## **What Are Docker Volumes**

The purpose of using Docker volumes is to persist data outside the container so it can be backed up or shared.

Docker volumes are dependent on Docker’s file system and are the [preferred method](https://docs.docker.com/storage/#good-use-cases-for-volumes) of persisting data for Docker containers and services. When a container is started, Docker loads the read-only image layer, adds a read-write layer on top of the image stack, and mounts volumes onto the container filesystem.

## **Why Docker Volumes?**

If you are using Docker for development, you must be familiar with the **-v** or **--volume** flag that lets you mount your local files into the container. For instance, you can mount your local **./target** onto the **/usr/share/nginx/html** directory container or an nginx container to visualize your html files.

>\_echo "<h1>Hello from Host</h1>" > ./target/index.html  
docker run -it --rm --name nginx -p 8080:80 -v "$(pwd)"/target:/usr/share/nginx/html nginx

Navigate to [**http://localhost:8080/**](http://localhost:8080/) and you should see “Hello from Host”.

This is called a [bind mount](https://docs.docker.com/storage/bind-mounts/) and is commonly used by developers. But, if you are using Docker Desktop on Windows or MacOS bind, mounts have significant performance issues. As a result, using volumes may be the best alternative for holding state between container runs.

Unlike bind mount, where you can mount any directory from your host, volumes are stored in a single location (most likely /var/lib/docker/volumes/ on unix systems) and greatly facilitates managing data (backup, restore, and migration). Docker volumes can safely be shared between several running containers.

You can also save data to a remote server or in cloud Docker volumes with alternative [volume drivers](https://docs.docker.com/storage/volumes/#use-a-volume-driver) like [**sshfs**](https://github.com/vieux/docker-volume-sshfs).

In addition, Docker enables you to manage volume with the command line **docker volume**, making their management simple.

## **Creating and Managing Docker Volumes**

In this section, you’ll learn how to create a Docker volume implicitly and explicitly and then declare it from a Docker file. Then you’ll learn how to view a data volume, mount it to a container, and configure it using **docker-compose**.

### **Create a Docker Volume Implicitly**

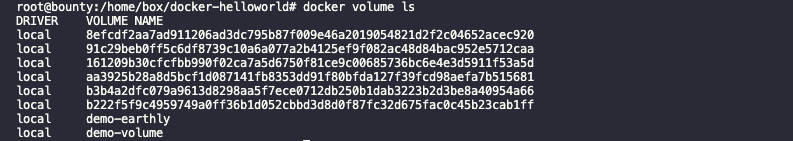
The easiest way to create and use a volume is with **docker run** and the **-v** or **--volume** flag. This flag takes three arguments separated by **:**:

-v <source>:<destination>:<options>

If the “source” is a path that was used in the previous example, Docker will use a mount bind. If the “source” is a name, then Docker tries to find this volume or creates one if one cannot be found. Below, the previous example has been updated to use a volume instead of a mount bind:

docker run -it --rm --name nginx -p 8080:80 -v demo-earthly:/usr/share/nginx/html nginx

You can check to make sure the container was properly created with **docker volume ls** which lists all existing volumes.

**docker volume ls**

*Note that the volume in question is not empty. If a volume is completely empty, the container’s content is copied to the volume.*

You can check the status of your volumes on Linux. This gives you a chance to see where volumes are stored:

>\_ls /var/lib/docker/volumes/target/\_data/demo-earthly

On Mac and Windows it’s a bit more tricky. In order to keep things simple, you can mount the volume on an [ubuntu container](https://ubuntu.com/containers) and use **ls** to see the content of your volume:

>\_docker run -it --rm -v demo-earthly:/opt/demo-earthly ubuntu ls /opt/demo-earthly

### **Create a Docker Volume Explicitly**

Alternatively you can use the **docker volume create** command to explicitly create a data volume. This command gives you the option to choose and configure the [volume driver](https://docs.docker.com/storage/volumes/#use-a-volume-driver). The implicit creation of volumes always uses the **local** driver with default settings.

>\_docker volume create --name demo-earthly

### **Declare a Docker Volume from Dockerfile**

Volumes can be declared in your Dockerfile using the **VOLUME** statement. This statement declares that a specific path of the container must be mounted to a Docker volume. When you run the container, Docker will create an anonymous volume (volume with a unique id as the name) and mount it to the specified path.

Dockerfile**FROM** nginx:latest  
  
RUN echo "<h1>Hello from Volume</h1>" > /usr/share/nginx/html/index.html  
VOLUME /usr/share/nginx/html

Lets build and run your new image:

>\_docker build -t demo-earthly .  
docker run -p 8080:80 demo-earthly

You can now validate that nginx serves your message at [**http://localhost:8080/**](http://localhost:8080/).

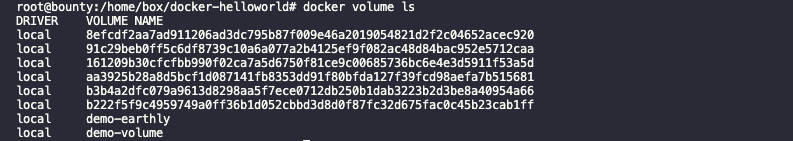
More importantly, an anonymous Docker volume has been created, and every time you start a new container, another volume is created with the content of **/usr/share/nginx/html**.

>\_docker volume ls

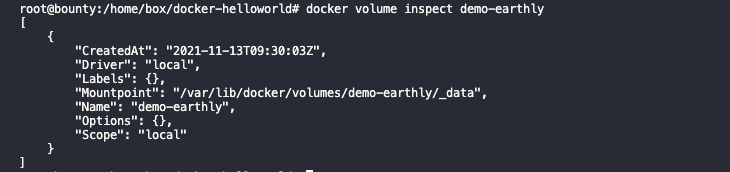
From the above example, a volume directory **data** with the text file **test** containing “Hello from Volume” is created.

### **View a Data Volume**

To manage your data, sometimes you need to list data volumes from the command line as a point of reference, which is faster than repeatedly checking the configuration files. You can use the **docker volume ls** command to view a list of data volumes.

**docker volume ls**

Use the **docker volume inspect** command to view the data volume details.

**docker volume inspect**

### **Mount a Volume to a Container**

As you have seen through the various examples **-v** and **--volume** are the most common way to mount a volume to a container using the syntax:

-v <name>:<destination>:<options>

One notable option is **ro** which means that the volume will be mounted as read-only:

>\_docker run -it -v demo-volume:/data:ro ubuntu

Try to write into the folder/data to validate that the volume is in read-only mode:

>\_echo "test" > /data/test

An alternative to **-v** is to add the **—mount** option to the **docker run** command. **--mount** is the more verbose counterpart of **-v**.

To launch a container and mount a data volume to it, follow this syntax:

>\_docker run --mount source=[volume\_name],destination=[path\_in\_container] [docker\_image]

Replace **[path in container]** with the path to attach the Docker volume **[volume\_name]** in the container.

For example, run the following command to start an Ubuntu container and mount the data volume to it.

>\_docker run -it --name=example --mount source=demo-volume,destination=/data ubuntu

Remember if the volume doesn’t exist Docker will create it for you.

List the contents of the container to see if the volume is mounted successfully. You should find the Docker volume name defined in the above data syntax.

Container content

### **Configure a Volume Using docker-compose**

Although there are many ways to create a volume, it’s more convenient to use the **docker-compose** command to easily share data between multiple containers.

The use of the **volume** property in compose files is very similar to **-v** and **--volume**. That being said, to perform a bind mount (mount a directory from your local machine), you can use a relative path unlike **-v** with the command **docker run** that requires an absolute path.

docker-compose.yamlversion**:** "3.2"  
services**:**  
web**:**  
image**:** nginx:latest  
 ports**:**  
 **-** 8080:80  
 volumes**:**  
 **-** ./target:/usr/share/nginx/html

The containers and hosts in the above configuration use **volumes** in the **services** definition (**web**) to mount **./target** from the host to **/usr/share/nginx/html** of the container. As with the first example, if you navigate to [**http://localhost:8080/**](http://localhost:8080/) you should read “Hello from Host”.

With **docker-compose**, volumes must be declared at the same level as **services**. Then you can refer to them by their name.

docker-compose.yamlversion**:** "3.2"  
services**:**  
web**:**  
image**:** nginx:latest  
 ports**:**  
 **-** 8080:80  
 volumes**:**  
 **-** html\_files:/usr/share/nginx/html  
 web1**:**  
image**:** nginx:latest  
 ports**:**  
 **-** 8081:80  
 volumes**:**  
 **-** html\_files:/usr/share/nginx/html  
   
volumes**:**  
html\_files**:**

In this example, you declared a volume named **html\_files** and used it in both **web** and **web1** service. Multiple containers can mount the same volume.

Running **docker-compose up** will create a volume named **<project\_name>\_html\_files** if it doesn’t already exist . Then run **docker volume ls** to list the two volumes created, starting with the project name.

You can also manage container outside of you docker-compose file, but you still need to declare them under **volumes** and set the property **external: true**.

docker-compose.ymlversion**:** "3.2"  
services**:**  
web**:**  
image**:** nginx:latest  
 ports**:**  
 **-** 8080:80  
 volumes**:**  
 **-** html\_files:/usr/share/nginx/html  
   
volumes**:**  
html\_files**:**  
external**:** true

If you don’t have **html\_files**, you can use **docker volume create html\_files** to create it. When you add **external**, Docker will find out if the volume exists; but if it doesn’t, an error will be reported.

### **Copy Files Between Containers from a Shared Volume**

Let’s look at how Docker volumes enable file sharing across containers.

In this example, use the volume and container we previously defined and execute the following commands:

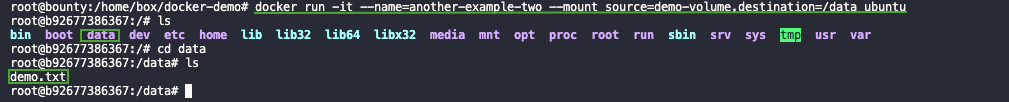
>\_docker create volume demo-earthly

>\_docker run -it --name=another-example --mount source=demo-volume,destination=/data ubuntu

Navigate to the data volume directory and create a file using the command **touch demo.txt**. Exit the container, then launch a new container **another-example-two** with the same data volume:

>\_docker run -it --name=another-example-two --mount source=demo-volume,destination=/data ubuntu

The **demo.txt** file you created in the preceding container should list **another-example** in the output.

Copying files

## **Docker Volume Best Practices**

Now that you’ve learned how to implement Docker volumes, it’s important to keep in mind a few best practices:

* Always [mount volumes as read-only](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Docker_Security_Cheat_Sheet.md#rule-8---set-filesystem-and-volumes-to-read-only) if you only need to read from them.
* Always set the permissions and ownership on a volume.
* Always use [environment variables](https://earthly.dev/blog/understanding-bash) for the host path or volume name in a production environment.

## **Conclusion**

Often, you want your containers to use or persist data beyond the scope of the container’s lifetime. You can use volumes to solve this problem by working with Docker to mount, create, and share volumes between containers.

BIND MOUNTS vs VOLUMES

Let’s take a hypothetical scenario where you are running a database (SQL) in a container (though I believe it is not a good idea to run stateful applications in a container) without any Volumes or Bind Mounts attached. You create some tables and add some rows to them. But, for some reason, you need to delete this container. As soon as the container is deleted, all your tables and their corresponding data get lost.

But Docker provides us with a couple of solutions to persist your data even if the container is deleted. The two possible ways to persist your data are:

* Bind Mounts
* Volumes

Though both methods are similar, there is a slight difference. **Docker manages Volumes** and is usually not affected by other processes running on the same host. In contrast, **Bind Mounts** are just a directory on the host file system and may be modified by other processes other than docker. But the base similarity is that the lifecycle of both solutions does not depend on the container lifecycle. It is possible to confuse the two, so we’ll try to make it clear with an example for both methods:

**Note:** You should have installed docker on your machine if you want to follow along with the activity and commands in the blog. I will be using Docker Desktop on a Windows machine.

Let’s begin…

### **What are Bind Mounts in Docker?**

A Bind Mount is a storage area (file/directory) on your local machine available inside your container. So any changes you make to this storage space (file/directory) from the outside container will be reflected inside the docker container and vice-versa.

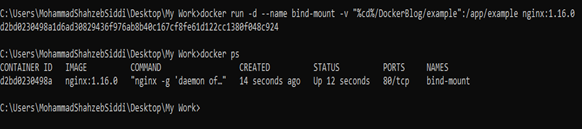
***Please Note:*** *A bind mount will overwrite the content of the docker container.*

Let’s see this with an example. We will be running a container from an Nginx image and use a bind-mount to mount a directory from the host machine into the container.

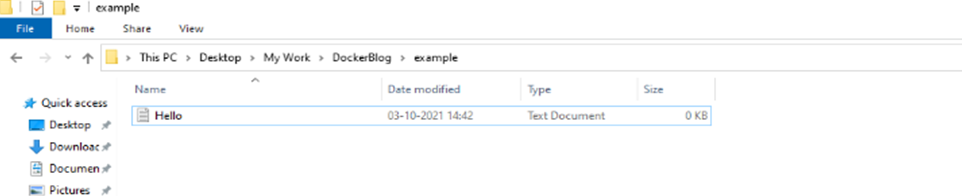
I have an empty directory “**example**” located at “**C:\Users\MohammadShahzebSiddi\Desktop\My Work\DockerBlog**” on a local machine, and I want to access the content of this directory at the location “**/app/example**” from within my container.

The command for running a container with the above requirements will be:

docker run -d — name bind-mount -v “%cd%/DockerBlog/example”:/app/example nginx:1.16.0

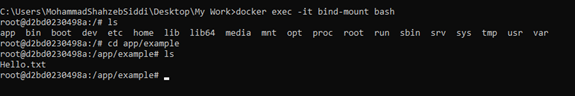


We will try creating a file in the example directory on the host machine and then check if it is available from inside the container.

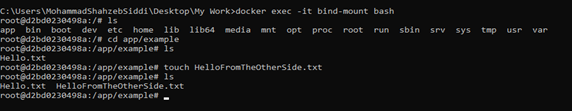


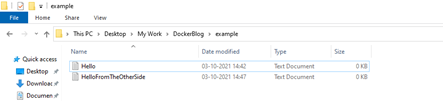
We created an empty text file named “Hello.” Let’s bash into the container and check if we can access it from there. The command for that will be :

docker exec -it bind-mount bash

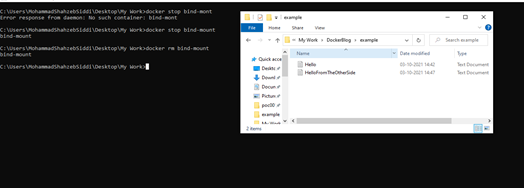


We notice that the “Hello.txt” file is available inside the container. Now let’s create a file inside the container and check if it’s available on the local machine.





We can observe that the changes are reflected on the host machine also. I hope this made a little clear how the Bind Mount works. Now let’s delete the container and see if data is persisted on your local machine. You can notice in the image attached below how these changes persist even after we delete the (Bind-Mount)container.



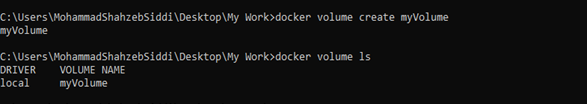
A practical application of Bind Mounts can be when you want to update the source code of your image and run it inside the container without creating a new image. You can do so by creating a Bind Mount and changing the code from your local machine, which will then reflect in your container.

#### 

### **What are Volumes in Docker?**

Volumes are also file-storage areas but act more like a resource that Docker manages. Unlike Bind Mounts, we can explicitly create a volume without having it associated with a container. It is mainly recommended to use Volumes instead of Bind Mounts because of the flexibility and features they provide over the latter. If you want to read more about its advantages, you can visit the official documentation of Docker. For now, we will try to create a named volume and then bind it to a location inside a container. Let’s see how it can be achieved.

> First, we will create a volume with the name “myVolume,” using **docker volume create volume** and get the list of available Volumes using the command **docker volume ls**

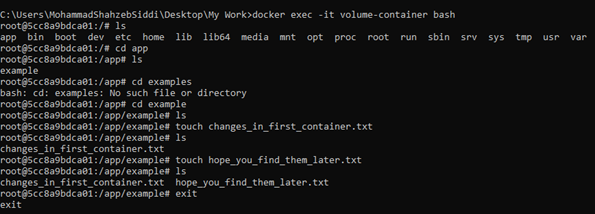


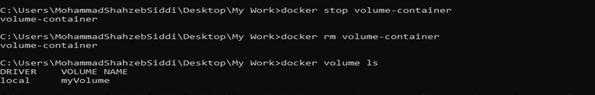
As you can see, our volume is created; now, we would like to attach it to a container before running it. The command to achieve this will be:

**run -d — name volume-container -v myVolume:/app/example nginx:1.16.0:**



Now we’ll insert bash code inside the container -> do some changes at the location the volume is mounted in -> then, stop and delete the container.



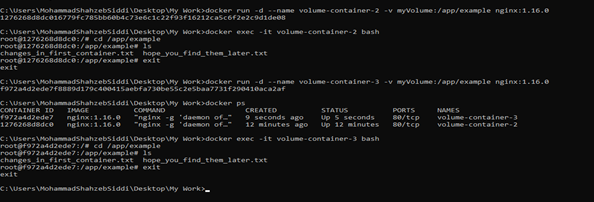


We will create a new container, attach the same volume to it, and see if the changes made earlier inside the first container, also reflect inside the new container.



We can observe in the above image that the volume securely saves the changes from the first container and mounts them onto the second container.

***Please Note:*** *Volumes can be shared between two or more containers simultaneously. The proof is attached below. We can see that the same volume(myVolume) is mounted on another container (volume-container-3).*



***Please note:*** *You can use two options to attach a Volume or Bind Mount to a container:*

* -v / — volume, or,
* --mount

The **--mount** option is more expressive than the **-v/ — volume** option.

### Creating and using Bind Mounts and Volumes in Docker

The general syntax for using both are:

**> Bind Mounts**

using *-v /--volume*

docker run -d \  
-v <path\_on\_host>:<path\_on\_container> <image\_name>

using *--mount*

docker run -d \  
--mount type=bind,source=<path\_on\_host >,target=< path\_on\_container >

**> Volumes**

docker run -d -v <volume\_name>:<path\_on\_container> <image\_name>.  
docker run -d \  
mount source=<volume\_name >,target=< path\_on\_container >

As you can see, I have successfully used a bind mount and a volume with my container. I hope this post helped you understand how it is created and used. To learn more on this topic, skim through Docker’s official documentation.

# **How to Use Docker for Your MySQL Database**

[Docker](https://www.docker.com/) is among the more popular platforms for developing and deploying containerized applications. Containers are isolated environments that hold an application along with all the software packages it needs. With Docker, you can run or scale your application in any environment.

[MySQL](https://www.mysql.com/) is one of the most popular SQL-compatible relational databases. Running MySQL inside a Docker container lets you separate your database from your code. You can also use a container orchestrator like Kubernetes to scale MySQL independently of your API server instance.

Using containers gives you the benefit of consistency. Once you’re done building your system, you can deploy your containers to the cloud without manually installing and configuring MySQL on bare-metal hardware.

## **Planning Your Deployment**

While using Docker with MySQL simplifies many aspects of your [deployment](https://earthly.dev/blog/deployment-strategies), such as installing the server and creating a database, it does come with some technical issues. The most significant is data storage: Docker is primarily designed around stateless containers, while a MySQL database is inherently stateful.

You need to use [Docker volumes](https://docs.docker.com/storage/volumes) when deploying a MySQL container. Volumes provide a mechanism to persist files after the container stops. You’ll lose your database if you restart a MySQL container that’s not using volumes.

Volumes store data outside of any single container. After your MySQL container stops, the files stored in its mounted volumes will remain accessible on your host. You can mount the volumes back into new containers, avoiding data loss after you replace your MySQL instance with a new image version.

## **Use Cases for MySQL in Docker**

**Dockerized MySQL works well in development and staging environments where you want to quickly bring up isolated database instances. It’s much quicker and easier to start a database in Docker than to configure a conventional MySQL installation in a full virtual machine.**

Although you could run MySQL locally on your host, this becomes limiting when you’re working on several applications simultaneously. Using containers offers complete separation of each system’s data and the ability to provide a unique MySQL server configuration for each one.

There are some scenarios where choosing to Dockerize your database might be less impactful. Demanding production environments might be better off with a dedicated MySQL server. Docker’s performance overheads are modest but can stack up in I/O-intensive workloads like those of a write-heavy database. A bare-metal production server also keeps your instance accessible to people in database maintenance roles who are unfamiliar with Docker.

Nonetheless, Docker is perfectly capable of supporting MySQL database deployments, from local development environments through to production. Using it for your whole cycle guarantees consistency. If your production instance uses the same Docker image as development, you can be sure your live systems will behave predictably. Here’s how to get a MySQL server running in a Docker container.

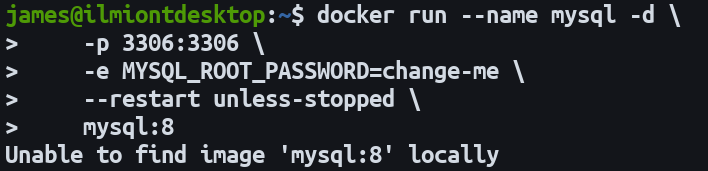
## **Starting Your MySQL Container**

MySQL has an official Docker image available [on Docker Hub](https://hub.docker.com/_/mysql). First identify the image tag you should use. MySQL versions 5.6, 5.7, and 8.0 are available.

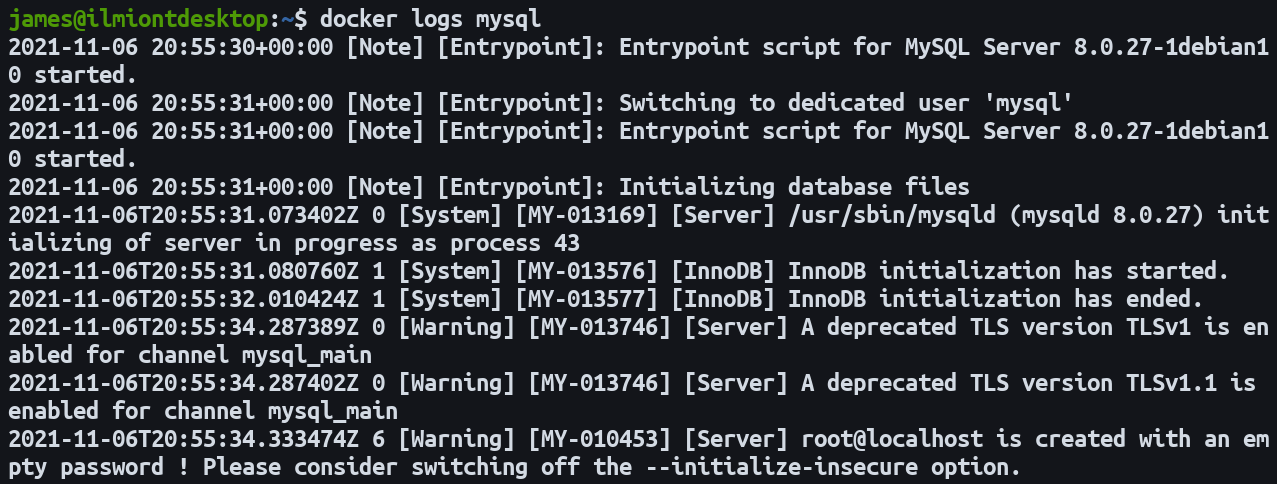
The **latest** tag points to the latest release, currently 8.0. Avoid using this tag, as it means you could unintentionally receive a major MySQL version upgrade in the future. Specifically referencing the version you want allows for a more controlled approach to updates.

Starting a MySQL container for the first time will automatically create an initial **root** user. You need to either supply a password for this user or ask MySQL to generate one. Here’s an example of running a basic MySQL container with a specified root password:

>\_docker run --name mysql -d **\**  
 **-p 3306:3306 \**  
 **-e MYSQL\_ROOT\_PASSWORD=change-me \**  
 **--restart unless-stopped \**  
 **mysql:8**

Starting a MySQL container and pulling the Docker image

This command starts a container with MySQL 8. The password for the **root** user is set manually. The **-d** flag means the container will run in the background until it’s stopped, independently of your terminal session. You can view the container’s startup logs with **docker logs mysql --follow**. When “ready for connections” appears, your MySQL database is accessible.

Early MySQL bootstrap logs

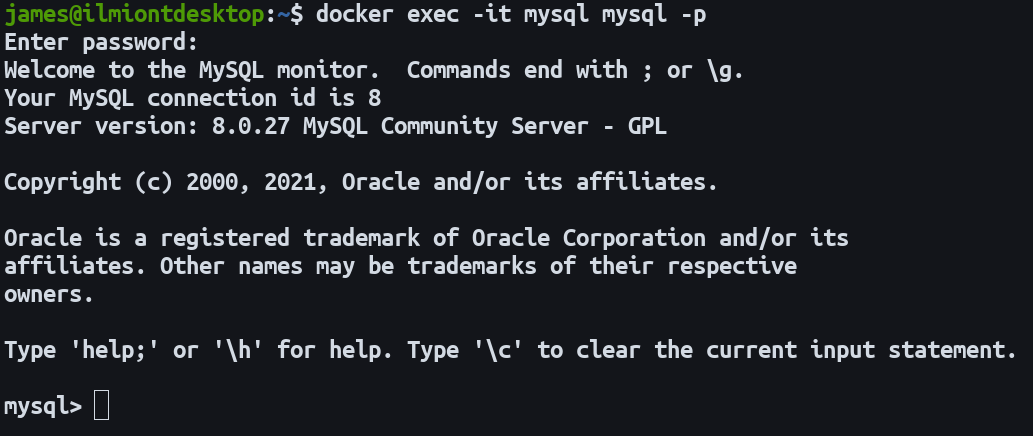
The **--restart** parameter instructs Docker to [always restart](https://docs.docker.com/config/containers/start-containers-automatically) the container. This means your MySQL database will run without intervention after host machine reboots or Docker daemon updates. The **unless-stopped** policy used here won’t start the container if you manually stopped it with **docker stop**.

Docker’s **-p** flag [enables port forwarding](https://docs.docker.com/config/containers/container-networking) into the container so you’ll be able to access your database on **localhost:3306**. This is the default MySQL port; this example forwards port 3306 on your host to the same port inside the container. Use your favorite MySQL client to connect over this port with **root** and your chosen password as user credentials.

Without port forwarding enabled, you’d only be able to access your database from within the container. You can do this at any time by using **docker exec** to get a shell inside the container:

>\_docker exec -it mysql mysql -p

This command runs **mysql -p** inside the **mysql** container. The **-it** flags mean your terminal’s input stream will be forwarded to the container as an interactive TTY.

Launching the MySQL shell inside a Docker container

### **Persisting Data With Volumes**

While the container created above is a fully functioning MySQL server, you need to set up volumes so your data isn’t lost when the container stops. The MySQL Docker image is configured to store all its data in the **/var/lib/mysql** directory. Mounting a volume to this directory will enable persistent data storage that outlives any single container instance.

Stop and remove your earlier container to avoid naming conflicts:

>\_docker stop mysql  
docker rm mysql

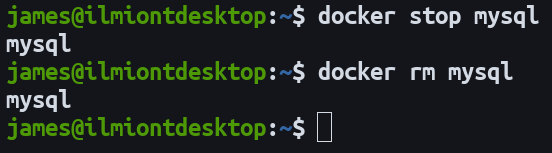
Then start a new container with the revised configuration:

>\_docker run --name mysql -d **\**  
 **-p 3306:3306 \**  
 **-e MYSQL\_ROOT\_PASSWORD=change-me \**  
 **-v mysql:/var/lib/mysql \**  
 **mysql:8**

Using this command to start your MySQL container will create a new Docker volume called **mysql**. It’ll be mounted into the container at **/var/lib/mysql**, where MySQL stores its data files. Any data written to this directory will now be transparently stored in the Docker-managed volume on your host.

Repeat the steps to stop and remove your container:

>\_docker stop mysql  
docker rm mysql

Stopping and removing a Docker container

Repeat the **docker run** command with the same arguments. As the **mysql** named volume will already exist, the new container will retain the data created by the old one. If you want to destroy the volume, use **docker volume rm mysql**.

### **Using Container Networks**

In the examples above, port forwarding was used to expose the MySQL server on your host’s network. If you’ll only be connecting to MySQL from within another Docker container, such as your API server, a better approach is to create a dedicated [Docker network](https://docs.docker.com/network). This improves security by limiting your database’s exposure.

First create a Docker network for your application:

>\_docker network create example-app

Specify this network when starting your MySQL container:

>\_docker run --name mysql -d **\**  
 **-e MYSQL\_ROOT\_PASSWORD=change-me \**  
 **-v mysql:/var/lib/mysql \**  
 **--network example-app \**  
 **mysql:8**

Connect another container to the same network:

>\_docker run --name api-server -d **\**  
 **-p 80:80 \**  
 **--network example-app \**  
 **example-api-server:latest**

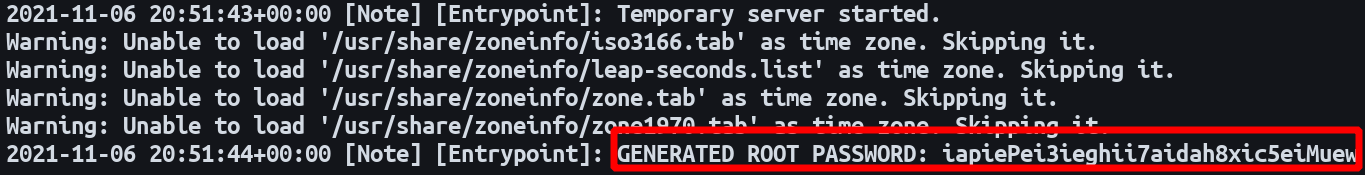
Your API and MySQL containers now share a network. You can connect to MySQL from your API container by referencing the MySQL container’s hostname. This matches the container’s name by default. Here your application should connect to port 3306 on the **mysql** host.

### **MySQL Configuration**

The official MySQL image supports several environment variables that you can use to configure your container’s initial state. You’ve already seen one, **MYSQL\_ROOT\_PASSWORD**. Use the **-e** flag with **docker run** to set each of these variables. They’re only respected the first time the container starts, when the MySQL data directory is empty.

* **MYSQL\_DATABASE** - The name of a database schema to be created when the container starts.
* **MYSQL\_USER and MYSQL\_PASSWORD** - Create a new ordinary user when the container starts.
* **MYSQL\_RANDOM\_ROOT\_PASSWORD** - Set this instead of **MYSQL\_ROOT\_PASSWORD** if you’d like MySQL to generate a secure **root** password for you. If you enable this setting, the password will be emitted to the container’s logs (accessible via the **docker logs** command) during the first start. It will not be possible to retrieve the password afterward.
* **MYSQL\_ALLOW\_EMPTY\_PASSWORD** - Setting this will create the **root** user with an empty password. Only use this option for throwaway database instances. It is insecure and would let anyone connect to MySQL with superuser privileges.

Using these environment variables means their values will be visible to anyone able to **docker inspect** your container. A more secure approach is to use [Docker secrets](https://earthly.dev/blog/docker-secrets) or volumes to inject values as files.

MySQL generating a random root password

The MySQL image supports an additional variant of each of the above variables. Suffix a variable’s name with **\_FILE** to have its value interpreted as a path to a file containing the real value. This example securely sets the **root** user’s password in a way that can’t be inspected from outside the container:

>\_mkdir secrets  
echo "P@$$w0rd" > secrets/mysql-root-password  
  
docker run --name mysql -d **\**  
 **-p 3306:3306 \**  
 **-e MYSQL\_ROOT\_PASSWORD\_FILE=/run/secrets/mysql-root-password \**  
 **-v ./secrets:/run/secrets \**  
 **--restart unless-stopped \**  
 **mysql:8**

The password is written to a file that’s mounted into the container using a Docker volume. MySQL instructs that the password be sourced from that mounted file by way of the **MYSQL\_ROOT\_PASSWORD\_FILE** environment variable. Anyone viewing the container’s environment variables will see the file path instead of the plain text password.

### **Creating a Custom Image**

It can be helpful to create your own Docker image if your app requires custom MySQL configuration. Adding extra layers atop the official MySQL base image gives you a ready-to-use image where you can omit manual injection of a MySQL config file.

Here’s an [example **my.cnf**](https://dev.mysql.com/doc/refman/8.0/en/option-files.html) that changes some MySQL settings:

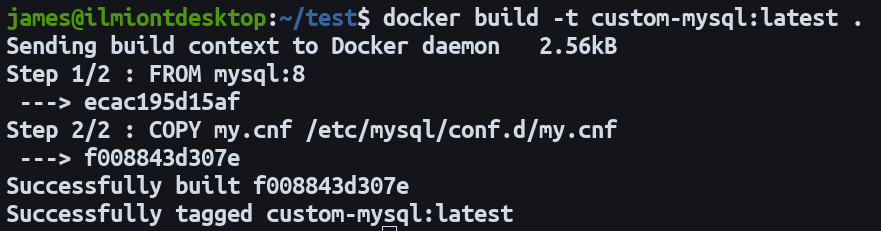
my.cnf**[mysqld]**  
**innodb-ft-enable-stopword** = 0  
innodb-ft-min-token-size = 1

The MySQL image loads config files stored in the **/etc/mysql/conf.d** directory. Files will only be read when the MySQL server starts, which is when you start your Docker container. To get your config into your container, either use another Docker volume to bind mount your file, or use a Dockerfile to bake your changes into a new image:

Dockerfile**FROM** mysql:8  
COPY my.cnf /etc/mysql/conf.d/my.cnf

Build your image:

>\_docker build -t custom-mysql:latest .

Building a custom MySQL Docker image

Now you can run your image to start a MySQL instance that automatically uses your config file:

>\_docker run --name custom-mysql -d **\**  
 **-p 3306:3306 \**  
 **-e MYSQL\_ROOT\_PASSWORD=change-me \**  
 **-v mysql:/var/lib/mysql \**  
 **custom-mysql:latest**

Since your custom image is based on the official Docker Hub version, you can use all the existing [environment variables](https://earthly.dev/blog/understanding-bash) described above.

## **Conclusion**

Running MySQL in a Docker container provides consistency and cross-environment isolation for your database deployments. You can either use the official MySQL image as-is or create a custom image.

Once you’re ready to move to production, you can reuse your development workflow to get your database live. Automate the process by launching your containers within your CI/CD pipeline, where tools such as [Earthly](https://docs.earthly.dev/docs/guides/docker-in-earthly) can offer repeatable builds and insights into any failures. Earthly offers on-demand Docker daemons and high reproducibility to help you automate your builds more quickly.