the host machine's memory.

This is where the `docker run` command shines again. It offers flags like `--memory` and `--cpus` to restrict how much CPU and memory a container can use.

```
$ docker run -d -e POSTGRES_PASSWORD=secret --memory="512m" --cpus=".5"
postgres
```

The `--cpus` flag specifies the CPU quota for the container. Here, it's set to half a CPU core (0.5) whereas the `--memory` flag specifies the memory limit for the container. In this case, it's set to 512 MB.

Override the default CMD and ENTRYPOINT in Docker Compose

Sometimes, you might need to override the default commands (`CMD`) or entry points (`ENTRYPOINT`) defined in a Docker image, especially when using Docker Compose.

1. Create a `compose.yml` file with the following content:

```
services:
  postgres:
    image: postgres
    entrypoint: ["docker-entrypoint.sh", "postgres"]
    command: ["-h", "localhost", "-p", "5432"]
    environment:
      POSTGRES_PASSWORD: secret
```

The Compose file defines a service named `postgres` that uses the official Postgres image, sets an entrypoint script, and starts the container with password authentication.

8. Bring up the service by running the following command:

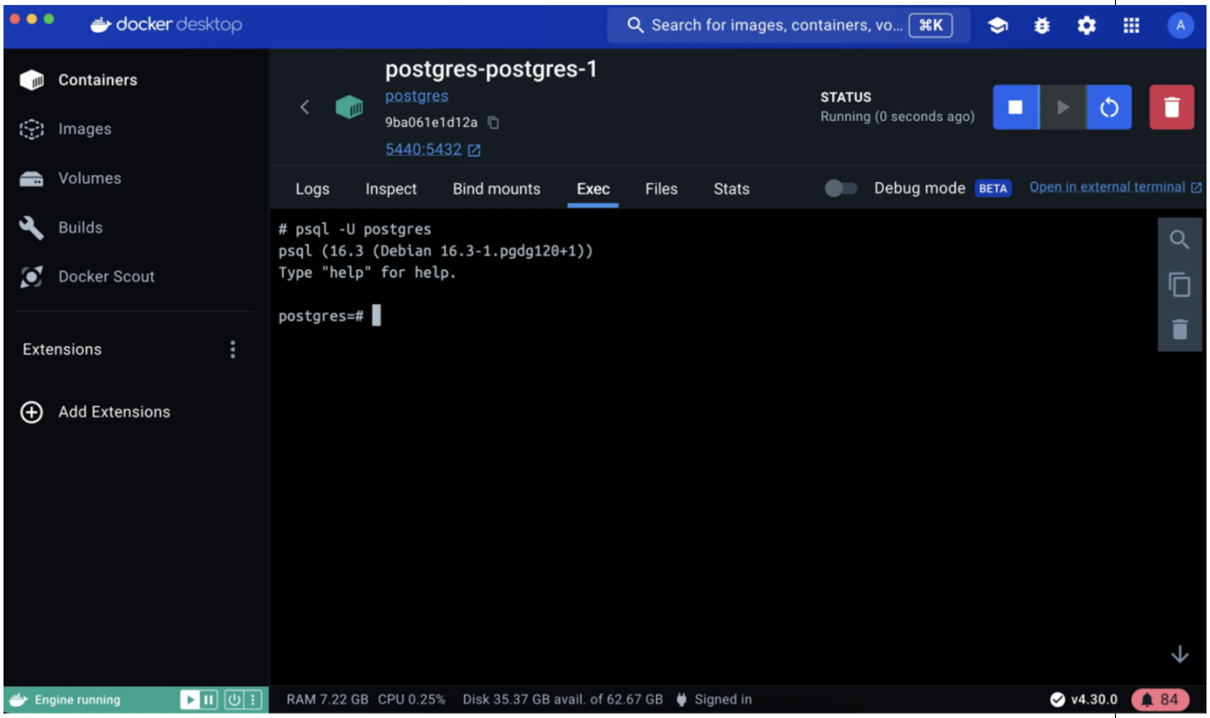
```
$ docker compose up -d
```

This command starts the Postgres service defined in the Docker Compose file.

10. Verify the authentication with Docker Desktop Dashboard.

Open the Docker Desktop Dashboard, select the **Postgres** container and select **Exec** to enter into the container shell. You can type the following command to connect to the Postgres database:

```
# psql -U postgres
```



```
# psql -U postgres
psql (16.3 (Debian 16.3-1.pgdg120+1))
Type "help" for help.

postgres=#
```

Note

The PostgreSQL image sets up trust authentication locally so you may notice a password isn't required when connecting from localhost (inside the same container). However, a password will be required if connecting from a different host/container.

Override the default CMD and ENTRYPOINT with `docker run`

You can also override defaults directly using the `docker run` command with the following command:

```
$ docker run -e POSTGRES_PASSWORD=secret postgres docker-entrypoint.sh
-h localhost -p 5432
```

This command runs a Postgres container, sets an environment variable for password authentication, overrides the default startup commands and configures hostname and port mapping.

Additional resources

- Ways to set environment variables with Compose
- What is a container

Next steps

Now that you have learned about overriding container defaults, it's time to learn how to persist container data.

Persisting container data

Explanation

When a container starts, it uses the files and configuration provided by the image. Each container is able to create, modify, and delete files and does so without affecting any other containers. When the container is deleted, these file changes are also deleted.

While this ephemeral nature of containers is great, it poses a challenge when you want to persist the data. For example, if you restart a database container, you might not want to start with an empty database. So, how do you persist files?

Container volumes

Volumes are a storage mechanism that provide the ability to persist data beyond the lifecycle of an individual container. Think of it like providing a shortcut or symlink from inside the container to outside the container.

As an example, imagine you create a volume named `log-data`.

```
$ docker volume create log-data
```

When starting a container with the following command, the volume will be mounted (or attached) into the container at `/logs`:

```
$ docker run -d -p 80:80 -v log-data:/logs docker/welcome-to-docker
```

If the volume `log-data` doesn't exist, Docker will automatically create it for you.
When the container runs, all files it writes into the `/logs` folder will be saved in this volume, outside of the container. If you delete the container and start a new container using the same volume, the files will still be there.

Sharing files using volumes

You can attach the same volume to multiple containers to share files between containers. This might be helpful in scenarios such as log aggregation, data pipelines, or other event-driven applications.

Managing volumes

Volumes have their own lifecycle beyond that of containers and can grow quite large depending on the type of data and applications you're using. The following commands will be helpful to manage volumes:

- `docker volume ls` - list all volumes

- `docker volume rm <volume-name-or-id>` - remove a volume (only works when the volume is not attached to any containers)
- `docker volume prune` - remove all unused (unattached) volumes

Try it out

In this guide, you'll practice creating and using volumes to persist data created by a Postgres container. When the database runs, it stores files into the `/var/lib/postgresql/data` directory. By attaching the volume here, you will be able to restart the container multiple times while keeping the data.

Use volumes

1. Download and install Docker Desktop.
2. Start a container using the Postgres image with the following command:

```
3. $ docker run --name=db -e POSTGRES_PASSWORD=secret -d -v
   postgres_data:/var/lib/postgresql/data postgres
```

   This will start the database in the background, configure it with a password, and attach a volume to the directory PostgreSQL will persist the database files.

4. Connect to the database by using the following command:

```
5. $ docker exec -ti db psql -U postgres
```

6. In the PostgreSQL command line, run the following to create a database table and insert two records:

```
7. CREATE TABLE tasks (
8.     id SERIAL PRIMARY KEY,
9.     description VARCHAR(100)
10. );
   INSERT INTO tasks (description) VALUES ('Finish work'), ('Have
   fun');
```

11. Verify the data is in the database by running the following in the PostgreSQL command line:

```
SELECT * FROM tasks;
```

   You should get output that looks like the following:

```
 id | description
----+-------------
  1 | Finish work
  2 | Have fun
(2 rows)
```

12. Exit out of the PostgreSQL shell by running the following command:

```
13.  \q
```

14. Stop and remove the database container. Remember that, even though the container has been deleted, the data is persisted in the `postgres_data` volume.

```
15.  $ docker stop db
16.  $ docker rm db
```

17. Start a new container by running the following command, attaching the same volume with the persisted data:

```
18.  $ docker run --name=new-db -d -v
    postgres_data:/var/lib/postgresql/data postgres
```

You might have noticed that the `POSTGRES_PASSWORD` environment variable has been omitted. That's because that variable is only used when bootstrapping a new database.

19. Verify the database still has the records by running the following command:

```
20.  $ docker exec -ti new-db psql -U postgres -c "SELECT * FROM
    tasks"
```

View volume contents

The Docker Desktop Dashboard provides the ability to view the contents of any volume, as well as the ability to export, import, and clone volumes.

1. Open the Docker Desktop Dashboard and navigate to the **Volumes** view. In this view, you should see the **postgres_data** volume.
2. Select the **postgres_data** volume's name.
3. The **Data** tab shows the contents of the volume and provides the ability to navigate the files. Double-clicking on a file will let you see the contents and make changes.
4. Right-click on any file to save it or delete it.

Remove volumes

Before removing a volume, it must not be attached to any containers. If you haven't removed the previous container, do so with the following command (the `-f` will stop the container first and then remove it):

```
$ docker rm -f new-db
```

There are a few methods to remove volumes, including the following:

- Select the **Delete Volume** option on a volume in the Docker Desktop Dashboard.
- Use the `docker volume rm` command:
- `$ docker volume rm postgres_data`
- Use the `docker volume prune` command to remove all unused volumes:
- `$ docker volume prune`

Additional resources

The following resources will help you learn more about volumes:

- Manage data in Docker
- Volumes
- Volume mounts

Next steps

Now that you have learned about persisting container data, it's time to learn about sharing local files with containers.

Sharing local files with containers

Explanation

Each container has everything it needs to function with no reliance on any pre-installed dependencies on the host machine. Since containers run in isolation, they have minimal influence on the host and other containers. This isolation has a major benefit: containers minimize conflicts with the host system and other containers. However, this isolation also means containers can't directly access data on the host machine by default.

Consider a scenario where you have a web application container that requires access to configuration settings stored in a file on your host system. This file may contain sensitive data such as database credentials or API keys. Storing such sensitive information directly within the container image poses security risks, especially during image sharing. To address this challenge, Docker offers storage options that bridge the gap between container isolation and your host machine's data.

Docker offers two primary storage options for persisting data and sharing files between the host machine and containers: volumes and bind mounts.

Volume versus bind mounts

If you want to ensure that data generated or modified inside the container persists even after the container stops running, you would opt for a volume. See Persisting container data to learn more about volumes and their use cases.

If you have specific files or directories on your host system that you want to directly share with your container, like configuration files or development code, then you would use a bind mount. It's like opening a direct portal between your host and container for sharing. Bind mounts are ideal for development environments where real-time file access and sharing between the host and container are crucial.

Sharing files between a host and container

Both `-v` (or `--volume`) and `--mount` flags used with the `docker run` command let you share files or directories between your local machine (host) and a Docker container. However, there are some key differences in their behavior and usage.
The `-v` flag is simpler and more convenient for basic volume or bind mount operations. If the host location doesn’t exist when using `-v` or `--volume`, a directory will be automatically created.

Imagine you're a developer working on a project. You have a source directory on your development machine where your code resides. When you compile or build your code, the generated artifacts (compiled code, executables, images, etc.) are saved in a separate subdirectory within your source directory. In the following examples, this subdirectory is `/HOST/PATH`. Now you want these build artifacts to be accessible within a Docker container running your application. Additionally, you want the container to automatically access the latest build artifacts whenever you rebuild your code.
Here's a way to use `docker run` to start a container using a bind mount and map it to the container file location.

```
$ docker run -v /HOST/PATH:/CONTAINER/PATH -it nginx
```

The `--mount` flag offers more advanced features and granular control, making it suitable for complex mount scenarios or production deployments. If you use `--mount` to bind-mount a file or directory that doesn't yet exist on the Docker host, the `docker run` command doesn't automatically create it for you but generates an error.

```
$ docker run --mount
type=bind,source=/HOST/PATH,target=/CONTAINER/PATH,readonly nginx
```

Note

Docker recommends using the `--mount` syntax instead of `-v`. It provides better control over the mounting process and avoids potential issues with missing directories.

File permissions for Docker access to host files

When using bind mounts, it's crucial to ensure that Docker has the necessary permissions to access the host directory. To grant read/write access, you can use the `:ro` flag (read-only) or `:rw` (read-write) with the `-v` or `--mount` flag during container creation. For example, the following command grants read-write access permission.

```
$ docker run -v HOST-DIRECTORY:/CONTAINER-DIRECTORY:rw nginx
```

Read-only bind mounts let the container access the mounted files on the host for reading, but it can't change or delete the files. With read-write bind mounts, containers can modify or delete mounted files, and these changes or deletions will also be reflected on the host system. Read-only bind mounts ensures that files on the host can't be accidentally modified or deleted by a container.

Synchronized File Share

As your codebase grows larger, traditional methods of file sharing like bind mounts may become inefficient or slow, especially in development environments where frequent access to files is necessary. Synchronized file shares improve bind mount performance by leveraging synchronized filesystem caches. This optimization ensures that file access between the host and virtual machine (VM) is fast and efficient.

Try it out

In this hands-on guide, you'll practice how to create and use a bind mount to share files between a host and a container.

Run a container

1. Download and install Docker Desktop.
2. Start a container using the httpd image with the following command:
3.
```
$ docker run -d -p 8080:80 --name my_site httpd:2.4
```

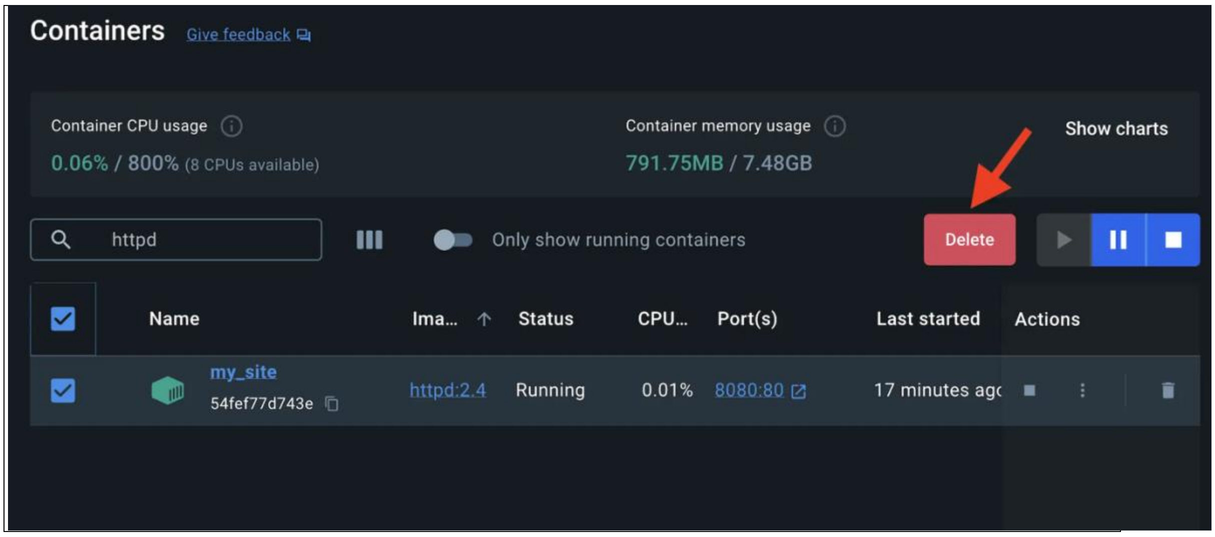
This will start the `httpd` service in the background, and publish the webpage to port `8080` on the host.

4. Open the browser and access http://localhost:8080 or use the curl command to verify if it's working fine or not.
5.
```
$ curl localhost:8080
```

Use a bind mount

Using a bind mount, you can map the configuration file on your host computer to a specific location within the container. In this example, you'll see how to change the look and feel of the webpage by using bind mount:

1. Delete the existing container by using the Docker Desktop Dashboard:



2. Create a new directory called `public_html` on your host system.
3.
```
$ mkdir public_html
```

4. Change the directory to `public_html` and create a file called `index.html` with the following content. This is a basic HTML document that creates a simple webpage that welcomes you with a friendly whale.

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title> My Website with a Whale & Docker!</title>
</head>
<body>
<h1>Whalecome!!</h1>
<p>Look! There's a friendly whale greeting you!</p>
<pre id="docker-art">
   ##         .
  ## ## ##        ==
 ## ## ## ## ##    ===
 /"""""""""""""""""\___/ ===
{                       /  ===-
\______ O           __/
\    \         __/
 \____\_______/

Hello from Docker!
</pre>
</body>
</html>
```

27. It's time to run the container. The `--mount` and `-v` examples produce the same result. You can't run them both unless you remove the `my_site` container after running the first one.

`-v` `--mount`

```
$ docker run -d --name my_site -p 8080:80 -v
.:/usr/local/apache2/htdocs/ httpd:2.4
```

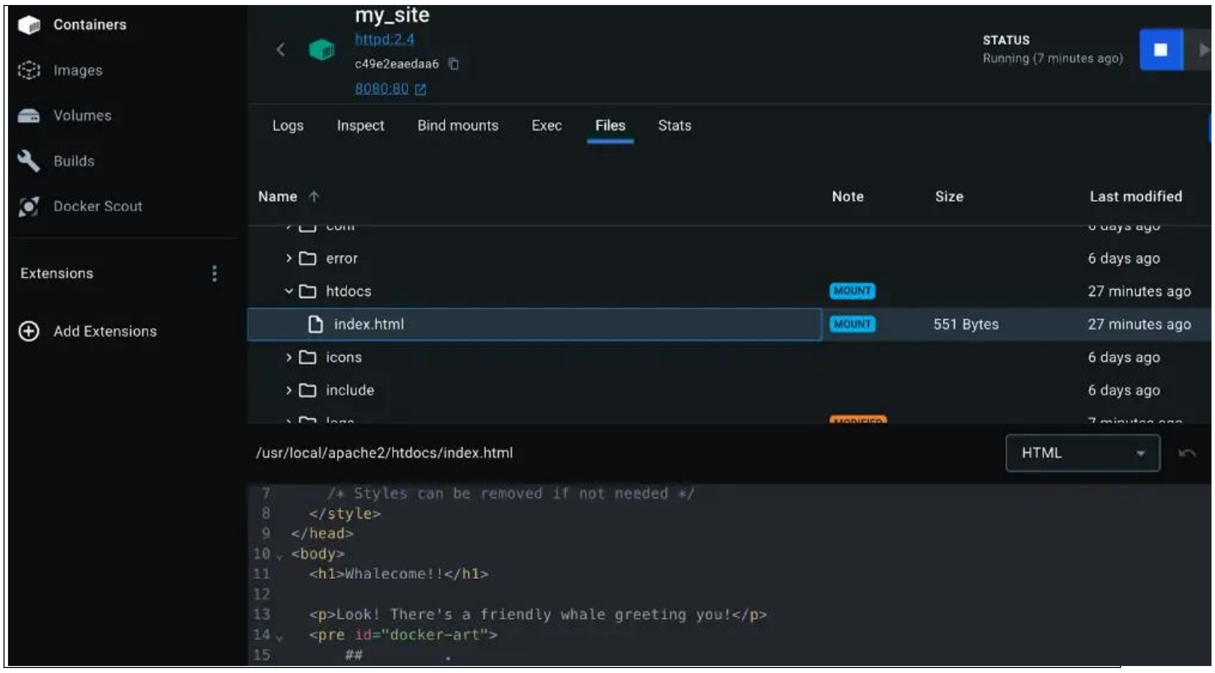
Tip

When using the `-v` or `--mount` flag in Windows PowerShell, you need to provide the absolute path to your directory instead of just `./`. This is because PowerShell handles relative paths differently from bash (commonly used in Mac and Linux environments).

With everything now up and running, you should be able to access the site via http://localhost:8080 and find a new webpage that welcomes you with a friendly whale.

Access the file on the Docker Desktop Dashboard

1. You can view the mounted files inside a container by selecting the container's **Files** tab and then selecting a file inside the `/usr/local/apache2/htdocs/` directory. Then, select **Open file editor**.



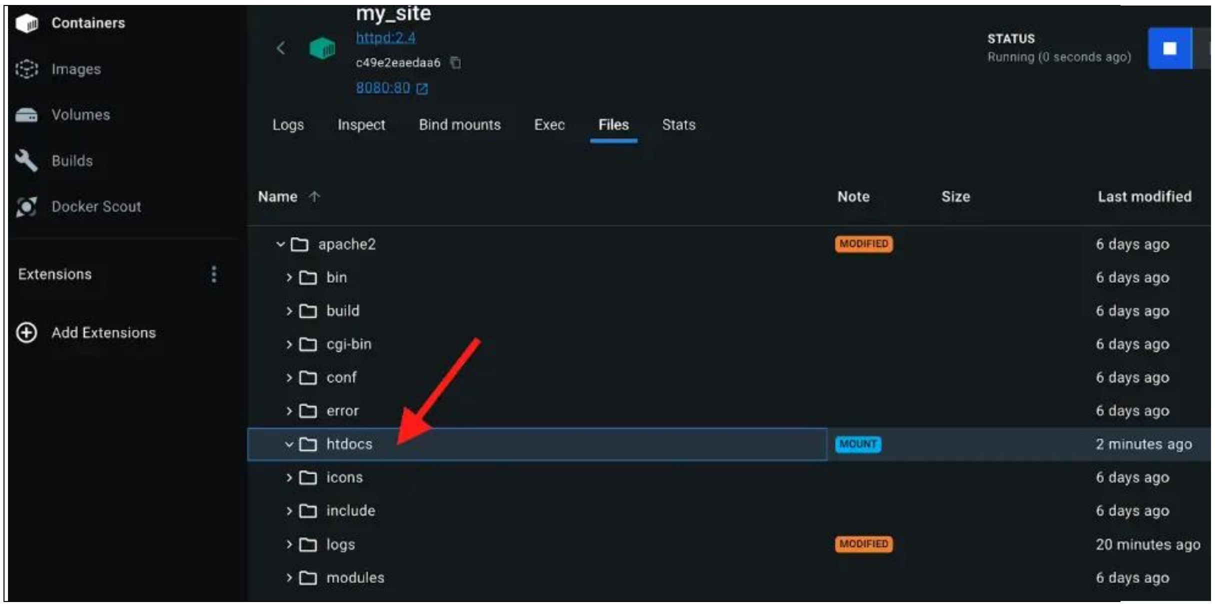
Name	Note	Size	Last modified
error			6 days ago
htdocs	MOUNT		27 minutes ago
index.html	MOUNT	551 Bytes	27 minutes ago
icons			6 days ago
include			6 days ago

/usr/local/apache2/htdocs/index.html

```
    /* Styles can be removed if not needed */
  </style>
</head>
<body>
  <h1>Whalecome!!</h1>

  <p>Look! There's a friendly whale greeting you!</p>
  <pre id="docker-art">
      ##        .
```

2. Delete the file on the host and verify the file is also deleted in the container. You will find that the files no longer exist under **Files** in the Docker Desktop Dashboard.



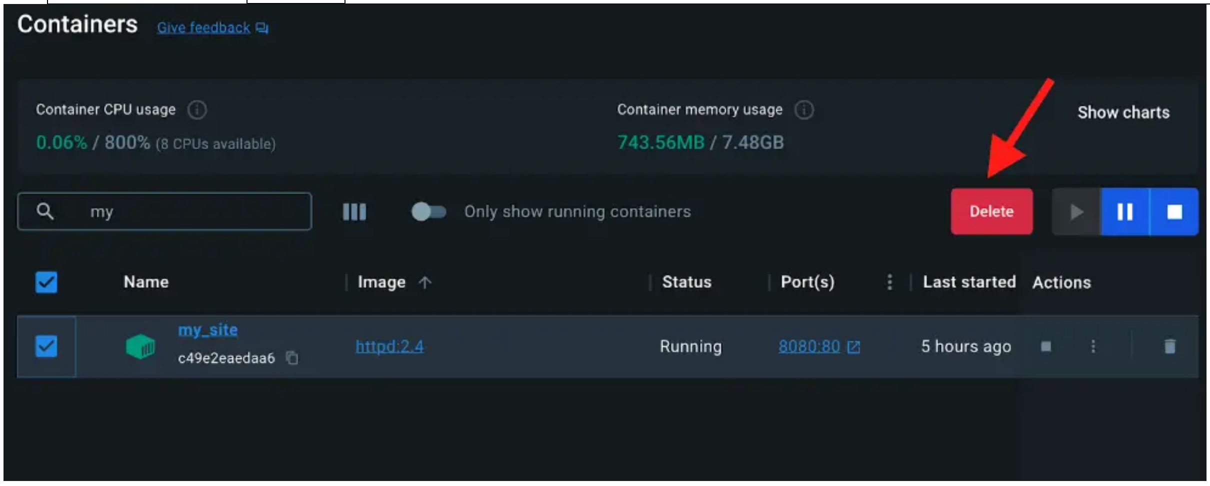
Name	Note	Size	Last modified
apache2	MODIFIED		6 days ago
bin			6 days ago
build			6 days ago
cgi-bin			6 days ago
conf			6 days ago
error			6 days ago
htdocs	MOUNT		2 minutes ago
icons			6 days ago
include			6 days ago
logs	MODIFIED		20 minutes ago
modules			6 days ago

3. Recreate the HTML file on the host system and see that file re-appears under the **Files** tab under **Containers** on the Docker Desktop Dashboard. By now, you will be able to access the site too.

Stop your container

The container continues to run until you stop it.

1. Go to the **Containers** view in the Docker Desktop Dashboard.
2. Locate the container you'd like to stop.
3. Select the **Delete** action in the Actions column.



Name	Image	Status	Port(s)	Last started	Actions
my_site c49e2eaedaa6	httpd:2.4	Running	8080:80	5 hours ago	

Additional resources

The following resources will help you learn more about bind mounts:

- Manage data in Docker

- Volumes
- Bind mounts
- Running containers
- Troubleshoot storage errors
- Persisting container data

Next steps

Now that you have learned about sharing local files with containers, it’s time to learn about multi-container applications.

Multi-container applications

Explanation

Starting up a single-container application is easy. For example, a Python script that performs a specific data processing task runs within a container with all its dependencies. Similarly, a Node.js application serving a static website with a small API endpoint can be effectively containerized with all its necessary libraries and dependencies. However, as applications grow in size, managing them as individual containers becomes more difficult.

Imagine the data processing Python script needs to connect to a database. Suddenly, you're now managing not just the script but also a database server within the same container. If the script requires user logins, you'll need an authentication mechanism, further bloating the container size.

One best practice for containers is that each container should do one thing and do it well. While there are exceptions to this rule, avoid the tendency to have one container do multiple things.

Now you might ask, "Do I need to run these containers separately? If I run them separately, how shall I connect them all together?"

While `docker run` is a convenient tool for launching containers, it becomes difficult to manage a growing application stack with it. Here's why:

- Imagine running several `docker run` commands (frontend, backend, and database) with different configurations for development, testing, and production environments. It's error-prone and time-consuming.
- Applications often rely on each other. Manually starting containers in a specific order and managing network connections become difficult as the stack expands.
- Each application needs its `docker run` command, making it difficult to scale individual services. Scaling the entire application means potentially wasting resources on components that don't need a boost.
- Persisting data for each application requires separate volume mounts or configurations within each `docker run` command. This creates a scattered data management approach.
- Setting environment variables for each application through separate `docker run` commands is tedious and error-prone.

That's where Docker Compose comes to the rescue.

Docker Compose defines your entire multi-container application in a single YAML file called `compose.yml`. This file specifies configurations for all your containers, their dependencies, environment variables, and even volumes and networks. With Docker Compose:

- You don't need to run multiple `docker run` commands. All you need to do is define your entire multi-container application in a single YAML file. This centralizes configuration and simplifies management.
- You can run containers in a specific order and manage network connections easily.
- You can simply scale individual services up or down within the multi-container setup. This allows for efficient allocation based on real-time needs.
- You can implement persistent volumes with ease.
- It's easy to set environment variables once in your Docker Compose file.

By leveraging Docker Compose for running multi-container setups, you can build complex applications with modularity, scalability, and consistency at their core.

Try it out

In this hands-on guide, you'll first see how to build and run a counter web application based on Node.js, an Nginx reverse proxy, and a Redis database using the `docker run` commands. You'll also see how you can simplify the entire deployment process using Docker Compose.

Set up

1. Get the sample application. If you have Git, you can clone the repository for the sample application. Otherwise, you can download the sample application. Choose one of the following options.

Clone with git Download

Use the following command in a terminal to clone the sample application repository.

```
$ git clone https://github.com/dockersamples/nginx-node-redis
```

Navigate into the `nginx-node-redis` directory:

```
$ cd nginx-node-redis
```

Inside this directory, you'll find two sub-directories - `nginx` and `web`.

---

2. Download and install Docker Desktop.

Build the images

1. Navigate into the `nginx` directory to build the image by running the following command:
2. `$ docker build -t nginx .`
3. Navigate into the `web` directory and run the following command to build the first web image:
4. `$ docker build -t web .`

Run the containers

1. Before you can run a multi-container application, you need to create a network for them all to communicate through. You can do so using the `docker network create` command:
2. `$ docker network create sample-app`
3. Start the Redis container by running the following command, which will attach it to the previously created network and create a network alias (useful for DNS lookups):
4. `$ docker run -d  --name redis --network sample-app --network-alias redis redis`
5. Start the first web container by running the following command:
6. `$ docker run -d --name web1 -h web1 --network sample-app --network-alias web1 web`
7. Start the second web container by running the following:
8. `$ docker run -d --name web2 -h web2 --network sample-app --network-alias web2 web`
9. Start the Nginx container by running the following command:
10. `$ docker run -d --name nginx --network sample-app  -p 80:80 nginx`

Note

Nginx is typically used as a reverse proxy for web applications, routing traffic to backend servers. In this case, it routes to the Node.js backend containers (web1 or web2).

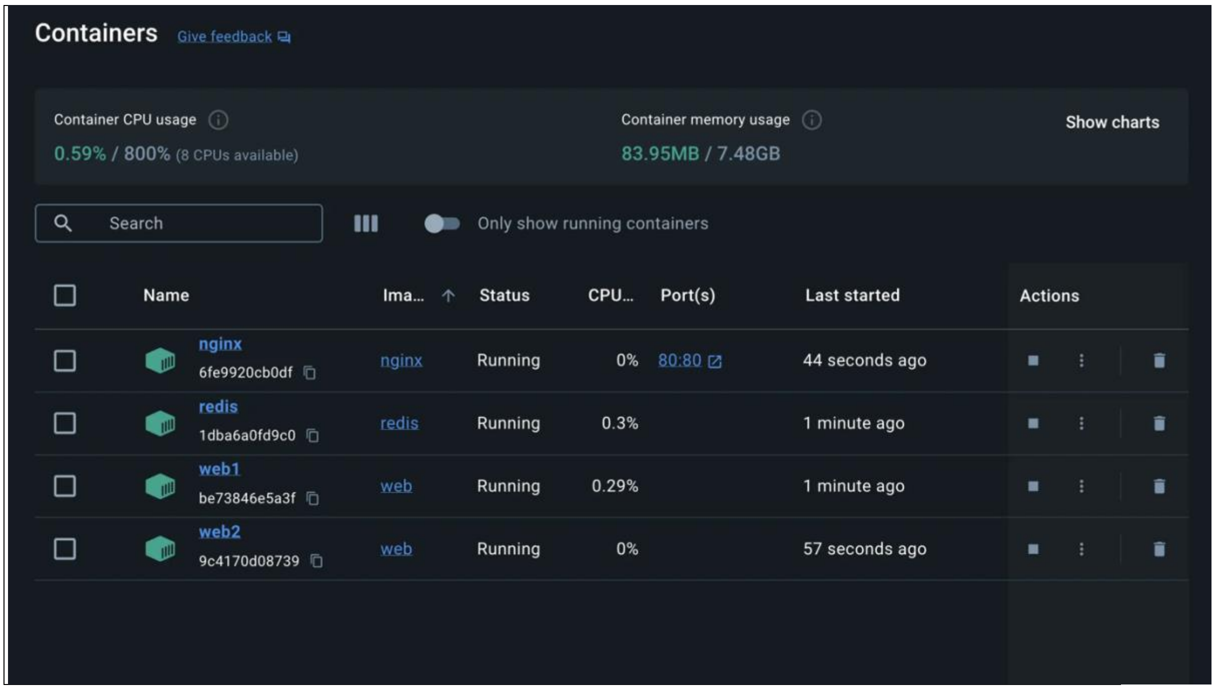
11. Verify the containers are up by running the following command:
12. `$ docker ps`

You will see output like the following:

```
CONTAINER ID   IMAGE      COMMAND                         CREATED
STATUS                  PORTS                     NAMES
```

```
2cf7c484c144   nginx      "/docker-entrypoint.…"   9 seconds ago
Up 8 seconds        0.0.0.0:80->80/tcp   nginx
7a070c9ffeaa   web        "docker-entrypoint.s…"   19 seconds ago
Up 18 seconds                            web2
6dc6d4e60aaf   web        "docker-entrypoint.s…"   34 seconds ago
Up 33 seconds                            web1
008e0ecf4f36   redis      "docker-entrypoint.s…"   About a minute
ago   Up About a minute   6379/tcp             redis
```

13. If you look at the Docker Desktop Dashboard, you can see the containers and dive deeper into their configuration.



Containers

Container CPU usage 0.59% / 800% (8 CPUs available)

Container memory usage 83.95MB / 7.48GB

Name	Image	Status	CPU	Port(s)	Last started
nginx 6fe9920cb0df	nginx	Running	0%	80:80	44 seconds ago
redis 1dba6a0fd9c0	redis	Running	0.3%		1 minute ago
web1 be73846e5a3f	web	Running	0.29%		1 minute ago
web2 9c4170d08739	web	Running	0%		57 seconds ago

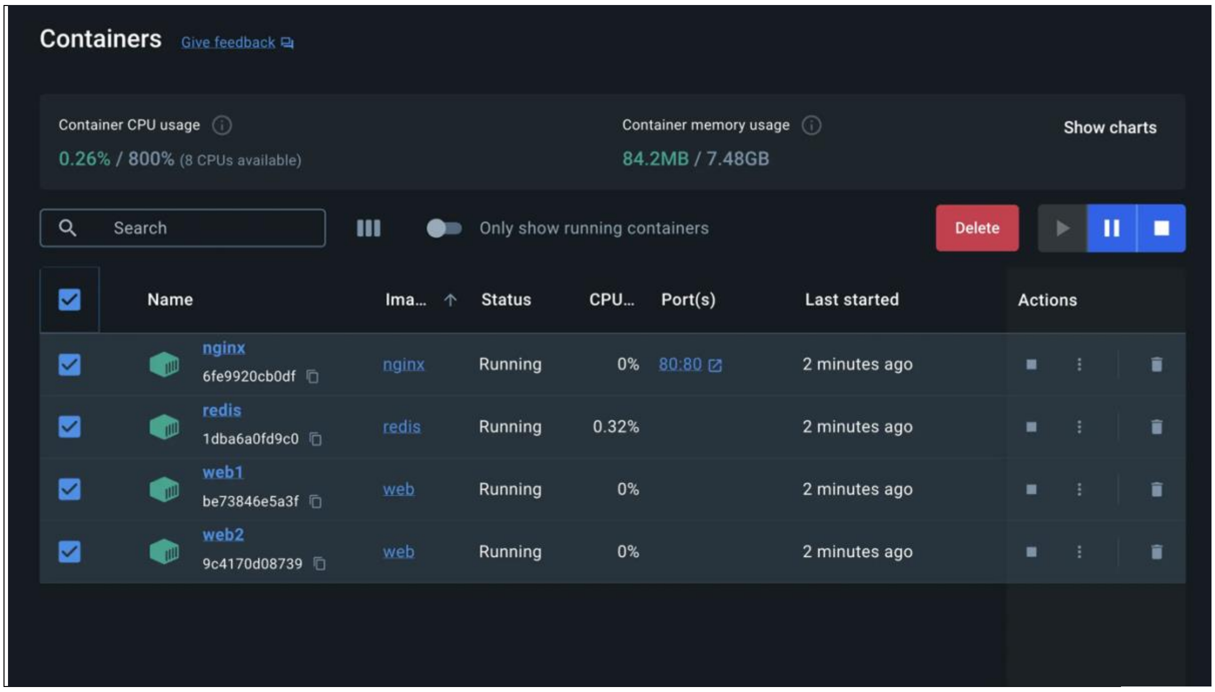
14. With everything up and running, you can open http://localhost in your browser to see the site. Refresh the page several times to see the host that’s handling the request and the total number of requests:

```
15.  web2: Number of visits is: 9
16.  web1: Number of visits is: 10
17.  web2: Number of visits is: 11
18.  web1: Number of visits is: 12
```

Note

You might have noticed that Nginx, acting as a reverse proxy, likely distributes incoming requests in a round-robin fashion between the two backend containers. This means each request might be directed to a different container (web1 and web2) on a rotating basis. The output shows consecutive increments for both the web1 and web2 containers and the actual counter value stored in Redis is updated only after the response is sent back to the client.

19. You can use the Docker Desktop Dashboard to remove the containers by selecting the containers and selecting the **Delete** button.



Container CPU usage
0.26% / 800% (8 CPUs available)

Container memory usage
84.2MB / 7.48GB

Name	Ima...	Status	CPU...	Port(s)	Last started	Actions
nginx 6fe9920cb0df	nginx	Running	0%	80:80	2 minutes ago	
redis 1dba6a0fd9c0	redis	Running	0.32%		2 minutes ago	
web1 be73846e5a3f	web	Running	0%		2 minutes ago	
web2 9c4170d08739	web	Running	0%		2 minutes ago	

Simplify the deployment using Docker Compose

Docker Compose provides a structured and streamlined approach for managing multi-container deployments. As stated earlier, with Docker Compose, you don't need to run multiple `docker run` commands. All you need to do is define your entire multi-container application in a single YAML file called `compose.yml`. Let's see how it works.

Navigate to the root of the project directory. Inside this directory, you'll find a file named `compose.yml`. This YAML file is where all the magic happens. It defines all the services that make up your application, along with their configurations. Each service specifies its image, ports, volumes, networks, and any other settings necessary for its functionality.

1. Use the `docker compose up` command to start the application:

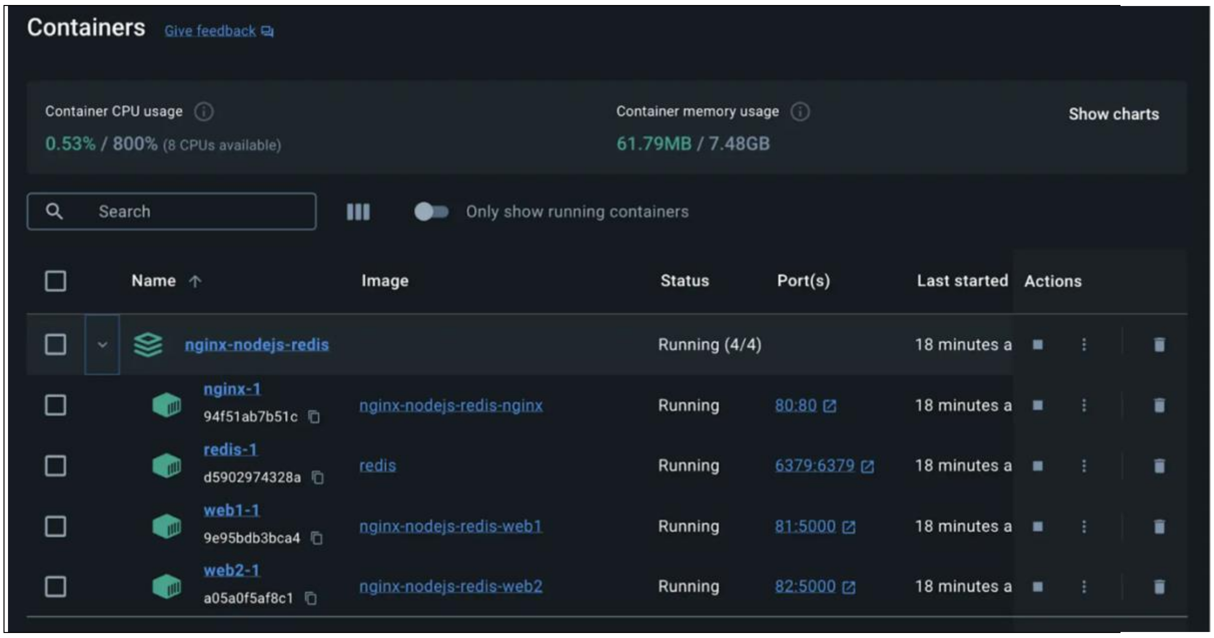
```
2. $ docker compose up -d --build
```

When you run this command, you should see output similar to the following:

```
Running 5/5
✔ Network nginx-nodejs-redis_default     Created
0.0s
✔ Container nginx-nodejs-redis-web1-1    Started
0.1s
✔ Container nginx-nodejs-redis-redis-1   Started
0.1s
```

```
✓ Container nginx-nodejs-redis-web2-1   Started
0.1s
✓ Container nginx-nodejs-redis-nginx-1  Started
```

3. If you look at the Docker Desktop Dashboard, you can see the containers and dive deeper into their configuration.



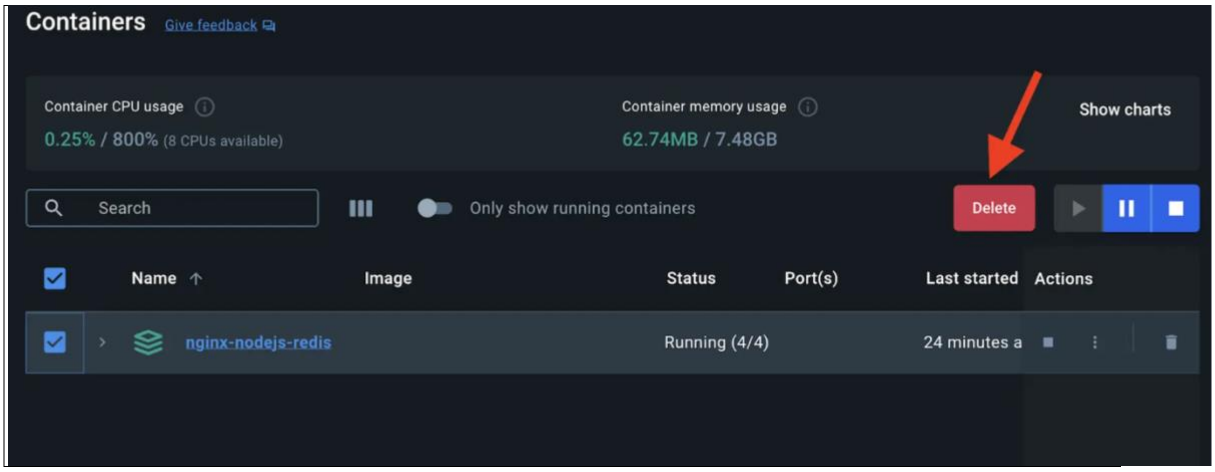
Containers Give feedback

Container CPU usage: 0.53% / 800% (8 CPUs available)
Container memory usage: 61.79MB / 7.48GB
Show charts

Search | Only show running containers

	Name ↑	Image	Status	Port(s)	Last started	Actions
☐	nginx-nodejs-redis		Running (4/4)		18 minutes a	
☐	nginx-1 94f51ab7b51c	nginx-nodejs-redis-nginx	Running	80:80	18 minutes a	
☐	redis-1 d5902974328a	redis	Running	6379:6379	18 minutes a	
☐	web1-1 9e95bdb3bca4	nginx-nodejs-redis-web1	Running	81:5000	18 minutes a	
☐	web2-1 a05a0f5af8c1	nginx-nodejs-redis-web2	Running	82:5000	18 minutes a	

4. Alternatively, you can use the Docker Desktop Dashboard to remove the containers by selecting the application stack and selecting the **Delete** button.



Containers Give feedback

Container CPU usage: 0.25% / 800% (8 CPUs available)
Container memory usage: 62.74MB / 7.48GB
Show charts

Search | Only show running containers | Delete

	Name ↑	Image	Status	Port(s)	Last started	Actions
☑	nginx-nodejs-redis		Running (4/4)		24 minutes a	

In this guide, you learned how easy it is to use Docker Compose to start and stop a multi-container application compared to `docker run` which is error-prone and difficult to manage.

Additional resources

- `docker container run` CLI reference

- What is Docker Compose