- acend gmbh

Setup

Docker architecture

- Docker is a client-server application
- The **Docker daemon** (or “Engine”)
  - Receives and processes incoming Docker API requests
- The **Docker client**
  - Talks to the Docker daemon via the Docker API
  - We’ll use mostly the CLI embedded within the Docker binary
- The **Docker Hub** Registry
  - Is a collection of public images
  - The Docker daemon talks to it via the registry API

The Docker installation will install the Docker daemon and client on your workstation.

Setup introduction

This training depends on an installation of Docker. Follow the instructions on the subsequent pages to complete the setup on your platform of choice.

Installation for Windows

Installation for Windows

Please follow the instructions on Docker’s official documentation to install Docker CE for Windows.

When asked to use Windows container, choose *NOT* to.

> **Note**
> You don’t have to register for a Docker Cloud account.

Shell recommendation for Windows

We highly recommend to use the Bash emulation *Git Bash* from Git for Windows to do the exercises in this training.

Proxy configuration for Windows

If your organization has a proxy in place you have to set the proxy environment variables in order to be able to do `docker pull` or `docker push` .

Git Bash:

```
export HTTP_PROXY="http://<username>:<password>@<proxy>:<port>"
export HTTPS_PROXY="http://<username>:<password>@<proxy>:<port>"
```

> **Note**
> If you have special characters in your password, you have to encode them according to Percent-encoding reserved characters .

See also setting the proxy environment variables on Windows for alternative instructions on setting proxy environment variables.

Next steps

When you’re ready to go, head on over to the labs and begin with the training!

Installation for Mac

Installation for Mac

Please follow the instructions on Docker’s official documentation to install Docker CE for Mac.

> **Note**
> You don’t have to register for a Docker Cloud account.

Proxy configuration for Mac

If your organization has a proxy in place you have to set the proxy environment variables in order to be able to do `docker pull` or `docker push` .

```
export http_proxy="http://<username>:<password>@<proxy>:<port>"
export https_proxy="http://<username>:<password>@<proxy>:<port>"
```

> **Note**
> If you have special characters in your password, you have to encode them according to Percent-encoding reserved characters .

See also setting the proxy environment variables on Mac for alternative instructions on setting proxy environment variables.

Next steps

When you’re ready to go, head on over to the labs and begin with the training!

- acend gmbh

Installation for Linux

Installation for Linux

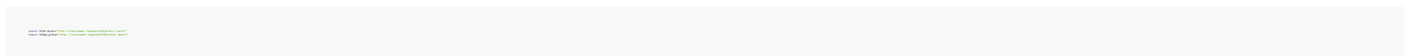
Please follow the instructions for your appropriate distribution to install Docker. The recommended way of installing is using the repository, except if you already know you’re going to remove the package again soon.

- Ubuntu
- Fedora
- Debian
- CentOS

Unrelated to what distribution you use, also have a look at the Post-installation steps for Linux . Please note however that these are optional steps and some are quite advanced, so going with the default might be the most appropriate way to go.

Proxy configuration for Linux

If your organization has a proxy in place you have to set the proxy environment variables in order to be able to do `docker pull` or `docker push` .



```
export http_proxy="http://<username>:<password>@<proxy>:<port>"
export https_proxy="http://<username>:<password>@<proxy>:<port>"
```

Note

If you have special characters in your password, you have to encode them according to Percent-encoding reserved characters .

Next steps

When you’re ready to go, head on over to the labs and begin with the training!

Try Docker without installation

Try Docker without installation

The page https://training.play-with-docker.com offers additional tutorials which also come with an interactive shell. The disadvantage is that you have to create an account, but if you don’t want to install Docker locally, this is a great way to do the exercises in this training using a browser-based Docker shell.

To do this lab with *Play with Docker*:

- Go to https://labs.play-with-docker.com
- Click on *Login*
- Enter your Docker login or register first
- Click *ADD NEW INSTANCE* and you are ready to do this training

Next steps

When you’re ready to go, head on over to the labs and begin with the training!

Labs

In this training, you’re going to learn the basics behind the container technology Docker.

- Introductory presentation: What are containers?
- Install Docker on your computer
- Run Docker containers
- Mount local volumes into containers
- Build Docker containers
- How networking works

Additional docs

- Official Docker Docs

1. Getting started

The command line tool

With Docker installed and working, now’s the time to become familiar with the command line utility. Using Docker consists of passing at least one command. `docker --help` shows the available options:

```
docker --help
```

```
Usage: docker COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default "/home/user/.docker")
  -D, --debug              Enable debug mode
      --help               Print usage
  -H, --host list          Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default "/home/user/.docker/ca.pem")
      --tlscert string     Path to TLS certificate file (default "/home/user/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default "/home/user/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit

Management Commands:
  config      Manage Docker configs
  container   Manage containers
  image       Manage images
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
  system      Manage Docker
  volume      Manage volumes

Commands:
  attach      Attach local standard input, output, and error streams to a running container
  build       Build an image from a Dockerfile
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  events      Get real time events from the server
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  history     Show the history of an image
  images      List images
  import      Import the contents from a tarball to create a filesystem image
  info        Display system-wide information
  inspect     Return low-level information on Docker objects
  kill        Kill one or more running containers
  load        Load an image from a tar archive or STDIN
  login       Log in to a Docker registry
  logout      Log out from a Docker registry
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  ps          List containers
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  run         Run a command in a new container
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  search      Search the Docker Hub for images
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  version     Show the Docker version information
  wait        Block until one or more containers stop, then print their exit codes

Run 'docker COMMAND --help' for more information on a command.
```

To view the switches available to a specific command, type:

```
docker <command> --help
```

To view system-wide information about Docker, use:

```
docker info
```

Hello world (with Docker images)

Docker containers are run from Docker images. By default, they pull these images from Docker Hub, a Docker registry managed by Docker Inc, the company behind the Docker project. Anybody can build and host their Docker images on Docker Hub, so for many applications and Linux distributions you'll find Docker images that are hosted on Docker Hub.

To check whether you can access and download images from Docker Hub, type:

```
docker run hello-world
```

The output, which should include the following, indicates that Docker appears to be working correctly:

```
Hello from Docker.
This message shows that your installation appears to be working correctly.
...
```

Your first container 😃

With this command, we just ran our first container on our computers. It ran a simple process that printed a message to standard out, the container itself is not very useful though.

Getting familiar with the Docker docs

Browse https://docs.docker.com and get familiar with the docs as well as the references. In this training, we're going to use Docker CE, so this is the docs section you might want to check out as well.

No! There are tons of images provided by companies, open source projects, and individuals. You can search for these images in various registries, some of the better-known are Docker Hub and Quay . Checkout the next lab for more details.

2. Images

Docker images

You can search for images available on Docker Hub by clicking the **Explore** link or by typing `mariadb` into the search field: https://hub.docker.com/search/?q=mariadb&type=image

You will get a list of results and the first hit will probably be the official image: https://hub.docker.com/_/mariadb

This page contains instructions on how to pull the image. Let's do that:

```
docker pull mariadb
```

Note

Care about security! Check the images before you run them.

- Is it an official image ?
- What is installed in the image?
  - Read the Dockerfile that was used to build the image
  - Check the base image

After an image has been downloaded, you may then run a container using the downloaded image with the sub-command `run` . If an image has not been downloaded when Docker is executed with the sub-command `run` , the Docker client will first download the image, then run a container using it:

```
docker run hello-world:linux
```

Note

Here we use the `linux` tag of the hello-world image instead of using `latest` again.

To see the images that have been downloaded to your computer type:

```
docker images
```

The output should look similar to the following:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mariadb             latest              58730544b81b        2 weeks ago         397MB
hello-world         latest              1815c82652c0        2 months ago        1.84kB
hello-world         linux               1815c82652c0        2 months ago        1.84kB
```

The hello world container you ran in the previous lab is an example of a container that runs and exits, after emitting a test message. Containers, however, can be much more useful than that, and they can be interactive. After all, they are similar to virtual machines, only more resource-friendly.

As an example, let's run a container using the latest image of MariaDB. The combination of the -i and -t switches gives you interactive shell access to the container:

```
docker run -it mariadb
```

Note for Windows

If you are using `git-bash` , `cmd` or `powershell` on a Windows system, be aware of the error:

```
the input device is not a TTY.  If you are using mintty, try prefixing the command with 'winpty'
```

Put `winpty` at the beginning of every command that uses the `-it` paramaters. e.g.:

```
winpty docker run -it mariadb
```

An error has popped up!

```
2022-08-09 08:19:21+00:00 [Note] [Entrypoint]: Entrypoint script for MariaDB Server 1:10.8.3+maria~jammy started.
2022-08-09 08:19:21+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2022-08-09 08:19:21+00:00 [Note] [Entrypoint]: Entrypoint script for MariaDB Server 1:10.8.3+maria~jammy started.
2022-08-09 08:19:21+00:00 [ERROR] [Entrypoint]: Database is uninitialized and password option is not specified
        You need to specify one of MARIADB_ROOT_PASSWORD, MARIADB_ALLOW_EMPTY_ROOT_PASSWORD and MARIADB_RANDOM_ROOT_PASSWORD
```

Everything is fine, to run this image there is some configuration needed. Read the following excerpt carefully.

```
error: database is uninitialized and password option is not specified
        You need to specify one of MARIADB_ROOT_PASSWORD, MARIADB_ALLOW_EMPTY_ROOT_PASSWORD and MARIADB_RANDOM_ROOT_PASSWORD
```

More on passing configuration to containers in the next lab .

Think of an image like a blueprint of what will be in a container when it runs.

- An image is a collection of files + some metadata (or in technical terms: those files form the root filesystem of a container)
- Images are made of layers, conceptually stacked on top of each other
- Each layer can add, change or remove files
- Images can share layers to optimize disk usage, transfer times and memory use
- You build these images using Dockerfiles (in later labs)
- Images are immutable, you cannot change them after creation

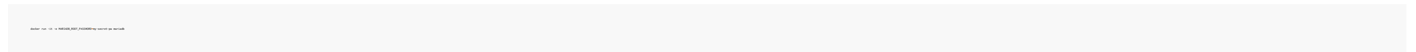
When you run an image, it becomes a container.

- An image is a read-only filesystem
- A container is an encapsulated set of processes running in a read-write copy of that filesystem
- To optimize container boot time, copy-on-write is used instead of regular copy
- docker run starts a container from a given image

3. Environment variables

Environment variables

So why was there an error in the previous lab? The MariaDB server is not able to run without a proper configuration. Docker can pass variables into the instantiation process via environment variables. Environment variables are passed via the parameter -e , e.g.:



```
docker run -it -e MARIADB_ROOT_PASSWORD=my-secret-pw mariadb
```

Once you run the command you will see an output like this:

```
Initializing database

PLEASE REMEMBER TO SET A PASSWORD FOR THE MariaDB root USER !
To do so, start the server, then issue the following commands:

'/usr/bin/mysqladmin' -u root password 'new-password'
'/usr/bin/mysqladmin' -u root -h  password 'new-password'

Alternatively you can run:
'/usr/bin/mysql_secure_installation'

which will also give you the option of removing the test
databases and anonymous user created by default.  This is
strongly recommended for production servers.

See the MariaDB Knowledgebase at http://mariadb.com/kb or the
MySQL manual for more instructions.

Please report any problems at http://mariadb.org/jira

The latest information about MariaDB is available at http://mariadb.org/.
You can find additional information about the MySQL part at:
http://dev.mysql.com
Consider joining MariaDB's strong and vibrant community:
https://mariadb.org/get-involved/

Database initialized
MySQL init process in progress...
2020-05-27  6:21:03 0 [Note] mysqld (mysqld 10.3.7-MariaDB-1:10.3.7+maria~jessie) starting as process 101 ...
2020-05-27  6:21:03 0 [Note] InnoDB: Using Linux native AIO
2020-05-27  6:21:03 0 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
2020-05-27  6:21:03 0 [Note] InnoDB: Uses event mutexes
2020-05-27  6:21:03 0 [Note] InnoDB: Compressed tables use zlib 1.2.8
2020-05-27  6:21:03 0 [Note] InnoDB: Number of pools: 1
2020-05-27  6:21:03 0 [Note] InnoDB: Using SSE2 crc32 instructions
2020-05-27  6:21:03 0 [Note] InnoDB: Initializing buffer pool, total size = 256M, instances = 1, chunk size = 128M

...

2020-05-27  6:21:08 0 [Note] mysqld: ready for connections.
Version: '10.3.7-MariaDB-1:10.3.7+maria~jessie'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306  mariadb.org binary distribution
```

If you re-read the command above you will notify that we used the arguments -it (interactive/terminal). And you might have also found out that mariadb does not react to the usual CTRL-c . So how do we exit this terminal? Docker has an escape sequence to detach from a container and leave it running. For this you have to press CTRL-p and then CTRL-q in bash.

Note for Webshell

In the webshell the shortcuts CTRL-p and CTRL-q are not working. Simply close the terminal and open a new one as a workaround.

Note for Windows

This might not work on Windows since the shortcuts CTRL-p and CTRL-q are already used for other purposes. Use docker ps in a separate shell to get the container ID and then stop it using docker stop <container>. After that, you have to restart the container with docker start <container> or start a new container with docker run -d ... as described in the section *Detached containers* below.

This will leave the container running while you are back in your shell. To verify that the container is running use the following command:

```
docker ps
```

The output should look much like this:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
7cb31f821233        mariadb             "docker-entrypoint..."   5 minutes ago       Up 5 minutes        3306/tcp            upbeat_black
well
```

Access the container

To connect to the container again you can use the following command:

```
docker exec -it <container> bash
```

Where `<container>` can refer to the `CONTAINER ID` (the first two characters are normally sufficient) or one of the `NAMES` from the output of `docker ps`. In the above output this would be `7cb31f821233` or `upbeat_blackwell`.

> **Note**
> The docker exec command needs either the ID or NAME of the container. Additionally, at the end, an executable.

In this example, it's `bash` as we want to do something interactively in the container.

Once the command is executed you should see this:

```
root@7cb31f821233:/#
```

> **Note**
> Every time you connect yourself to a container you will always be the user that was defined in the Dockerfile.

Now that we are connected, let's find out if the MariaDB is working…

```
mariadb -uroot -pmy-secret-pw
```

If everything works as expected, you should now see the MariaDB command line:

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 8
Server version: 10.3.8-MariaDB-1:10.3.8+maria~jessie mariadb.org binary distribution

Copyright (c) 2000, 2020, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Type

```
exit;
```

to leave the mysql client. Type

```
exit
```

one more time to leave the container.

Detached containers

One might think: *This whole starting process is a bit cumbersome with* `CTRL-p` *and then* `CTRL-q`. Therefore, you can run a Docker container directly with `-d` (detached) mode, e.g.:

```
docker run -it -e MARIADB_ROOT_PASSWORD=my-secret-pw -d mariadb
```

Instead of the output of the container itself, you will now only get the ID of the started container. If you have a look into the container list, you should see two running containers:

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
699e82ed8f1f        mariadb             "docker-entrypoint..."   3 minutes ago       Up 3 minutes        3306/tcp            jolly_bardeen
7cb31f821233        mariadb             "docker-entrypoint..."   32 minutes ago      Up 32 minutes       3306/tcp            upbeat_blackwell
```

We don’t need both of them running, let us take care of that in the next lab.

4. Deleting containers

Deleting a container

There are two ways to stop a container, we start with the recommended way. You first have to stop the container. Some might still know the CONTAINER ID or the NAME of the container but for those who don’t:

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
699e82ed8f1f        mariadb             "docker-entrypoint..."   3 minutes ago       Up 3 minutes        3306/tcp            jolly_bardee
n
7cb31f821233        mariadb             "docker-entrypoint..."   32 minutes ago      Up 32 minutes       3306/tcp            upbeat_black
well
```

To stop a container use the command:

```
docker stop <container>
```

After that, check the new state with

```
docker ps
```

This should show only one container running:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
699e82ed8f1f        mariadb             "docker-entrypoint..."   9 minutes ago       Up 9 minutes        3306/tcp            jolly_bardee
n
```

We just exited the container “gracefully”, but as an alternative you can also kill a container with the `docker kill <container>` command. This stops the container immediately by using the KILL signal.

You may recognize that the container *upbeat_blackwell* is not present in the container list anymore. That is because `docker ps` only shows running containers, but as always you have a parameter that helps:

```
docker ps --all
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                          PORTS               N
AMES
699e82ed8f1f        mariadb             "docker-entrypoint..."   12 minutes ago      Up 12 minutes                   3306/tcp            j
olly_bardeen
7cb31f821233        mariadb             "docker-entrypoint..."   41 minutes ago      Exited (0) 2 minutes ago                            u
pbeat_blackwell
67d79f95c712        hello-world         "/hello"                 About an hour ago   Exited (0) About an hour ago                        u
pbeat_boyd
```

Now that the *upbeat_blackwell* container is stopped delete it:

```
docker rm <container>
```

Now the container has disappeared from the list:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                          PORTS               N
AMES
699e82ed8f1f        mariadb             "docker-entrypoint..."   13 minutes ago      Up 13 minutes                   3306/tcp            j
olly_bardeen
67d79f95c712        hello-world         "/hello"                 About an hour ago   Exited (0) About an hour ago                        u
pbeat_boyd
```

Note

It is a good idea to delete unused containers to save disk space.

The `CONTAINER ID` and `NAME` values are unique identifiers for a container. If we don't provide one, `docker` will come up with a unique name. More on that in the next lab !

5. Names

Naming a container

Unlike the `CONTAINER ID` , the `NAME` is something we can manipulate. The name is handy, not only for starting/connecting/stopping/destroying a container, but also for networking (which we will see in a later lab).

To set a name, add the `--name` parameter to Docker's `run` command:

```
docker run --name mariadb-container -e MARIADB_ROOT_PASSWORD=my-secret-pw -d mariadb
```

As always, to check if this has really worked out, look at the container list:

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
6f08ac657320        mariadb             "docker-entrypoint..."   58 seconds ago      Up 57 seconds       3306/tcp            mariadb-cont
ainer
699e82ed8f1f        mariadb             "docker-entrypoint..."   24 minutes ago      Up 24 minutes       3306/tcp            jolly_bardee
n
```

Instead of accessing the database from inside the container like in the last lab, we access it from outside using a local mysqlclient.

This is a bit tricky. First find out the IP address of your docker container. Therefore, use this command:

```
docker inspect mariadb-container  -f '{{ range.NetworkSettings.Networks }}{{ .IPAddress }}{{ end }}'
```

`docker inspect <container>` shows you details about a running container in JSON format (run it yourself and take a look at it).
We filtered the json to only get the IP address of the container.
We could also have filtered the output with grep: `docker inspect mariadb-container | grep IPAddress` but our solution is more elegant 😊.

Once you have the IP (in your example `172.17.0.2`) connect to it:

```
mariadb -h172.17.0.2 -uroot -pmy-secret-pw
```

If everthings works, exit mysql-client

```
exit
```

Note for Windows

The mysql client must be installed on your computer. On Windows, you can use the binary from the ZIP archive at https://dev.mysql.com/downloads/mysql/ .

Also on Windows, you must use port-forwarding to access the database:

```
docker run --name mariadb-container -p 3306:3306 -e MARIADB_ROOT_PASSWORD=my-secret-pw -d mariadb
```

Now you should be able to access the database with:

```
winpty mysql.exe -hlocalhost -uroot -pmy-secret-pw mariadb
```

Instead of entering the container with bash, we could also directly run mysql inside the container: `docker exec -it mariadb-container mariadb -uroot -pmy-secret-pw`

6. Working with volumes

It’s gone. The docker instance has no persistence layer to store data permanently, let us address that problem in this chapter.

Mounting a volume in a container

The MariaDB container is a good example as to why it’s good to have an external volume. There are several possibilities on how to work with volumes in Docker, in this case, we’re going to create a docker volume to store the persistent data of our MariaDB. The volume is managed by Docker itself.

Create the docker-managed volume with:

```
docker volume create volume-mariadb
```

Now use the created volume and attach it to the MariaDB database.

With the parameter `-v` attach the volume to a path in the container:

```
docker run --name mariadb-container-with-external-volume -v volume-mariadb:/var/lib/mysql -e MARIADB_ROOT_PASSWORD=my-secret-pw -d mariadb
```

See Docker’s Volumes documentation for more information.

Okay, now create a new user in the MariaDB container:

```
docker exec -it mariadb-container-with-external-volume mariadb -uroot -pmy-secret-pw
```

Inside the mariadb-client execute some SQL commands:

```
use mysql
CREATE USER 'peter'@'%' IDENTIFIED BY 'venkman';
GRANT SELECT ON * . * TO 'peter'@'%';
```

Once all steps are completed quit the mysql session and exit the container:

```
exit;
```

To test if Peter has been created correctly, just login using his credentials.

Now stop and remove the `mariadb-container-with-external-volume` container.

```
docker stop mariadb-container-with-external-volume
docker rm mariadb-container-with-external-volume
```

Next, check if the data is still available. Create a new MariaDB container with the previous volume:

```
docker run --name mariadb-container-with-existing-external-volume \
           -v volume-mariadb:/var/lib/mysql \
           -e MARIADB_ROOT_PASSWORD=my-secret-pw \
           -d mariadb
```

The moment of truth… Connect to the database server using Peter’s credentials:

```
docker exec -it mariadb-container-with-existing-external-volume mariadb -upeter -pvenkman
```

You should now be connected to your database instance as `peter` . You can test this by listing the users with the sql client:

```
SELECT User FROM mysql.user;
```

```
+-------------+
| User        |
+-------------+
| peter       |
| root        |
| healthcheck |
| healthcheck |
| healthcheck |
| mariadb.sys |
| root        |
+-------------+
7 rows in set (0.001 sec)
```

Now exit the mariadb client

```
exit;
```

Additional info for working with Docker volumes

Docker volumes can be used for:

- Decoupling the data that is stored from the container which created the data
- Bypassing the copy-on-write system to obtain native disk I/O performance
- Bypassing copy-on-write to leave some files out of docker commit
- Sharing a directory between multiple containers
- Sharing a directory between the host and a container
- Sharing a single file between the host and a container An alternative to working with volumes would be to mount local directories (host folders) by a path into your container. We will use this in chapter 08.

Docker storage driver

When running a lot of Docker containers on a machine you usually need a lot of storage. Docker volumes and container storage are provided on a filesystem. The following link provides additional information on how to choose the correct storage setup:

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/managing_containers/managing_storage_with_docker_formatted_containers

At the moment, `overlay2` is the recommended storage driver .

7. Frontend container

Now that we have a “backend”, why not deploy a frontend container (e.g. httpd & php) and make them speak to each other?

Deploying a frontend container

First thing: Find the fitting Docker image –> Where? Exactly… Docker Hub .

Use the `php:8-apache` image.

```
docker pull php:8-apache
```

Once it is pulled check your local `docker images` :

```
docker images
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
php                 8-apache            41f84befd707        5 days ago          390MB
mariadb             latest              58730544b81b        2 weeks ago         397MB
hello-world         latest              1815c82652c0        2 months ago        1.84kB
hello-world         linux               1815c82652c0        2 months ago        1.84kB
```

This will show the images in the local registry with their name and tags.

By using `docker pull php:8-apache` Docker downloaded the `php` image with the `8-apache` tag. If you omit the tag (so here `docker pull php`) docker would try to download the image with the `latest` tag.

For the hello-world image we see that we have the same image (read same image id) but two different tags (linux/latest).

Now deploy the new container using the correct tag:

```
docker run -d --name apache-php php:8-apache
```

`docker ps` shows all running containers. Check that `apache-php` is running:

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
b901d6c0473a        php:8-apache        "docker-php-entryp..."   18 seconds ago      Up 17 seconds       80/tcp              apache-php
50197361e87b        mariadb             "docker-entrypoint..."   42 minutes ago      Up 42 minutes       3306/tcp            mariadb-cont
ainer-with-existing-external-volume
6f08ac657320        mariadb             "docker-entrypoint..."   4 hours ago         Up 2 hours          3306/tcp            mariadb-cont
ainer
```

Now, try to connect to the server using the container-assigned docker IP address:

Use the familiar command from lab 5:

```
docker inspect apache-php -f '{{ range.NetworkSettings.Networks }}{{ .IPAddress }}{{ end }}'
```

Which will show only the IP of the container as output:

```
172.17.0.4
```

With this IP navigate to the web server at http://172.17.0.4 .

Note for Webshell

As we don’t have a browser in the webshell use `curl http://172.17.0.4` to open the page in your terminal.

Note for Windows and macOS

As the Docker Linux bridge is not reachable from your Windows or macOS host you cannot access the container directly via IP address. See:

- https://docs.docker.com/docker-for-windows/networking/
- https://docs.docker.com/docker-for-mac/networking/

If you’ve already started the `apache-php` container without port forwarding you have to stop and remove it first:

```
docker stop apache-php
docker rm apache-php
```

Now start the container again with port forwarding:

```
docker run -p 8080:80 -d --name apache-php php:8-apache
```

Now you can access the web server at http://LOCALHOST:8080 .

Unfortunately we get a “403 Error - Forbidden”.

403 is the error code from the apache container. There is some configuration missing: The Apache web server does not allow you to scan its own document root.

Docker can port-forward your request to a running container. In a windows environment you have just used this feature. But more explanation in the next lab.

For now stop and remove this container:

```
docker stop apache-php
docker rm apache-php
```

8. Embedding the source code

It looks like the apache image is working but needs a website to display. Create a small web application and run it inside the container.

Create a PHP app

For this lab you're going to need a small PHP app consisting of two files.

First, create a directory for the app's files called `php-app` .

Then, inside that directory, create a new file named `index.php` with the following content:

```
<?php
echo "Welcome to Docker (my young padawan)!";
?>
```

Note for play-with-docker.com

- Create a directory with this shell command: `mkdir php-app`
- Create a file with this shell command: `touch index.php`
- Open your editor
- Select the folder and then the file
- Add the content and save the changes

Lastly, create another file named `db.php` with the following content:

```
<?php
$servername = "mariadb-container-with-existing-external-volume";
$username = "peter";
$password = "venkman";

// Create connection
$conn = new mysqli($servername, $username, $password);

// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
echo "Connected successfully";
?>
```

That's it for the app part.

Mounting the dev environment into your Docker container

Make sure you're outside that freshly created app directory when you execute the next commands.

Now mount the php-app as the host directory into your container:

Linux:

```
docker run -d --name apache-php -v $(pwd)/php-app:/var/www/html php:8-apache
```

Windows (Git Bash):

```
MSYS_NO_PATHCONV=1 docker run -d --name apache-php -v $(pwd)/php-app/:/var/www/html php:8-apache
```

Note

You need to set the absolute path on the -v option, e.g. -v /home/<username>/php-app:/var/www/html or -v C:\Temp\php-app:/var/www/html

You can now check whether the error is still present.

Port forwarding for your Docker container

Docker is able to forward any port you specify to your local machine by using `-p <host-port>:<container-port>` . This is great but also has the possibility of causing port trouble.

Imagine you have a local httpd service running on port 8080, and you are forwarding this same port to your Docker instance.

So first, let us check out if port 8080 is occupied:

In Linux or macOS do the following:

```
lsof -i:8080
```

or in Windows (Git Bash):

```
netstat -aon | findstr :8080 | findstr LISTENING
```

If you get an output, this means port 8080 is used. In that case just us a different port, like 8888, in the examples below.

Now run the image using the `-p` flag:

Linux/Webshell:

```
docker stop apache-php
docker rm apache-php
docker run -p 8080:80 -d --name apache-php -v $(pwd)/php-app:/var/www/html php:8-apache
```

Windows (Git Bash):

```
docker stop apache-php
docker rm apache-php
MSYS_NO_PATHCONV=1 docker run -p 8080:80 -d --name apache-php -v $(pwd)/php-app/:/var/www/html php:8-apache
```

Note regarding the webshell

Do not forget to stop/remove the existing instance of the `apache-php` container before you start a new one.

If you take a look into `docker ps` you'll find an interesting change for the PORTS column

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                  NAMES
6b0721fa6103        php:8-apache        "docker-php-entryp..."   5 seconds ago       Up 4 seconds        0.0.0.0:8080->80/tcp   apache-ph
p
50197361e87b        mariadb             "docker-entrypoint..."   2 hours ago         Up 2 hours          3306/tcp               mariadb-con
tainer-with-existing-external-volume
```

You see that every request coming to port 8080 on your local machine is forwarded to your Docker instance's port 80. If you now type http://LOCALHOST:8080/index.php in your browser you should get the message:
"Welcome to Docker…".
http://LOCALHOST:8080/db.php will produce an error. This is on purpose. Please be patient until the end of lab 10!

Note regarding the Webshell

- Instead of the browser use `curl http://LOCALHOST:8080/index.php` .

Note for play-with-docker.com

To access the frontend app use a special URL

- Copy the SSH connection command (`ssh ip172-18-0-30-bcvhrp0abk8g00cnf9jg@direct.labs.play-with-docker.com`)

> - Remove *ssh* and replace the **@** with a **.**
> - With that URL you will see the app page: `ip172-18-0-30-bcvhrp0abk8g00cnf9jg.direct.labs.play-with-docker.com`

We have succeded in running our own app inside a container .

9. Linking frontend and backend

Now it is time to link your frontend and backend together.

Linking containers

If you have properly worked through all the previous labs you now have the following setup:

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                  NAMES
6b0721fa6103        php:8-apache        "docker-php-entryp..."   25 minutes ago      Up 25 minutes       0.0.0.0:8080->80/tcp   apache-ph
p
50197361e87b        mariadb             "docker-entrypoint..."   2 hours ago         Up 2 hours          3306/tcp               mariadb-con
tainer-with-existing-external-volume
6f08ac657320        mariadb             "docker-entrypoint..."   5 hours ago         Up 3 hours          3306/tcp               mariadb-con
tainer
```

We will put the containers into a new, dedicated network. But first get rid of the currently running ones:

```
docker stop apache-php mariadb-container mariadb-container-with-existing-external-volume
docker rm apache-php mariadb-container mariadb-container-with-existing-external-volume
```

To enable communication between containers, use `docker network` . By default, Docker provides three networks. Verify this is the case:

```
docker network ls
```

Will output the following:

```
NETWORK ID          NAME                DRIVER              SCOPE
9233283df4a6        bridge              bridge              local
640877f8aec4        host                host                local
72f9a9996909        none                null                local
```

For this exercise create a new network with:

```
docker network create container-basics-training
```

If you rerun the list command for Docker networks now, `container-basics-training` will show up.

To make the backend accessible from the frontend run both containers with the `--network` option:

Linux/Webshell:

```
docker run -d --network container-basics-training --name apache-php -v $(pwd)/php-app:/var/www/html -p 8080:80 php:8-apache
docker run -d --network container-basics-training --name mariadb-container-with-existing-external-volume -v volume-mariadb:/var/lib/mysql -e MARIADB_ROOT_PASSWORD=my-secret-pw mariadb
```

Windows (Git Bash):

```
MSYS_NO_PATHCONV=1 docker run -d --network container-basics-training --name apache-php -v $(pwd)/php-app:/var/www/html -p 8080:80 php:8-apache
docker run -d --network container-basics-training --name mariadb-container-with-existing-external-volume -v volume-mariadb:/var/lib/mysql -e MARIADB_ROOT_PASSWORD=my-secret-pw mariadb
```

If you access either container, you should be able to resolve the other container’s address with its container name.

Execute an interactive `bash` shell on the mariadb container.

```
docker exec -it mariadb-container-with-existing-external-volume bash
```

You are now in the Bash session of the mariadb container and the prompt will look like `root@6f08ac657320:/#`

Get the address of the `apache-php` container.

```
getent hosts apache-php
```

The two containers are now able to talk to each other. But let’s check this:

If you type http://localhost:8080/db.php in your browser or use `curl http://localhost:8080/db.php` in the webshell you get… an error! This is because the mysqli extension is not installed in the container.

Obviously, we will not install it in the running container but build a new image. More on that in the next lab.

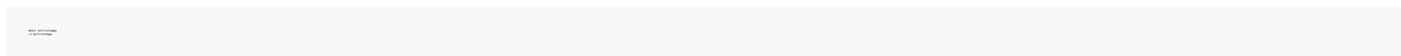
10. Building your own Docker image

Dockerfile

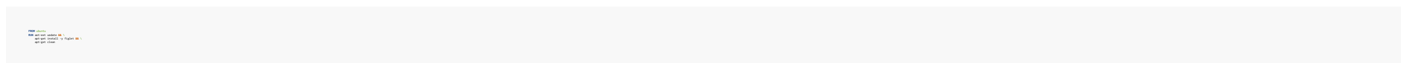
Docker can build container images by reading the instructions on how to build the image from a so-called Dockerfile or more generally, Containerfile. The basic docs on how Dockerfiles work can be found at https://docs.docker.com/engine/reference/builder/ .

Write your first Dockerfile

Before we extend our php image we are going to have a more general look at how to build a container image. For that, create a new directory with an empty Dockerfile in there.



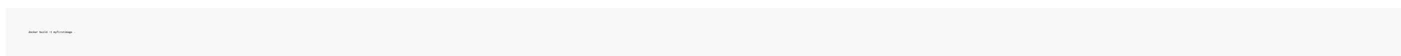
Create a new File with the name `Dockerfile` and add the following content to that `Dockerfile` using your editor of choice:



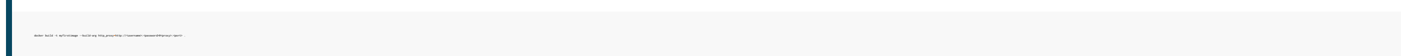
- `FROM` indicates the base image for our build
- Each `RUN` line will be executed by Docker during the build
- Our RUN commands must be non-interactive (no input can be provided to Docker during the build)
- Check https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/ for further best practices on how to write Dockerfiles.

Build the image

Just run:



- `-t` indicates the tag to apply to the image
- `.` indicates the location of the build context (which we will talk more about later, but is basically the directory where our Dockerfile is located)

> **Note**
> Use the additional parameter `--build-arg` when behind a corporate proxy:
>
> 

Please note that the tag can be omitted in most Docker commands and instructions. In that case, the tag defaults to `latest` . Besides being the default tag there's nothing special about `latest` . Despite its name, it does not necessarily identify the latest version of an image.
Depending on the build system it can point to the last image pushed, to the last image built from some branch, or to some old image. It can even not exist at all.
Because of this, you must never use the `latest` tag in production, always use a specific image version.
Also see: https://medium.com/@mccode/the-misunderstood-docker-tag-latest-af3babfd6375

What happens when we build the image

The output of the Docker build looks like this:

```
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM ubuntu
 ---> ea4c82dcd15a
Step 2/2 : RUN apt-get update &&     apt-get install -y figlet &&     apt-get clean
 ---> b3c08112fd1c
Successfully built b3c08112fd1c
Successfully tagged myfirstimage:latest
```

Sending the build context to Docker

```
Sending build context to Docker daemon 84.48 kB
...
```

- The build context is the `.` directory given to docker build
- It is sent (as an archive) to the Docker daemon by the Docker client
- This allows you to use a remote machine to build using local files
- Be careful (or patient) if that directory is big and your connection is slow

Inspecting step execution

```
...
Step 1/2 : FROM ubuntu
 ---> ea4c82dcd15a
Step 2/2 : RUN apt-get update &&     apt-get install -y figlet &&     apt-get clean
 ---> b3c08112fd1c
Successfully built b3c08112fd1c
Successfully tagged myfirstimage:latest
```

- A container (ea4c82dcd15a) is created from the base image
  - The base image will be pulled, if it was not pulled before
- The `RUN` command is executed in this container
- The container is committed into an image (b3c08112fd1c)
- The build container (ea4c82dcd15a) is removed
- The output of this step will be the base image for the next one
- …

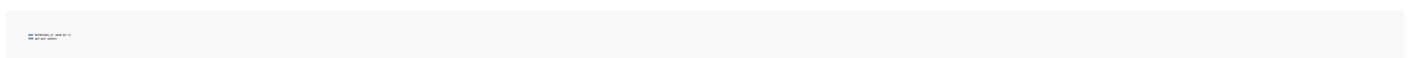
The caching system

If you run the same build again, it will be instantaneous. Why?

- After each build step, Docker takes a snapshot
- Before executing a step, Docker checks if it has already built the same sequence
- Docker uses the exact strings defined in your Dockerfile:
  - `RUN apt-get install figlet cowsay` is different from
  - `RUN apt-get install cowsay figlet`
  - `RUN apt-get update` is not re-executed when the mirrors are updated
- All steps after a modified step are re-executed since the filesystem it's based on may have changed

You can force a rebuild with docker build –no-cache …

If you only want to trigger a partial rebuild, e.g. run `apt-get update` to install the latest updates, you can use the following pattern:



```
ENV REFRESHED_AT 2020-03-13
RUN apt-get update
```

If you update the value of `REFRESHED_AT` it will invalidate the Docker build cache of that and all the following steps, thus installing the latest updates.

Run it

Now run your image

```
docker run -ti myfirstimage
```

You'll find yourself inside a Bash shell in the container, execute

```
figlet hello
```

and you will see the following output:

```
root@00f0766080ed:/# figlet hello
 _          _ _
| |__   ___| | | ___
| '_ \ / _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/

root@00f0766080ed:/#
```

exit the container by executing:

```
exit
```

The CMD instruction in Dockerfile

With the `CMD` instruction in the Dockerfile, we can define the command that is executed when a container is started.

You did find yourself in a shell, so the instruction must either be `/usr/bin/bash` or `/usr/bin/sh` .

Modify the previously created Dockerfile as follows:

```
FROM ubuntu
RUN apt-get update && \
    apt-get install -y figlet && \
    apt-get clean

CMD ["figlet", "hello"]
```

Build the image with:

```
docker build -t myfirstimagecmd .
```

Note

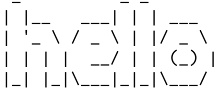
Again, use the additional parameter `--build-arg` when behind a corporate proxy:

```
docker build -t myfirstimagecmd --build-arg http_proxy=http://<username>:<password>@<proxy>:<port> .
```

And run it:

```
docker run -ti myfirstimagecmd
```

It directly executes the defined command and prints out



Check out https://docs.docker.com/engine/reference/builder/#understand-how-cmd-and-entrypoint-interact for more information.

Frontend app image build

After we got to grips with the image building basics, we now want to include the source code of our frontend app in an already-built container image. To achieve this we will create a Dockerfile.

The base image is our `php:8-apache` image which we used before. The `ADD` command allows us to add files from our current directory to the Docker image. We use this command to add the application source code into the image.

> **Note**
> Use `.dockerignore` to exclude files from the Docker context being added to the container. It works the same as `.gitignore`: https://docs.docker.com/engine/reference/builder/#dockerignore-file

In the directory containing the subdirectory `php-app` create a Dockerfile with the following content:

```
FROM php:8-apache

# Copies the php source code to the correct location
ADD ./php-app/ /var/www/html/

# Install additional php extension
RUN docker-php-ext-install mysqli
```

> **Note**
> The `docker-php-ext-install` command might not be able to download the required dependencies if there's a proxy in the way. You can use the additional parameter `--build-arg http_proxy=$HTTP_PROXY` .
>
> Alternatively, you can use the already-built image `puzzle/php-apache-mysqli` for the following labs. Instead of the above Dockerfile you'd use:
>
> ```
> FROM puzzle/php-apache-mysqli
>
> # Copies the php source code to the correct location
> ADD ./php-app/ /var/www/html/
> ```

Build the php-app image

> **Note**
> Stop and delete the running `php-app` container first. Leave the database container running.

Now build the image:

```
docker build -t php-app .
```

Run the php-app container

After a succesful build, run it:

```
docker run -d --network container-basics-training --name php-app -p8080:80 php-app
```

Now open a browser and navigate to http://localhost:8080/db.php (or in the webshell use `curl http://localhost:8080/db.php`). You should get a response saying “Connected successfully”.

Optional lab

Configuration should always be separate from the source code, so the database connection details must not be inside the php file `db.php` . Fix the code in the db.php file. According to the continuous delivery principles, we don't want usernames and passwords in our source code. Use the PHP global variable `$_ENV["<environment variable name>"]` to read environment variables inside the container. Challenge yourself, this time the code is hidden. Try to find the solution before looking at it.

Replace the line

```
$password = "venkman";
```

with

```
$password = $_ENV["password"];
```

and run the container by passing the necessary env var:

```
docker run -d --name apache-php -e password=venkman -v $(pwd)/php-app:/var/www/html php:8-apache
```

10.1 Multi-stage builds

Often you're going to use some kind of libraries, tools or dependencies during the build phase of your application that are not necessary during the runtime of the container. We want to keep the actual artifact as independent and small as possible. So we often remove these dependencies in the build phase after the application itself has been built.

In this lab you're going to learn how to use multistage builds and what they are good for.

Purpose

If the application is not available as a prebuilt artifact, in many cases, the application itself gets built directly during the docker build process `docker build -t ...`

Examples

Let's have a look at some examples:

Java Spring Boot Gradle build

The complete example can be found at https://github.com/appuio/example-spring-boot-helloworld .

```
FROM registry.access.redhat.com/ubi9/openjdk-17

LABEL org.opencontainers.image.authors="midcicd@puzzle.ch" \
      io.k8s.description="APPUiO Example Spring Boot App" \
      io.k8s.display-name="APPUiO Spring Boot App" \
      io.openshift.expose-services="8080:http" \
      io.openshift.tags="springboot"

EXPOSE 8080 9000

RUN mkdir -p /tmp/src/
ADD . /tmp/src/

RUN cd /tmp/src && sh gradlew build --no-daemon

RUN ln -s /tmp/src/build/libs/springboots2idemo*.jar /deployments/springboots2idemo.jar
```

During the docker build the actual application source code is added to the context and built using the `gradlew build` command. Gradle in this case is only used during the build phase, since it produces a jar that is then executed with `java -jar ...` at execution time.

Build phase dependencies:

- Java
- Gradle

Runtime phase dependencies:

- Java

Static HTML, CSS, JS example

A docker image that serves static content like HTML, CSS, JS which will only be served via a simple web server like nginx or Apache. We don't want/need the build tools to be in the resulting docker image.

During the build phase, tools are needed to do:

- Creating HTML from templates, for example with a rendering framework
- Installing Javascript dependencies with Yarn, NPM, …
- Compiling CSS with less, Sass, …
- Minify, uglify, caching
- Optimizing images, creating different sizes of the images
- And so on …

during the execution time of the image, actually only the created static content must be available.

Go application

A go application is a great use case for multi-stage builds, since the resulting artifact is an executable binary containing every dependency that is needed to run the application. That means that the resulting docker image can be very small, so the base image we use is a simple alpine linux.

Multi stage Docker build:

```
FROM golang:1.9
WORKDIR /go/src/myapp
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:3.6
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/myapp .
CMD ["./app"]
```

Content of the Go application `app.go` :

```
package main
import (
        "net/http"
        "fmt"
)
func main() {
        http.HandleFunc("/", hello)
        http.ListenAndServe(":8180", nil)
}
func hello(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "Docker workshop")
}
```

Multi-stage builds

With multistage builds you now have the possibility to actually split these two phases, so that you can pass the built artifact from phase one into the runtime phase, without the need of installing build time dependencies in the resulting docker image. Which means that the image will be smaller and consist of less unneeded dependencies.

Read more about Docker multi-stage builds at https://docs.docker.com/develop/develop-images/multistage-build/

Optional lab: Create a multi-stage build

Turn the docker build from the first example (Java Spring boot https://github.com/appuio/example-spring-boot-helloworld) into a docker multistage build. As a second image you can use `registry.access.redhat.com/ubi9/openjdk-17-runtime` . Try to find the solution before looking at it.

```
FROM registry.access.redhat.com/ubi9/openjdk-17

LABEL org.opencontainers.image.authors="noreply@acend.ch" \
      io.k8s.description="acend example spring boot app" \
      io.k8s.display-name="acend spring boot app" \
      io.openshift.expose-services="8080:http" \
      io.openshift.tags="springboot"

EXPOSE 8080 9000

RUN mkdir -p /tmp/src/
ADD . /tmp/src/

RUN cd /tmp/src && sh gradlew build --no-daemon

FROM registry.access.redhat.com/ubi9/openjdk-17-runtime

EXPOSE 8080 9000

COPY --from=0 /tmp/src/build/libs/springboots2idemo*.jar /deployments/springboots2idemo.jar
```

10.2 Security scanning

A lot of the container images contain security issues. If you want to use images from public registries you should be able to scan them to verify the image is clean.

There are some tools on the market which can help you to check these images.

In this lab you will learn how to scan an image with the tool Trivy .

Scan image for vulnerabilities and secrets

Trivy is as simple as it sounds! Just point it to your image and it will do the rest.

When Trivy runs for the very first time, it will download the latest vulnerability database from the internet. Without this, Trivy is not able to scan anything.

> **Note**
> A pre-download of the database is possible if Trivy has no direct access to the internet.

Scan an image for vulnerabilities:

```
trivy image quay.io/acend/example-web-python:latest
```

Above command will generate output looking similar to this:

```
2022-05-11T11:47:17.787Z        INFO    Need to update DB
2022-05-11T11:47:17.787Z        INFO    DB Repository: ghcr.io/aquasecurity/trivy-db
2022-05-11T11:47:17.787Z        INFO    Downloading DB...
31.76 MiB / 31.76 MiB [----------------------------------------------------------------------------------------------------] 100.00% 10
.19 MiB p/s 3.3s
2022-05-11T11:47:26.485Z        INFO    Detected OS: alpine
2022-05-11T11:47:26.485Z        INFO    Detecting Alpine vulnerabilities...
2022-05-11T11:47:26.487Z        INFO    Number of language-specific files: 1
2022-05-11T11:47:26.487Z        INFO    Detecting python-pkg vulnerabilities...

quay.io/acend/example-web-python:latest (alpine 3.15.4)
=======================================================
Total: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)


Python (python-pkg)
===================
Total: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)
```

No issues were detected in this image (at time of writing). So let’s check another image:

```
trivy image debian:stable-slim
```

As you can see, the output is now much larger than before. You could use Trivy in pipelines where it can break the build if the image has serious severities like HIGH or CRITICAL. For more information check the `trivy --help` page.

Scan filesystem for vulnerabilities, secrets and misconfigurations

You can also scan projects locally on disc for several things. This scan checks your code for best practices and gives you hints about configurations issues and mistakes which can help you to improve your codebase.

```
git clone https://github.com/acend/awesome-apps.git
trivy fs --security-checks vuln,secret,config awesome-apps/
```

```
go/Dockerfile (dockerfile)
==========================
Tests: 23 (SUCCESSES: 23, FAILURES: 0, EXCEPTIONS: 0)
Failures: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)


python/Dockerfile (dockerfile)
==============================
Tests: 23 (SUCCESSES: 23, FAILURES: 0, EXCEPTIONS: 0)
Failures: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)
```

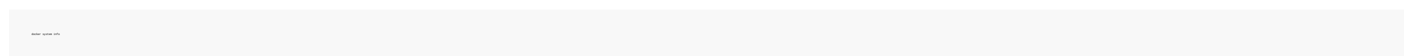
For more information, check out the Trivy Github page .

11. Debugging running containers

In this lab, you’ll learn some more advanced features used when debugging, stopping and starting containers

Docker info

To receive general info about your docker environment use



```
Containers: 42
 Running: 0
 Paused: 0
 Stopped: 42
Images: 75
Server Version: 18.06.1-ce
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 468a545b9edcd5932818eb9de8e72413e616e86e
runc version: 69663f0bd4b60df09991c08812a60108003fa340
init version: fec3683
Security Options:
 apparmor
 seccomp
  Profile: default
Kernel Version: 4.15.0-36-generic
Operating System: Ubuntu 18.04.1 LTS
OSType: linux
Architecture: x86_64
CPUs: 4
Total Memory: 15.31GiB
Name: system-name
ID: QSH5:AZKW:AZKW:FZLG:AZKW:2WRM:AZKW:OOFU:UUTI:AZKW:YBG5:QI4F
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Username: philipona
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
```

Listing containers

Note

`docker ps` and `docker container ls` are equivalent. This also applies to other top-level commands.

For brevity, we are going to use `docker ps` etc. even though the `docker container` commands are more consistent. See `docker container --help` for a full list of sub-commands.

To see all running containers:

```
docker ps
```

To see all containers, also exited containers:

```
docker ps -a
```

To see only the last container that was started:

```
docker ps -l
```

To see only the ID of containers:

```
docker ps -q
```

To see only the ID of the last started container:

```
docker ps -ql
```

To see the size of the containers:

```
docker ps -s
```

To see the general docker storage usage:

```
docker system df
```

This is helpful for scripting or doing a lot of experimentation where you start and delete a container quite a lot of times. As an example `docker rm -f $(docker ps -ql)` , which will delete the last started container.

Stopping containers

Take a look at lab 04 .

Restarting and attaching to containers

We have started containers in the foreground and in the background. Now we will see how to:

- Put a container in the background.
- Attach to a background container to bring it to the foreground.
- Restart a stopped container

Background and foreground

From Docker's point of view, all containers are the same. All containers run the same way, whether there is a client attached to them or not.

It is always possible to detach from a container, and to re-attach to a container.

Detaching from a container

If you have started an interactive container (with option -it), you can detach from it.

- The detach key sequence is `CTRL-p CTRL-q` .
- Or you can detach by killing the Docker client.

Don’t hit `CTRL-c` , as this would deliver SIGINT to the container

What does `-it` stand for?

- `-t` means “terminal” as in “allocate a terminal.”
- `-i` means “interactive” as in “connect stdin to the terminal.”

Attaching to a container

You can attach to a container:

```
docker attach <container>
```

- The container must be running.
- There can be multiple clients attached to the same container.
- Warning: if the container was started without `-it` :
  - You won’t be able to detach with `CTRL-p CTRL-q` .
  - If you hit `CTRL-c` , the signal will be proxied to the container.

Remember: you can always detach by killing the Docker client (e.g. close the bash window).

Executing a command in a container

You can execute a command in a container:

```
docker exec -it <container> <command>
```

E.g. if you want to execute a shell, then run the following command:

```
docker exec -it <container> /bin/bash
```

Checking container output

Use `docker attach` if you intend to send input to the container. If you just want to see the output of a container, use `docker logs` .

Restarting a container

When a container has exited, it is in stopped state. It can then be restarted with the start command: `docker start <container>`

The container will be restarted using the same options you launched it with. You can re-attach to it if you want to interact with it.

Listing images

We already stumbled upon the command to list Docker images. See [lab 02](#) .

Viewing logs of containers

```
docker logs <container>
```

This will show the whole log of that container, sometimes it’s enough to display just a few lines:

```
docker logs --tail 3 <container>
```

With the `-follow` option you can tell the `docker logs` command to follow the log file in real time:

```
docker logs --tail 3 --follow <container>
```

Housekeeping

There are various housekeeping commands.

Dev environment

> Note
> Do not use these commands in production!

Stop all running containers and then delete them:

```
docker stop $(docker ps -a -q)
docker rm $(docker ps -a -q)
```

Delete all images:

```
docker rmi $(docker images -q)
```

Pruning

Remove unused data:

```
docker system prune
```

Remove all stopped containers:

```
docker container prune
```

Remove unused images:

```
docker image prune
```

12. Orchestration

Docker Compose

Docker Compose is a tool for defining and managing multi-container Docker applications. It allows you to configure all your app’s services (like a web server, database, cache, etc.) using a single YAML file, and run them with a single command.

Docker Swarm

Docker Swarm is Docker’s native orchestration tool that allows you to deploy, manage, and scale a cluster of Docker containers across multiple machines (called nodes) as if they were a single system.

Think of it as a way to coordinate a group of Docker hosts to work together and run containerized applications in a distributed, highly available way.

12.1 Docker Compose

Instead of managing the containers with the `docker` command, you may use Docker Compose to handle them.

> Note
>
> Ubuntu users need to install the `docker-compose` command:
>
> ```
> sudo apt-get install docker-compose
> ```
>
> On Windows, the Docker installer usually includes `docker-compose` already.

Docker Compose file

Previously we ran:

```
docker run --name mariadb-container-with-existing-external-volume -v$(pwd)/datastore-mysql:/var/lib/mysql -it -e MARIADB_ROOT_PASSWORD=my-secret-pw -d mariadb
```

and:

```
docker run -itd --name php-app -p8080:80 --link mariadb-container-with-existing-external-volume php-app
```

We now create a file called `docker-compose.yml` :

```
version: '3'

services:

  php-app:
    image: php-app
    ports:
      - '8080:80'
    networks:
      - container-basics-training

  mariadb-container-with-existing-external-volume:
    image: mariadb
    environment:
      - MARIADB_ROOT_PASSWORD=my-secret-pw
    volumes:
      - 'volume-mariadb:/var/lib/mysql'
    networks:
      - container-basics-training

networks:
  container-basics-training:

volumes:
  volume-mariadb:
```

For each of the `docker run` commands, you add an entry under `services` , containing the appropriate options. The various options are described in the Compose file reference .

Having this file, you can run both containers with a simple command:

```
docker-compose up
```

Then again, check http://LOCALHOST:8080/db.php in a browser (or curl in another terminal in the webshell).

To stop the containers, hit CTRL+c followed by

```
docker-compose down
```

This will stop and remove the services.

12.2 Docker Swarm

Installation

You can create a simple docker swarm server on any machine in the same network.

> **Note**
> For this lab, you should pair up with one of your neighbors and create a cluster.

Creating a Swarm

First of all we need to run the following command on one shell:

```
docker swarm init
```

Add nodes to swarm

As the cluster has now been created, we can just add another node by running:

```
docker swarm join --token ...
```

To display the token afterwards you can use the following:

```
docker swarm join-token manager
```

Viewing the current nodes

Check all nodes which are part of the cluster

```
docker node ls
```

Services

The cluster is running, but really empty. Let’s change that!

Deploy a service

Here we go:

```
docker service create --replicas 2 --name awesome-app quay.io/acend/example-web-go:latest
```

Check the running service by one of these commands, it there a difference?

```
docker service ls
docker ps
```

Try this on both nodes!

Scale a service

Scale the service by increasing the replicas and inspect the config

```
docker service scale awesome-app=4
docker inspect awesome-app
```

Where are the pods scheduled? Check all the nodes!

Deleting a service

To delete a service we can simply use this command.

```
docker service rm awesome-app
```

Stacks

Docker stacks is a collection of more than one service using the already known docker compose config.

Creating a stack

Create a file called docker-compose.yaml with:

```yaml
version: "3.7"
services:
  nginx:
    image: quay.io/acend/example-web-go:latest
    ports: ["5000:5000"]
    deploy:
      replicas: 3
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 100M
```

And deploy the stack with

```
docker stack deploy -c docker-compose.yaml stack-example-web-go
```

Managing a stack

Check what we’ve got

```
docker stack ls
docker ps
```

Try to access our application by running curl

```
curl localhost:5000
```

Delete a stack

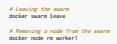
Finally we can delete this simple example by running:

```
docker stack rm stack-example-web-go
```

Cleanup

Leaving the swarm

To remove one node you have to “leave” the swarm or remove it from the manager node.



```
# Leaving the swarm
docker swarm leave

# Removing a node from the swarm
docker node rm worker1
```

- acend gmbh

13. Registry and Docker Hub

So far, we only built and ran the Docker images locally on our computers, but what if we want to integrate them into a CI/CD pipeline or even only distribute the built images? Similar to a Maven artifact repo, where the built JARs, WARs, and EARs get deployed to distribute later, there is the concept of a Docker registry.

In this lab we're going to learn:

- Docker Hub platform
- Pull and push images from and to the registry
- Create automated docker builds on Docker Hub

Docker Hub

Most of the images and base images you ran in the previous labs were downloaded from Docker Hub, when not already present on your computer.

Docker Hub is an online Docker registry, repository management, and build platform. It consists of public and private repositories. Docker Hub integrates with various online services such as GitHub or GitLab.

Publicly available images can be pulled without logging into the platform.

The following command pulls the `nginx` image to your local machine

```
docker pull nginx
```

```
Using default tag: latest
latest: Pulling from library/nginx
f17d81b4b692: Pull complete
d5c237920c39: Pull complete
a381f92f36de: Pull complete
Digest: sha256:4ddaf6043a77aa145ce043d4c662e3768556421d6c0a65d303e89977ad3c9636
Status: Downloaded newer image for nginx:latest
```

Run the following command to verify whether the image is now locally available or not.

```
docker images
```

```
REPOSITORY                                                 TAG                 IMAGE ID            CREATED             SIZE
nginx                                                      latest              dbfc48660aeb        4 hours ago         109MB
```

`docker run` or `docker build` does a `docker pull` implicitly when an image is not yet locally available.

Create a Docker Hub account

Now it's time to prepare for pushing a docker image to a Docker Hub repository.

- Create a Docker Hub account, if you don't already have one: https://hub.docker.com/
- Create a new public repository: use the **Create Repository** button after logging in.
- Check the repository info, the newly created repository is empty.

Log in to Docker Hub with your account:

```
docker login
```

```
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to
create one.
Username: <username>
Password: <password>
WARNING! Your password will be stored unencrypted in /home/home/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

Push your first image

Let’s push our “myfirstimage”. But first we need to tag the image with the same name as the repository we’ve created previously:

```
docker tag myfirstimage <docker-user>/<docker-repo>
docker push <docker-user>/<docker-repo>
```

```
The push refers to repository docker.io/<docker-user>/<docker-repo>
213d31159bea: Pushed
9f9cacfd69bf: Pushed
a1950e3866f0: Mounted from library/php
2dc96af1a4a5: Mounted from library/php
8cb5b8d4756a: Mounted from library/php
bcc4727d0912: Mounted from library/php
59e338bee70e: Mounted from library/php
369e6fd590f3: Mounted from library/php
1805144065e1: Mounted from library/php
b6311cdc5fb6: Mounted from library/php
e30181a94bbf: Mounted from library/php
481da43a1302: Mounted from library/php
a4ace4ed0385: Mounted from library/php
fd29e0f8792a: Mounted from library/php
687dad24bb36: Mounted from library/php
237472299760: Mounted from library/php
latest: digest: sha256:604c7893ff67fab19fddcaaf89935f9bd252a8711e1d42ab3c8c837144b57eee size: 3659
```

Tagging images

Tagging images allows us to keep track of different versions without the need of remembering the image ID, e.g. `ca751384b926` . This said, a tag is a pointer to a specific image ID.

```
docker images
```

```
REPOSITORY                        TAG         IMAGE ID       CREATED            SIZE
myfirstimage                      latest      ca751384b926   About an hour ago  368MB
acend/container-basics-training   1           ca751384b926   About an hour ago  368MB
acend/container-basics-training   latest      ca751384b926   About an hour ago  368MB
nginx                             latest      dbfc48660aeb   5 hours ago        109MB
...
```

Tagging an image by ID:

```
docker tag <image-id> acend/container-basics-training:<tag>
```

Tagging an image by name:

```
docker tag myfirstimage acend/container-basics-training:<tag>
```

Tagging an image by name and tag:

```
docker tag myfirstimage:1 acend/container-basics-training:<tag>
```

Tagging an image for a private registry:

```
docker tag <image> myregistryhost:5000/acend/container-basics-training:<tag>
```

> **Note**
> Using the `latest` tag can be tricky. First of all it's important to define dependencies explicit and under version control, so that builds are reproducible. Second the latest tag basically means `"the last build/tag that ran without a specific tag/version specified"` https://medium.com/@mccode/the-misunderstood-docker-tag-latest-af3babfd6375

Docker Enterprise

Docker has its own enterprise features, which are available for Enterprises at https://www.docker.com/products/docker-enterprise .

Docker Hub alternatives

There are a couple of other public registries available, such as https://quay.io/ for example.

Private registries

A docker registry can be deployed on premises as well. Most of the available artifact repository applications such as Nexus and Artifactory are able to manage Docker images as well.

Additional lab

Visit https://hub.docker.com/ , https://docker.com/ , https://quay.io/ and compare the features available for free and subscription based plans. Take a look at the security image scan functionalities of those services.

14. Additional best practices

In addition to the best practices under https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ we encourage you to be aware of the following additional points

Base images

- Use only trusted base images
- Use only regularly updated and supported images
- If you use self-built base images make sure to trigger the complete dependency chain of your application images
- Trigger your build when the base images get updated
- Use security scans to check your images for security issues