



The Flask Mega-Tutorial, Part XIX: Deployment on Docker Containers

Posted by [Miguel Grinberg](#) on [December 3, 2023](#) under [Flask](#) [Python](#)
[Programming](#) [Docker](#)

This is the nineteenth installment of the Flask Mega-Tutorial series, in which I'm going to deploy Microblog to the Docker container platform.

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from [Amazon](#). Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it [here](#).

For your reference, here is the complete list of articles in this series:

- [Chapter 1: Hello, World!](#)
- [Chapter 2: Templates](#)
- [Chapter 3: Web Forms](#)
- [Chapter 4: Database](#)
- [Chapter 5: User Logins](#)
- [Chapter 6: Profile Page and Avatars](#)
- [Chapter 7: Error Handling](#)
- [Chapter 8: Followers](#)
- [Chapter 9: Pagination](#)
- [Chapter 10: Email Support](#)
- [Chapter 11: Facelift](#)
- [Chapter 12: Dates and Times](#)
- [Chapter 13: I18n and L10n](#)
- [Chapter 14: Ajax](#)
- [Chapter 15: A Better Application Structure](#)
- [Chapter 16: Full-Text Search](#)
- [Chapter 17: Deployment on Linux](#)
- [Chapter 18: Deployment on Heroku](#)
- [Chapter 19: Deployment on Docker Containers \(this article\)](#)
- [Chapter 20: Some JavaScript Magic](#)
- [Chapter 21: User Notifications](#)

- [Chapter 22: Background Jobs](#)
- [Chapter 23: Application Programming Interfaces \(APIs\)](#)

In [Chapter 17](#) you learned about traditional deployments, in which you have to take care of every little aspect of the server configuration. Then in [Chapter 18](#) I took you to the other extreme when I introduced you to Heroku, a service that takes complete control of the configuration and deployment tasks, allowing you to fully concentrate on your application. In this chapter you are going to learn about a third application deployment strategy based on *containers*, more particularly on the [Docker](#) container platform. This third option sits somewhere in between the other two in terms of the amount of deployment work needed on your part.

Containers are built on a lightweight virtualization technology that allows an application, along with its dependencies and configuration to run in complete isolation, but without the need to use a full-blown virtualization solution such as virtual machines, which need a lot more resources and can sometimes have a significant performance degradation in comparison to the host. A system configured as a container host can execute many containers, all of them sharing the host's kernel and direct access to the host's hardware. This is in contrast to virtual machines, which have to emulate a complete system, including CPU, disk, other hardware, kernel, etc.

In spite of having to share the kernel, the level of isolation in a container is pretty high. A container has its own file system, and can be based on an operating system that is different from the one used by the container host. For example, you can run containers based on Ubuntu Linux on a Fedora host, or vice versa. While containers are a technology that is native to the Linux operating system, thanks to virtualization it is also possible to run Linux containers on Windows and macOS hosts. This allows you to test your deployments on your development system, and also incorporate containers in your development workflow if you wish to do so.

The GitHub links for this chapter are: [Browse](#), [Zip](#), [Diff](#).

Installing Docker

While Docker isn't the only container platform, it is by far the most popular, so that's going to be my choice.

To work with Docker, you first have to install it on your system. There are installers for Windows, macOS and several Linux distributions available at the [Docker website](#). By far the easiest way to get Docker set up on your computer is to use the Docker Desktop installer for your operating system. If you are working on a Microsoft Windows system, it is important to note that Docker requires Hyper-V. The installer will enable this for you if necessary, but keep in mind that enabling Hyper-V may prevent other virtualization technologies such as VirtualBox from working.

Once Docker Desktop is installed on your system, you can verify that the installation was successful by typing the following command on a terminal window or command prompt:

```
$ docker version
Client: Docker Engine - Community
Version:           23.0.5
API version:       1.42
Go version:        go1.19.8
Git commit:        bc4487a
Built:             Wed Apr 26 16:17:14 2023
OS/Arch:           darwin/amd64
Context:           default

Server: Docker Engine - Community
Engine:
Version:           23.0.5
API version:       1.42 (minimum version 1.12)
Go version:        go1.19.8
Git commit:        94d3ad6
Built:             Wed Apr 26 16:17:14 2023
OS/Arch:           darwin/amd64
Experimental:      false
containerd:
Version:           1.6.20
GitCommit:         2806fc1057397dbaefbea0e4e17bddfbd388f38
runc:
Version:           1.1.5
GitCommit:         v1.1.5-0-gf19387a
docker-init:
Version:           0.19.0
GitCommit:         de40ad0
```

Building a Container Image

The first step in creating a container for Microblog is to build an *image* for it. A container image is a template that is used to create a container. It contains a complete representation of the container file system, along with various settings pertaining to networking, start up options, etc.

The most basic way to create a container image for your application is to start a container for the base operating system you want to use (Ubuntu, Fedora, etc.), connect to a bash shell process running in it, and then manually install your application, maybe following the guidelines I presented in [Chapter 17](#) for a traditional deployment. After you install everything, you can take a snapshot of the container and that becomes the image. This type of workflow is supported with the `docker` command, but I'm not going to discuss it because it is not convenient to have to manually install the application every time you need to generate a new image.

A better approach is to generate the container image through a script. The command that creates scripted container images is `docker build`. This command reads and executes build instructions from a file called *Dockerfile*, which I will need to create. The Dockerfile is basically an installer script that executes the installation steps to get the application deployed, plus some container specific settings.

Here is a basic *Dockerfile* for Microblog:

Dockerfile: Dockerfile for Microblog.

```
FROM python:slim

COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
RUN pip install gunicorn

COPY app app
COPY migrations migrations
COPY microblog.py config.py boot.sh ./
RUN chmod a+x boot.sh

ENV FLASK_APP microblog.py
RUN flask translate compile

EXPOSE 5000
ENTRYPOINT ["./boot.sh"]
```

Each line in the Dockerfile is a command. The `FROM` command specifies the base container image on which the new image will be built. The idea is that you start from an existing image, add or change some things, and you end up with a derived image. Images are referenced by a name and a tag, separated by a colon. The tag is used as a versioning mechanism, allowing a container image to provide more than one variant. The name of my chosen image is `python`, which is the official Docker image for Python. The tags for this image allow you to specify the interpreter version and base operating system. The `slim` tag selects a container image that has only the minimal packages required to run the Python interpreter. You can see what other tags are available for Python in the [Python image repository](#).

Depending on when you are working on your Dockerfile, you may find that some packages fail to install on the latest version of Python used by the `slim` tag, which happens because when Python releases a new version many packages take some time to release binary packages for it. If you notice problems when you build your container image, you can try to use an older version of Python, which can be done by adding the desired version as a prefix in the tag. For example, the `3.11-slim` tag installs Python 3.11.

The `COPY` command transfers files from your machine to the container's file system. This command takes two or more arguments, the source and destination files or directories. The source file(s) must be relative to the directory where the Dockerfile is located. The destination can be an absolute path, or a path that is relative to the current directory, which by default is the root of the container file system. In this first `COPY` command, I'm copying the *requirements.txt* file to the root directory in the container file system.

Now that I have the *requirements.txt* file in the container, I can install all the requirements in it. Because the requirements file contains only generic dependencies, I then explicitly install `unicorn`, which I'm going to use as a web server. Alternatively, I could have added the package to my *requirements.txt* file.

The three `COPY` commands that follow install the application in the container, by copying the *app* package, the *migrations* directory with the database migrations, and the *microblog.py* and *config.py* scripts from

the top-level directory. I'm also copying a new file, *boot.sh* that I will discuss below.

The `RUN chmod` command ensures that this new *boot.sh* file is correctly set as an executable file. If you are in a Unix based file system and your source file is already marked as executable, then the copied file will also have the executable bit set. I decided to set the executable bit explicitly to allow the application to work on Microsoft Windows, which has no concept of executable bits. If you are working on macOS or Linux you probably don't need this statement, but it does not hurt to have it anyway.

The `ENV` command sets an environment variable inside the container. I need to set `FLASK_APP`, which is required to use the `flask` command.

The next `RUN` statement compiles the translations. This is a `flask` sub-command, so I had to first set `FLASK_APP` in the environment, or else the command would not be able to find the application instance, and my custom sub-commands.

The `EXPOSE` command configures the port that this container will be using for its server. This is necessary so that Docker can configure the network in the container appropriately. I've chosen the standard Flask port 5000, but this can be any port.

Finally, the `ENTRYPOINT` statement defines the default command that should be executed when a container is started with this image. This is the command that will start the application web server. To keep things well organized, I decided to create a separate script for this, and this is the *boot.sh* file that I copied to the container earlier. Here are the contents of this script:

boot.sh: Docker container start-up script.

```
#!/bin/bash
flask db upgrade
exec gunicorn -b :5000 --access-logfile - --error-logfile - microblog:
```

This is a fairly standard start up script that is similar to how the deployments in [Chapter 17](#) and [Chapter 18](#) were started. I upgrade the database through the migration framework, and run the server with Gunicorn.

Note the `exec` that precedes the Unicorn command. In a shell script, `exec` triggers the process running the script (the `bash` interpreter in this case) to be replaced with the command given, instead of starting the command as a sub-processes. This is important, because Docker associates the life of the container to the first process that runs on it. In cases like this one, where the start-up process is not the main process of the container, you need to make sure that the main process takes the place of that first process to ensure that the container is not terminated early by Docker.

If you are creating the `boot.sh` file on Windows, make sure that you select UNIX line-endings in your text editor. Since this file is going to execute inside a container running Linux, it must have the correct line endings for this operating system.

An interesting aspect of Docker is that anything that the container writes to `stdout` or `stderr` will be captured and stored as logs for the container. For that reason, the `--access-logfile` and `--error-logfile` are both configured with a `-`, which sends the log to standard output so that they are stored as logs by Docker.

With the Dockerfile created, I can now build a container image:

```
$ docker build -t microblog:latest .
```

The `-t` argument that I'm giving to the `docker build` command sets the name and tag for the new container image. The `.` indicates the base directory where the container is to be built. This is the directory where the `Dockerfile` is located. The build process is going to evaluate all the commands in the `Dockerfile` and create the image, which will be stored on your own machine.

You can obtain a list of the images that you have locally with the `docker images` command:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	slim	d7971c18b18e	7 months ago	132MB
microblog	latest	03978d7e1007	27 seconds ago	259MB

This listing will include your new image, and also the base image on which it was built. Any time you make changes to the application, you can update the container image by running the build command again.

Starting a Container

With an image already created, you can now run the container version of the application. This is done with the `docker run` command, which takes a long list of arguments. I'm going to start by showing you a basic example:

```
$ docker run --name microblog -d -p 8000:5000 --rm microblog:latest
021da2e1e0d390320248abf97dfbbe7b27c70fefed113d5a41bb67a68522e91c
```

The `--name` option provides a name for the new container. The `-d` option tells Docker to run the container in the background. Without `-d` the container runs as a foreground application, blocking your command prompt. The `-p` option maps container ports to host ports. The port number on the left is the port on the host computer, and the one on the right is the port inside the container. The above example exposes port 5000 in the container on port 8000 in the host, so you will access the application on 8000, even though internally the container is using 5000. I'm using different port numbers to demonstrate the flexibility of mapping ports, but if you prefer, you can use the same port number on the host and the container. The `--rm` option will delete the container once it is terminated. While this isn't required, containers that finish or are interrupted are usually not needed anymore, so they can be automatically deleted. The last argument is the container image name and tag to use for the container. After you run the above command, you can access the application at *<http://localhost:8000>*.

The output of `docker run` is the ID assigned to the new container. This is a long hexadecimal string, that you can use whenever you need to refer to the container in subsequent commands. In fact, only the first few characters are necessary, enough to make the ID unique.

If you want to see what containers are running, you can use the `docker ps` command:

```
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  PORTS              N
```



```
021da2e1e0d3  microblog:latest  "./boot.sh"  0.0.0.0:8000->5000/tcp  m
```

You can see that even the `docker ps` command shortens container IDs. If you now want to stop the container, you can use `docker stop` and pass either the container ID or the name that was given to it with the `--name` option:

```
$ docker stop microblog
microblog
```

If you recall, there are a number of options in the application's configuration that are sourced from environment variables. For example, the Flask secret key, database URL and email server options are all imported from environment variables. In the `docker run` example above I have not worried about those, so all those configuration options are going to use defaults.

In a more realistic example, you will be setting those environment variables inside the container. You saw in the previous section that the `ENV` command in the *Dockerfile* sets environment variables, and it is a handy option for variables that are going to be static. For variables that depend on the installation, however, it isn't convenient to have them as part of the build process, because you want to have a container image that is fairly portable. If you want to give your application to another person as a container image, you would want that person to be able to use it as is, and not have to rebuild it with different variables.

So build-time environment variables can be useful, but there is also a need to have run-time environment variables that can be set via the `docker run` command, and for these variables, the `-e` option can be used. The following example sets a secret key and sends email through a Gmail account:

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-se
-e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS
-e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmai
microblog:latest
```

It is not uncommon for `docker run` command lines to be extremely long due to having many environment variable definitions.

Using Third-Party "Containerized" Services

The container version of Microblog is looking good, but I haven't really thought much about storage yet. In fact, since I haven't set a `DATABASE_URL` environment variable, the application is using the default SQLite database, which is supported by a file on disk. What do you think is going to happen to that SQLite file when you stop and delete the container? The file is going to disappear!

The file system in a container is *ephemeral*, meaning that it goes away when the container goes away. You can write data to the file system, and the data is going to be there if the container needs to read it, but if for any reason you need to recycle your container and replace it with a new one, any data that the application saved to disk is going to be lost forever.

A good design strategy for a container application is to make the application containers *stateless*. If you have a container that has application code and no data, you can throw it away and replace it with a new one without any problems. With this usage the container becomes truly disposable, which is great in terms of simplifying the deployment of upgrades.

But of course, this means that the data must be put somewhere outside the application container. This is where the fantastic Docker ecosystem comes into play. The Docker Container Registry contains a large variety of container images. I have already told you about the Python container image, which I'm using as a base image for my Microblog container. In addition to that, Docker maintains images for many other languages, databases and other services in the Docker registry and if that isn't enough, the registry also allows companies to publish container images for their products, and also regular users like you or me to publish your own images. That means that the effort to install third party services is reduced to finding an appropriate image in the registry, and starting it with a `docker run` command with proper arguments.

So what I'm going to do now is create two additional containers, one for a MySQL database, and another one for the Elasticsearch service, and then I'm going to make the command line that starts the Microblog container even longer with options that enable it to communicate with these two new containers.

Adding a MySQL Container

In this section you are going to start a MySQL container, and then connect it to the Microblog container. The recommended way to run a collection of linked containers is to create a *network* for them. A Docker network is a virtual construct that allows all the containers that are added to it to see each other. Create a new Docker network with the following command:

```
$ docker network create microblog-network
```

Like many other products and services, MySQL has public container images available on the Docker registry. Like my own Microblog container, MySQL relies on environment variables that need to be passed to `docker run`. These configure passwords, database names etc. While there are many MySQL images in the registry, I decided to use one that is officially maintained by the MySQL team. You can find detailed information about the MySQL container image in its registry page: <https://hub.docker.com/r/mysql/mysql-server/>.

If you remember the laborious process to set up MySQL in [Chapter 17](#), you are going to appreciate Docker when you see how easy it is in comparison. Here is the `docker run` command that starts a MySQL server:

```
$ docker run --name mysql -d -e MYSQL_RANDOM_ROOT_PASSWORD=yes \
  -e MYSQL_DATABASE=microblog -e MYSQL_USER=microblog \
  -e MYSQL_PASSWORD=<database-password> \
  --network microblog-network \
  mysql:latest
```

That is it! On any machine that you have Docker installed, you can run the above command, and you'll get a fully installed MySQL server with a randomly generated root password, a brand-new database called `microblog`, and a user with the same name that is configured with full permissions to access the database. Note that you will need to enter a proper password as the value for the `MYSQL_PASSWORD` environment variable.

The `--network` option above tells Docker to put the MySQL container on the network created above.

Now on the application side, I need to use a MySQL client package, like I did for the traditional deployment on Ubuntu. I'm going to use `pymysql` once again, which I can add to the *Dockerfile*, along with the `cryptography` package that it uses for authentication against the MySQL server:

Dockerfile: Add pymysql and cryptography to Dockerfile.

```
# ...  
RUN pip install gunicorn pymysql cryptography  
# ...
```

Any time a change is made to the application or the *Dockerfile*, the container image needs to be rebuilt:

```
$ docker build -t microblog:latest .
```

Any now I can start Microblog again, but this time with a link to the database container so that both can communicate through the network:

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-se  
-e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS  
-e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-  
--network microblog-network \br/>-e DATABASE_URL=mysql+pymysql://microblog:<database-password>@mysql  
microblog:latest
```

The `--network` option tells Docker to include this container in the same network as the MySQL container from above. The containers that are in a network can reference each other using their names as the hostnames. For that reason, the `DATABASE_URL` uses `mysql` as the database hostname, as this is the name that I gave the MySQL container. Make sure you enter the correct database password that you selected for the MySQL container in the command above.

One thing I noticed when I was experimenting with the MySQL container is that it takes the MySQL container a few seconds to be fully running and ready to accept database connections. If you start the MySQL container and then start the application container immediately after, when the *boot.sh* script tries to run `flask db upgrade` it may fail due to the database not being ready to accept connections. To make my solution more robust, I decided to add a retry loop in *boot.sh*:

boot.sh: Retry database connection.

```
#!/bin/bash
while true; do
    flask db upgrade
    if [[ "$?" == "0" ]]; then
        break
    fi
    echo Upgrade command failed, retrying in 5 secs...
    sleep 5
done
exec gunicorn -b :5000 --access-logfile - --error-logfile - microblog:
```

This loop checks the exit code of the `flask db upgrade` command, and if it is non-zero it assumes that something went wrong, so it waits five seconds and then retries.

Adding an Elasticsearch Container

The [Elasticsearch documentation for Docker](#) shows how to run the service as a single-node for development, and as a two-node production-ready deployment. For now, I'm going to go with the single-node option, without encryption or passwords. The container is started with the following command:

```
$ docker run --name elasticsearch -d --rm -p 9200:9200 \
    -e discovery.type=single-node -e xpack.security.enabled=false \
    --network microblog-network \
    -t docker.elastic.co/elasticsearch/elasticsearch:8.11.1
```

So now that I have the Elasticsearch service up and running, I can modify the start command for my Microblog container to create a link to it and set the Elasticsearch service URL:

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-se
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TL
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmai
    --network microblog-network \
    -e DATABASE_URL=mysql+pymysql://microblog:<database-password>@mysq
    -e ELASTICSEARCH_URL=http://elasticsearch:9200 \
    microblog:latest
```

Before you run this command, remember to stop your previous Microblog container if you still have it running. Also be careful in setting

the correct passwords in the proper places in the command.

Now you should be able to visit *http://localhost:8000* and use the search feature. If you experience any errors, you can troubleshoot them by looking at the container logs. You'll most likely want to see logs for the Microblog container, where any Python stack traces will appear:

```
$ docker logs microblog
```

The Docker Container Registry

So now I have the complete application up and running on Docker, using three containers, two of which came from publicly available third-party images. If you would like to make your own container images available to others, then you have to *push* them to the Docker registry from where anybody can obtain images.


To have access to the Docker registry you need to go to *https://hub.docker.com* and create an account for yourself. Make sure you pick a username that you like, because that is going to be used in all the images that you publish.

To be able to access your account from the command line, you need to log in with the `docker login` command:

```
$ docker login
```

If you've been following my instructions, you now have an image called `microblog:latest` stored locally on your computer. To be able to push this image to the Docker registry, it needs to be renamed to include the account that owns it. This is done with the `docker tag` command:

```
$ docker tag microblog:latest <your-docker-registry-account>/microblog
```



If you list your images again with `docker images` you are now going to see two entries for Microblog, the original one with the `microblog:latest` name, and a new one that also includes your account name. These are really two aliases for the same image.

To publish your image to the Docker registry, use the `docker push` command:

```
$ docker push <your-docker-registry-account>/microblog:latest
```

Now your image is publicly available, and you can document how to install it and run from the Docker registry in the same way MySQL and others do.

Deployment of Containerized Applications

One of the best things about having your application running in Docker containers is that once you have the containers tested locally, you can take them to any platform that offers Docker support. For example, you could use the same servers I recommended in [Chapter 17](#) from Digital Ocean, Linode or Amazon Lightsail. Even the cheapest offering from these providers is sufficient to run Docker with a handful of containers.

The [Amazon Container Service \(ECS\)](#) gives you the ability to create a cluster of container hosts on which to run your containers, in a fully integrated AWS environment, with support for scaling and load balancing, plus the option to use a private container registry for your container images.

Finally, a container orchestration platform such as [Kubernetes](#) provides an even greater level of automation and convenience, by allowing you to describe your multi-container deployments in text files in YAML format, with load balancing, scaling, secure management of secrets and rolling upgrades and rollbacks.

Continue on to the [next chapter](#).

Buy me a coffee?

Thank you for visiting my blog! If you enjoyed this article, please consider supporting my work and keeping me caffeinated with a small one-time donation through [Buy me a coffee](#). Thanks!



Share this post

[Hacker News](#)[Reddit](#)[Twitter](#)[LinkedIn](#)[Facebook](#)[E-Mail](#)

20 comments



#1 **Dave** said 2 years ago

Could you use SQLite in docker if you wanted to? Any idea how do do that with volumes?

Ive read loads of reasons on stackoverflow why its a bad idea. But I find them really convenient to work with and better for my use cases.



#2 **Miguel Grinberg** said 2 years ago

@Dave: it wouldn't be my choice, but you can use SQLite in a Docker container. The only difference is that the file system is ephemeral, if you want your database to persist beyond the container, you have to create a volume, attach it to the container, and then write the database on the volume.



#3 **Grant** said 2 years ago

Awesome tutorial! Very comprehensive and well explained.

This seems like a bit of a silly question, and I may have overlooked something. But how does the data persist in the MySQL container? If the Docker container file systems are ephemeral, won't the MySQL database lose all its data if it shuts down? I don't see a volume being created, but it is the official MySQL image so I figure this is being done automatically as the image makes no sense without persistent storage.



#4 **Miguel Grinberg** said 2 years ago

@Grant: By default the MySQL container creates a volume that is separate from the ephemeral file system of the container. See

the section "Where to Store Data" in the [MySQL docker container documentation](#) to learn about this and other strategies to keep the data safe.



#5 **R Kovacs** said 2 years ago

Wow, this has been incredible. It was a tough challenge working through the entire tutorial up to this point, but well worth it. I recommend it to others. It has identified so many libraries that I now need to explore. Having a working starting point for expansion is a great way to learn. I do have the three docker images (mysql, elasticsearch, rvzk/microblog) working on my Raspberry Pi 5 8GB, and even the search works fine. It may be obvious, but just wanted to point out to others than nginx needs to be running with the docker images to expose the http/https ports. I haven't tested the translation service in the Docker image yet. Now on to 4 more chapters. Can't wait to learn more about APIs. Thanks, the effort you expended putting this together is greatly appreciated.



#6 **Miguel Grinberg** said 2 years ago

@R: Using nginx is optional, not required. The set up that I show in this article leaves everything deployed and ready to be used without nginx using http:// on port 8000. You can add nginx to configure https://, for example, as I show in my [blog post](#).



#7 **Walter** said 2 years ago

Thank you for the continued extension and updates of your work. I've learnt a lot from it.

While deploying a project with a docker container like you described the [booth.sh](#) gave me an error that took me some time to figure out. The error was: bash: ./boot.sh: cannot execute: required file not found.

I found the solution in this post:

<https://unix.stackexchange.com/questions/721844/linux-bash-shell-script-error-cannot-execute-required-file-not-found>

The problem was I edited the [boot.sh](#) script for my project to take out the babel command on a windows machine and the line

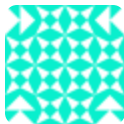
endings were changed. I followed the suggestion to use Notepad++ and change the line endings to Unix. This solved the problem and Docker deployment is now a breeze.



#8 Miguel Grinberg said 2 years ago

@Walter: maybe you missed it, but I do mention the requirement of using UNIX line-endings for this file:

If you are creating the [boot.sh](#) file on Windows, make sure that you select UNIX line-endings in your text editor. Since this file is going to execute inside a container running Linux, it must have the correct line endings for this operating system.



#9 Michael Matveev said a year ago

Thank you Miguel for your great work about flask. It helped me a lot, also your book.

In my application I have a lot of credentials in .env, later it will be deployed on EC2. But generally how to deal with .env files? starting docker run with 20 Variables?

Could you please recommend a best practice for that?

thank you.

Mischa



#10 Miguel Grinberg said a year ago

@Michael: Docker can read your .env file, both when using `docker run` and when using `docker compose`.



#11 Arman said a year ago

Hello sir. Thank you for this tutorial.

May i ask, where do these images go after we build them? Is there a way to find and locate them on our local machine? so we can transfer them offline from our machine to another machine? I would appreciate any thoughts on this.

(my OS is windows 11 by the way)

Thank you again.



#12 Miguel Grinberg said a year ago

@Arman: you can export an image to a file using the `docker save` command. Then on the other machine you can import the file with `docker load`. Another option that you have is to upload the image to the cloud, for example to Docker Hub, which you can do with `docker push`. To download an image from Docker Hub or other image registries you can use `docker pull`. Hope this helps!



#13 Finlay said a year ago

Thank you for the tutorial.

When I build an image, and then check the images by typing `docker images` it only shows my the image for my app, not the base image.

When I try to start a MYSQL server, I get this error:

Unable to find image 'mysql:latest' locally

docker: Error response from daemon: Head "<https://registry-1.docker.io/v2/library/mysql/manifests/latest>": unauthorized: incorrect username or password.

do you know where I might be going wrong?



#14 Miguel Grinberg said a year ago

@Finlay: the error indicates that your DockerHub login is incorrect.



#15 Tychus said a year ago

Hello Miguel. How could I use the SMTP debugging server when running a Container? I want to output the email to the terminal.

#16 Miguel Grinberg said a year ago



@Tychus: It works in the same way when the application or the SMTP debugging server are in containers. You just need to make sure that the IP address for the server is correctly configured in the application. You can't use localhost, you need to use the correct IP address for where the email server is running.



#17 **Yoma** said 10 months ago

Hello Miguel, what happens when

1. I stop and restart any of the containers, say mysql and elasticsearch? Do we lose the data stored in mysql db or the indexes stored in elastic search?
2. We need to restart our PC? I am assuming the containers crash?

Finally if we want to stop the containers in a network, is there a way to do them using a single command or do we have to stop each container individually?



#18 **Miguel Grinberg** said 10 months ago

@Yoma: For your first question, it really depends on the container. Both MySQL and Elasticsearch provide options to select a data directory that is hosted on your own machine outside of the Docker image, so that when the container dies or is restarted the data remains available and can be passed on to a new container. The documentation for the container image you want to use will normally show how to set this up. For the 2nd question, if you restart your PC the containers that are running will obviously stop, in the same way any applications or processes stop. When the machine is up again you will need to start containers. Docker also provides a "restart" option that you can set when you run a container to configure automatic restarts.



#19 **Ryan Golhar** said 8 months ago

Hi Miguel - I've deployed this app for AWS ElasticBeanStalk. This would be another great example/chapter for your blog. I'm happy to contribute to it if you are interested.



#20 Miguel Grinberg said 8 months ago

@Ryan: I'm not planning to make updates to this series right now, but you are most welcome to blog about your deployment process yourself.

««

«

»


»»

Leave a Comment

Name

Email

Comment



Captcha



I'm not a robot

reCAPTCHA
[Privacy](#) - [Terms](#)

Submit

The Flask Mega-Tutorial

New 2024 Edition!



If you would you like to support my work on my [Flask Mega-Tutorial series](#) on this blog and as a reward have access to the complete tutorial nicely structured as a book and/or a set of videos, you can now order it from my [Courses](#) site or from [Amazon](#).

[Click here to get the Book!](#)

[Click here to get the Video Course!](#)

About Miguel

Welcome to my blog!





I'm a software engineer and technical writer, currently living in Drogheda, Ireland.































You can also find me on [Github](#), [LinkedIn](#), [Bluesky](#), [Mastodon](#), [Twitter](#), [YouTube](#), and [Patreon](#).

Thank you for visiting!

Categories

-  [AI](#) 3
-  [Arduino](#) 7
-  [Authentication](#) 10
-  [Blog](#) 1
-  [C++](#) 5
-  [CSS](#) 1

	Cloud	11
	Database	23
	Docker	5
	Filmmaking	6
	Flask	129
	Games	1
	IoT	8
	JavaScript	36
	MicroPython	10
	Microdot	1
	Microservices	2
	Movie Reviews	5
	Personal	3
	Photography	7
	Product Reviews	2
	Programming	193
	Project Management	1
	Python	174
	REST	7
	Raspberry Pi	8
	React	19
	Reviews	1
	Robotics	6
	Security	12
	Video	22
	WebSocket	2
	Webcast	3
	Windows	1