# Containers

A container is a portable computing environment. It contains everything needed to run a workflow or application, including dependencies, code, and configuration.

## Making it less abstract

We can think of a container like a new computer on which we copy our code or workflow and install all needed dependencies. Once everything is installed and configured, we make backups of that computer. Imagine we could now use that backup on another computer, and everything we installed and configured would work just like it did originally. This backup is similar to a container; both are a packaging of code together with its dependencies (including the operating system) and configuration. That's where the analogy ends since a container has many advantages over something like a backup.

## Containers run identically every time

One of the main benefits of containers is that whenever a container is run, the workflow or application it contains will behave identically. That is to say; containers provide reproducibility. Reproducible means we have a container X that gives an output Y; every time it is run, now, in five minutes or two months, for the same input, it will give identical output.

The second main benefit of containers is that wherever a container is run, it will behave identically, containers provide portability. Portable means the container will run the same on our computer, your colleague's computer, and the cloud. No risk of removed dependencies, lost configuration files, or other changes that break our application.

## Isolation

This is possible because of isolation between the container and the rest of the environment; running a container will have no impact outside of the container and vice versa. Anything happening outside the container will not affect the result of a container. A container has limited resource access to the operating system it is running on; everything else is kept separate.

## Containers provide security

Because containers are completely isolated from each other, even if the security of one container is compromised, the other containers on the same host, and the host itself remain secure since there is no direct communication between containers. The compromised container still only has access to limited resources on the host and nothing more. This makes containers not only great for safely deploying applications but also for quickly prototyping workflows. You can be sure that whatever you do in the container won't affect anything outside of it and that you can start with a clean slate at any point.

## Containers are lightweight

Containers not only provide security, portability, and reproducibility. An additional advantage of containers is that they are lightweight, or in other words, use few extra resources in comparison to running an application outside of a container. Containers have little overhead compared to alternatives that also provide isolation. This is especially relevant when comparing containers to virtual machines, which we will do in detail later on.

## Containers and data science

All these advantages make containers relevant for data science; containers make any task or workflow automatically reproducible not only on our own machine but also everywhere else. Containers help us avoid many issues when sharing our work; dependencies are automatically included, and so are datasets. Most importantly, we can be sure our code will work on our colleague's machine. Additionally, the lightweight nature of containers makes them easier to share than alternatives.

# Docker

Docker is an open-source tool that allows us to create, run and manage containers. Docker first launched in 2013, and even though containers had existed for more than a decade, it was with the launch of Docker that containers exploded in popularity.

## Docker ecosystem

Over time, Docker has grown to be part of a large ecosystem of tools around containers; we will focus on Docker Engine, which is everything you need to create, run and manage containers. Other parts of this ecosystem are, for example, Docker Compose, a tool for defining and running multi-container Docker applications, and Kubernetes, a system for container scheduling and management.

## Docker Engine

Docker Engine has two main parts: server and client. The client, called Docker client, is a command line interface used to talk to the server. The server is a background process that requires no user interaction, which is called a daemon, a term we will encounter repeatedly to reference the Docker server. In addition to the Docker client and daemon, Docker Engine also includes so-called API specifications, which define how you can interact with the Docker Daemon. These APIs are not only used by the Docker client to talk to the daemon but also specify how other applications can work with the daemon.

## The Docker daemon

The Docker daemon is responsible for managing all Docker objects, such as images, containers, and more. However, we can't directly tell the daemon what to do; we need a client to give us a human-usable interface to it. Here the Docker command line interface is the default option, but there are others, like Docker Desktop, which gives us a Graphical User Interface to manage our containers.

## Images and Containers

The daemon manages both images and containers, but there is a difference between them. While an image is a blueprint or recipe, like an idle copy of a hard drive with all the software we want to run, a container is a running image, like a copy of that same hard drive plugged into a running computer. We could create an image with Ubuntu and python3 point 9 installed. Once we start this image, we'll have a running container with Ubuntu and python3 point 9 where we can execute our code.

## Containers are processes

Up until now, we talked about containers abstractly. To better understand containers, we can think of them as processes. When we start a container, a process is started, just as when we start a text editor, Spotify, or any other application. What makes a container process different is its permissions to resources like the file system, memory, and network.

For many resources of a container process, not only is access restricted, but they are also undetectable to the process. For example, instead of seeing all the files on your hard drive, the process is given access to only a single folder and cannot see files outside of that folder.

## Containers are isolated processes

Not only blocking access to but also hiding resources may seem like a small difference, but it allows running a process that is isolated from the rest of the machine. This lets an entirely separate operating system run inside the process. A container runs its own operating system instead of using the host operating system. The Docker daemon ensures that the OS running in the container is unaware of other containers and the host OS. The operating system inside the container can start and manage it's own processes without interfering with any processes running on the host OS. In other words the operating system in the container is separated and kept unaware of anything happening outside itself, isolated from anything on the host and other containers.

# Containers and Virtual Machines

Both Virtual Machines and containers aim to run software side by side on the same physical machine safely, that is, without interfering with each other. In that sense, Virtual Machines achieve many of the same goals as containers. However, from a technical perspective, there is a big difference making their use cases different.

## Resource Virtualization

Running software side by side on the same physical machine safely is done using virtualization. Virtualization means that resources like RAM, CPU, or Disk can be split up and look like separate resources to the software using them. For example, a hard disk of 100GB can be virtualized to look like four hard disks of 25GB. This way, different pieces of software can each use 25GB, yet they can't interfere with the other parts. Both containers and VMs are virtualization technologies.

## Containers vs Virtual Machines

The key difference between containers and virtual machines is that virtual machines virtualize the entire machine down to the hardware. Whereas with containers, their virtualization happens in a software layer above the operating system level. This means separation in VMs is better as only the hardware is shared, while for containers the host operating system is also shared.

## Security of Virtualization

This better separation of VMs over containers makes them more secure and points us to the main drawback of containers, that there is always a possibility for attackers to get access to the host OS. This, in turn, can give access to all containers running on the same machine. Since attackers breaking out of a container to the host operating system is the main risk of using containers, Docker and other container providers spend extensive resources on making their containers as secure as possible. The risk of attackers accessing the host is limited when using an industry-standard container provider like Docker. Nonetheless, it is worth considering VMs when security is paramount, for example, when working with sensitive data.

## Containers are lightweight

While containers have a slight disadvantage in the amount of security they provide, there are several advantages of using containers over VMs. One significant advantage is their size in memory and on disk compared to VMs. In other words, containers require less RAM and less disk space. Containers are significantly smaller because they only need to include a small part of a full OS, sharing the rest of the OS with the Host OS and other containers.

## Advantages of containers

The smaller size is at the base of many advantages of containers. It makes containers faster to start and stop. And also makes containers faster to distribute and to change or update. Because of their small size, there is a large ecosystem of pre-made containers with many popular software applications like programming languages, databases, or web servers pre-installed. In comparison, VMs can quickly become several GigaBytes in size, which means they are often built from scratch for every use case.

## Advantages of Virtual Machines

Of course, slightly better security is not the only advantage of VMs. If your use case needs a Graphical User Interface, then for now, a VM is the best option; no container supports GUI applications, while VMs support both GUIs and command lines fully.

# Running Docker containers

## The Docker CLI

Let's get started with the Docker command line interface or CLI. The CLI will send our instructions to the docker daemon, which is responsible for managing our containers and images. After receiving our command, the daemon will reply with the result, which we'll see in the terminal. The CLI command for Docker is simply `docker`. Before we can start a container, we have to choose which image to start it from. We need an image because images act as blueprints or recipes from which we can start containers. Images define what will be installed and available when the container is started. The container is then a running instance of the image, which we can interact with. For example, there exists an image called ubuntu that contains the full Ubuntu OS. Once we start a container from that image, we have a fully running Ubuntu OS that we can interact with using a shell.

## Docker container output

If we want to start a container from an image, we can use the docker run command, followed by the image-name. For example, to start the hello-world image, we would use Docker space run space hello dash world. By default, Docker starts a container and shows you the container's output while it's running. When we run a hello-world container, it prints an explanation of how the container works and then stops.

When you run the docker run command with an image, Docker will first check if the specified image already exists locally on your machine. If the image is not present locally, Docker will attempt to pull the image from the specified image repository, which is typically a container registry like Docker Hub, Amazon ECR, Google Container Registry, etc. Docker will then download the image and store it locally on your system before starting a container based on that image.

If you don't specify a tag for the image, Docker will assume the "latest" tag by default. For instance, if you want to run a container based on the Ubuntu image tagged with "20.04", you would use: “docker run ubuntu:20.04”.

## Choosing Docker container output

When an image is created, the creator can choose what will happen when a container is started from the image. For example, the creators of the hello-world image choose to print out text and then make the container stop itself. Another example is the Ubuntu image, which contains a full Ubuntu OS. When starting an Ubuntu container, it will start and then shut down immediately without printing any output. Its creators decided that for their image, it didn't make sense to do anything specific by default. The Ubuntu OS starts, and it stops again without any output.

## An interactive Docker container

Instead, the Ubuntu image is intended to be used with the dash it flag. Using 'docker run dash it' followed by an image-name, we can start a container and simultaneously get an interactive shell in this container. If we do this with the Ubuntu image, using 'docker run dash it space Ubuntu', we end up in a new shell inside the new container. The shell gives us access to a clean Ubuntu environment isolated from our host machine because it is running inside the container. Once we want to exit the container, we simply use the exit command, which returns us to the host machine and then stops the container.

## Running a container detached

We saw a container that just prints text and one that makes more sense to use interactively. A third type of container processes data or can be interacted with in some way externally, for example, a container with a database like Postgres or a data processing script. These are run using Docker run dash d, for detached, followed by the image-name. This makes containers run in the background without printing their output to our shell.

## Listing and stopping running containers

After the container has started, the docker ps (process status) command allows us to see it and any other running containers. The first column contains the container-id, a unique id identifying each container. The next step is stopping containers we don't need anymore. This can be done with the 'docker stop' command. To stop the desired container, we pass the container-id to the docker stop command.

If you want to see all containers (both running and stopped), you can use the -a or --all flag:

# Working with Docker containers

## Listing containers

When there are only a few containers, it's easy to find them in the list that docker ps returns. However, if you're working with lots of containers, it can quickly become challenging to identify the right one.

## Named containers

To solve this, the docker run command has a name flag that allows us to name a container. The name then shows up in the last column of the docker ps output. Here we created a container from the Postgres image and called it db underscore pipeline underscore v1. For any commands that require us to specify a container, we can use either the container-id or container name. For example, in exactly the same way we could stop containers using their id, we can use their name in docker stop.

## Filtering running containers

When using docker ps with so many containers that even naming them doesn't allow us to find them. We can use docker ps with the f, for filter flag to find a specific container. For example, docker ps dash f followed by, in quotes, name equals sign db underscore pipeline underscore v1. This will show you only the details of containers with the name you specified in the filter.

## Container logs

Now that we know how to find our running containers, it will also be useful to see their output, for example, to debug any issues. To look at the output a container has generated, we can use the docker logs command followed by the container id. Most containers quickly generate a lot of output, so you will often have to scroll through the result of docker logs to find what you're looking for.

## Live logs

If instead, you want to follow the logs your container is generating in real-time, you can use docker logs together with the f, for follow, flag. You will see any logs the container generates live. Even though docker ps also has a f flag, the docker ps flag allows us to filter. When working with docker logs instead, the f flag has another effect, allowing us to follow a container its logs. After using docker logs, you will see the output of a running container until either the end of the logs or until you press control plus c to exit the log view.

## Cleaning up

Previously, we learned how to stop containers. However, a stopped container is not fully gone; the stopped container still exists and is occupying some space on our hard drive. To fully remove an already stopped container, for example, because we want to reuse its name, we use docker container rm followed by the container-id to remove the container.

# Managing local docker images

## Docker Hub

Docker Hub is a registry of community-made Docker images. In other words, it's a website from which we can download thousands of pre-made images for all kinds of use cases. For any common use-case, we will find an image on Docker Hub.

## Pulling an image

Downloading an image from Docker Hub is called pulling an image. The command to pull an image is docker pull, followed by the name of the image you want to download. For example, you can download the hello-world, postgres, and ubuntu image from Docker Hub using docker pull. When pulling an image using just the image name, we will always get the latest version available of the image.

## Image versions

We can find older versions of an image on hub dot docker dot com. The example you see here is for the ubuntu image. To pull a specific version, we use docker pull image-name followed by a colon and then the image-version. A version can be a number, some text, or a combination of both. For example, if we want the 22 dot 04 release of ubuntu, we can use docker pull ubuntu colon 22 dot 04 or colon jammy.

## Listing images

Now that we know how to pull images, we need a way to view the images we have available on our machine. For this, we have the docker images command, which will give us a list of all available images and tags. It will also tell us when the image was created, the size of the image on disk, and the image-id, which is a unique id to identify the image.

## Removing images

Docker only has a limited amount of space it can use on our disk. Previously we saw how to remove containers using docker container rm. Similarly, we can use docker image rm to clear space for more containers and images. A container is a running image; a side effect of this is that you can only delete an image once there are no more containers based on it. If we try to delete an image for which we still have a container on our system, we'll get the warning you can see at the bottom of the slide. This error message also includes the container's id based on the image we're trying to remove. We can use the docker container rm command to remove the container, after which we can remove the image.

## Cleaning up containers

It's common to have multiple containers based on a single image, which can make it a tedious task to one by one remove all containers before you can remove an image. To more easily clear all stopped containers, we can use docker container prune.

## Cleaning up images

Then we can use docker image prune dash a to remove all unused images. The a flag, which stands for all, makes it so that unused containers are removed and not only dangling images.

## Dangling images

A dangling image is an image that no longer has a name because the name has been re-used for another image. This frequently occurs when creating our own images. For example, if we create an image called testsql, but we find a mistake and change our image slightly, the previous testsql image will then become dangling as our new fixed image now has the testsql name.

# Distributing Docker Images

In the previous lesson, we learned about the official Docker images registry and how we can pull images from it. This is just one way images are distributed. Now, we will learn how to share images with just a few people or a broader community. Either by sending docker images like we would send any other file or using a Docker registry server.

## Private Docker registries

First, we'll have a look at private Docker registry servers. The Docker organization maintains the official Docker images registry. Other Docker registries work the same way but are under the control of another person or group. This means there are no guarantees that the images will work or are safe to use. Images from any registry other than the one from the official Docker organization are easily recognizable because their name starts with the URL of the private Docker registry they come from. For example, the image we see here comes from dockerhub dot myprivateregistry dot com. Downloading or pulling an image from a private registry is also done using Docker pull. Because the image name includes the Docker registry URL, the image will automatically be pulled from the correct registry. For example, to pull the previously mentioned image, use Docker pull followed by the full name of the image. Here too, we can append a colon followed by the version.

## Pushing to a registry

Now that we know how to pull images from a custom registry, let's look at how to upload or push an image to a Docker registry. The command for this is Docker image push followed by the image name. To push an image to a specific registry, we only have to make sure the image name starts with the name of the registry we want to push to. We can do this by renaming the image using Docker tag. For example, if we have the image classify underscore spam colon v1 and we want to push it to docker dot myprivateregistry dot com, we would use Docker tag followed by the image we'd like to rename and then the desired new name including the url of the private registry. After renaming the image, we can use Docker image push followed by the image name to push the image to our private registry.

## Authenticating against a registry

While the Docker official images can be pulled without authentication, anybody creating a private Docker registry can make it private and require authentication. The standard way to secure private Docker registries requires users to log in. We do this with the Docker login command followed by the private registry we want to authenticate for. For example, Docker login dockerhub dot myprivateregistry dot com, the URL we authenticate for should be the same as the URL we put in front of an image name.

## Docker images as files

If, instead of using a Docker registry, we want to send a Docker image to just a few people, it can be easier to send the image as a file. Use the save command to save a Docker image to a file. This will create a minimized file which we can then share like any other file. We can pass the desired filename to the save command using the minus o flag. To load the file, we use the load command, passing the filename using the minus i flag.

# Creating your own Docker images

## Creating images with Dockerfiles

Docker images are the recipes or blueprints for Docker containers. To create this blueprint, we must write down a list of instructions in what is called a Dockerfile. A Dockerfile is a text file containing all the commands we would run in the command line to install the software we need, with the addition of some Docker-specific syntax. Conveniently this file should be called Dockerfile for Docker to be able to find it.

## Starting a Dockerfile

Just like when we would follow a recipe, Docker runs the lines in a Dockerfile from top to bottom. The first line in a Dockerfile is always the FROM instruction. This instruction indicates to Docker which image to start from. We can base our images on any other image, Postgres, Ubuntu, another image you made yourself, or even the hello-world image. As with pulling an image, if you want to start from a specific version, you can specify the version right after the image name, separating both with a colon.

## Building a Dockerfile

With the FROM instruction, we can create the most basic Dockerfile; we can then create an image from this Dockerfile using the Docker build command. The docker build command is followed by the location of the Dockerfile we want to build. If our Dockerfile is in the current folder, this is simply a dot. When running Docker build, in the last line of the output, we can see the id or hash docker assigns the new image. The hash starts by indicating its type, sha256, in this case. This is followed by the unique hash, which starts with a67f for the example on the slide.

## Naming our image

If we want to give our image a more recognizable name, we can use the t for tag flag followed by the name we want to give our image. If we also want to give a version to our image, we can add a colon and the version after the image name. In both cases we end the docker build command with a dot indicating our Dockerfile is in the current working directory. Once Docker has successfully built our image from our Dockerfile, we can run and use our image just like the images we downloaded from Dockerhub.

## Customizing images

Now that we can create a very basic image from a Dockerfile, the next step is to start customizing our image. To customize our Dockerfile we will use the RUN instruction. The RUN instruction allows us to run any valid shell command while building an image. To make an image that runs a python data analysis, we start from the ubuntu image, which has Ubuntu installed, by specifying the FROM ubuntu instruction followed by RUN apt-get update. Apt-get is a package manager which enables us to install all kinds of software. The apt-get update command we just added to our Dockerfile will update apt-get so it knows what the most up-to-date version is of all the different software it can install for us. Using another RUN instruction on the following line, we download python using RUN apt-get install python3. Like we can see at the bottom of the slide, some bash commands require user input. While a Docker image is building it is not possible to manually give any input to the bash commands docker runs. Instead we can pass the dash y flag to apt-get install to make sure it doesn't need any input.

## Building a non-trivial Dockerfile

Once we add RUN instructions to our Dockerfiles, we'll notice that building a Dockerfile can take seconds to sometimes tens of minutes because Docker is actually running the commands specified with RUN. For example, building a Dockerfile with FROM ubuntu and RUN apt-get update, will take the same time as us running apt-get update on ubuntu, which is 2 seconds for the example on the slide.

# Learning to add files in your image

## Copying files into an image

The RUN instruction allowed us to execute bash commands to create an image, but we can't use it to move files from our local file system onto the image we're building. To copy local files to our image we use the COPY instruction. The COPY instruction needs two parameters: first, we pass to it the path of the file we want to copy, including the name of the file we want to copy. The second parameter is the destination path inside the image. We can choose whether to end the destination path with a filename. If we do not pass a filename, the file will get its original name.

## Copying folders

If we don't specify a filename in the source path, then instead of just a single file, the entire contents of the folder will be copied, including sub-folders. For example, if we have a folder called pipeline underscore v3 with two files and a sub-folder with one file, we can copy both files and the subfolder with its file using the COPY instruction ending in pipeline v3 slash.

## Copy files from a parent directory

It is not possible to copy files from a parent directory when building a Dockerfile. For example, let's say we are in the projects folder when we run docker build. A COPY instruction in the Dockerfile that tries to copy the init dot py file from the parent directory of the current directory into the image will fail with the not found message we can see on the slide.

## Downloading files

Another common way to include files in an image is to download them during the image build. While there is an instruction that allows us to do this, the ADD instruction, best practice is to use several RUN instructions and bash commands to download and unzip files. First, use curl to download a file to a local directory. Then unzip it using the unzip command if it is an archive. Finally, once we don't need the zip file anymore, we can remove it with the rm command.

## Downloading files efficiently

Any instruction in a Dockerfile that downloads files will add to the size of the image. Even if the files are removed in a later instruction. To ensure images don't become unnecessarily big, we should download, unzip and remove the original file in a single RUN instruction. This can be done by chaining the commands using a backslash and double ampersand. The backslash makes it so bash commands can span multiple lines allowing us to keep our Dockerfile readable. The double ampersand tells the shell to execute the commands one after the other. Combining them allows us to create a single RUN instruction that is still easy to read over multiple lines. By using this best practice on downloading and unpacking archives, we ensure our image is as small as possible, making it easier to share and faster to run.

# Choosing a start command for your Docker image

## What is a start command?

When we start a container from the hello-world image, the container will start, print text, and then stop. The creators of this image chose a start command that prints text and then exits. This makes sense in the context of a hello-world image.

It wouldn't make sense for the creators of an image with the goal of running python to do the same. Instead, it would be useful for their image to start a python session for the user automatically and also to exit the session once that python session is stopped. Docker images have this flexibility; using the CMD instruction we can choose any shell command to execute when a container is started from the image.

## Running a shell command at startup

Like everything else we add to images, we can add a start command using an instruction in the Dockerfile. This instruction is CMD. CMD accepts a single parameter, the shell command to run when the image starts. The shell command runs when somebody starts a container; it is not executed when using docker build to create an image from the Dockerfile. Adding a CMD instruction to a Dockerfile does not increase the image size and does not add any time to the build. If multiple CMD instructions exist in a single Dockerfile, then only the last one will have any effect.

## Typical usage

When building an image for a specific use, it makes sense to set the CMD instruction to start an application related to this particular use. This could be starting a python based data analysis or starting software like a database that accepts outside connections. Another typical pattern is to run a script at startup that starts multiple other applications.

## When will it stop?

With the CMD instruction, we set a shell command to run when a container is started from the image. The container will stay running until this shell command exits. The hello-world image only prints text and exits right after doing so. An image that starts a database by default will stop if the database is stopped or crashes. The more general the use case of the image, the more flexible the start command should be. For example, if we make a general usage Ubuntu image, simply opening a shell would make sense as a start command. The image will then exit when the user exits the shell.

## Overriding the default start command

While the CMD instruction sets a default start command for the image, this default start command can be overridden when starting an image using the Docker run command. Just like we pass the image we want to start to Docker run, we can pass a second optional argument, which will override the CMD instruction set in the image. Often when replacing the start command of an image, we will run the image in interactive mode, using the 'dash it' flag. Using a shell as new start command, for example, bash, allows us to look around the image, discover files and see what's installed.

# Introduction to Docker layers and caching

## Docker build

When learning about the RUN instruction, we saw that building images can take significant time because the shell commands are actually run when the image is built. This is necessary because what is saved in the resulting image is not the instructions in the Dockerfile but the changes in the file system the instructions make during the build. For example, if we have a RUN instruction with a shell command that downloads and opens a zip file with several files inside, the resulting image will contain those files.

## Docker instructions are linked to File system changes

During the build, an image keeps track of which instructions in the Dockerfile created which change to the file system. We can view an image as a list of consecutive changes to the file system, with every entry in the list corresponding to a specific Docker instruction in the Dockerfile.

## Docker layers

Together all the changes to the filesystem for a single instruction in the Dockerfile are called a layer of the image. All layers together make up a Docker image. There is also some metadata, like the start command set with the CMD instruction. While building a Dockerfile, we can clearly see when docker is working on the next layer and on which layer it is working from the output. Like at the bottom of the slide, Docker will print which step it is building out of the total amount of steps whenever the next instruction in the Dockerfile starts building.

## Docker caching

When building a Dockerfile that we have built before, in front of each layer being built, it says cached in capital letters. Docker detects which Dockerfile instructions have not changed, and instead of re-running the Dockerfile instruction, it uses the known result it has stored. Docker will only use cached layers to speed up our builds if the Dockerfile instruction is exactly the same and all previous Dockerfile instructions are also identical to when it originally created and stored this layer.

## Understanding Docker caching

Understanding when Docker layers are cached when building images is important for two reasons. First, it will help us understand why sometimes our image stays the same even though we change and rebuild it. For example, if we have a RUN 'apt-get update' and 'apt-get install python3' instruction in our Dockerfile, and a new version of python3 is released. Rebuilding our Dockerfile will not change anything in the resulting image. Docker will see the same instructions as when it last built this Dockerfile and will assume that the result is the same. It can not know that re-running apt-get update will give another result.

Understanding when Docker layers are cached is also important because it will help us write Dockerfiles, which we can make changes to without all layers having to be rebuilt. This can be done by ordering the instructions in the Dockerfile from least changing to most changing.

This is why we always want to put the Dockerfile instructions to install packages before instructions to copy files into the image. Often we'll change the files when improving our work or fixing bugs, but the packages we need rarely change. This ensures as many cached layers as possible can be re-used.

# Changing users and working directory

## Dockerfile instruction interaction

The FROM, RUN, and COPY instructions only affect the file system, not each other. If we copy a start dot sh file from our local file system into an image, we can then use the RUN instruction to execute this file. The two instructions didn't change each other's behavior directly, but both used and changed the file system. However, some instructions can influence other instructions directly. The WORKIDR instruction changes the working directory instructions are executed in, and the USER instruction changes which user is executing the following instructions.

## WORKDIR - Changing the working directory

When using a Dockerfile instruction where we have to specify a path, we can always use a full path. For example, a path that starts at the root of the file system, like in the first example on the slide. When working with long paths, this can quickly become hard to read. The alternative to using full paths is the WORKDIR instruction, which allows us to change the directory inside the image from which all subsequent instructions will be executed. For the COPY instruction, we change the current path on which relative paths will be based.

## RUN in the current working directory

Like with the COPY instruction, other Dockerfile instructions are influenced when the working directory is changed with WORKDIR. This includes the RUN and CMD instructions. The effect on the RUN instruction is straightforward. The shell commands executed by the RUN instruction will be run in the directory set by WORKDIR. This allows us to make the RUN instructions more readable and removes any unclarity about where the files we are running are located.

## Changing the startup behavior with WORKDIR

The WORKDIR instruction also changes the working directory in which the shell command of the CMD instruction is run. If a user of the image overrides the CMD, their replacement start command will also be run in the path set with WORKDIR.

## Linux permissions

What you can do in a Linux operating system or OS depends on your permissions. Your permissions, in turn, are set by assigning you a user. For example, a data science user could be allowed to access the datasets folder while other users are not. There is a unique user called the root user, which has all permissions on the system. Best practice is to use the root user to create one or more new users and only give these users the permissions required for specific tasks. Then we should stop using the root user and use these better-scoped users instead.

## Changing the user in an image

When writing Dockerfiles, we should follow this best practice and not run everything as root. The image we start our Dockerfile from will determine the user. For example, the ubuntu image uses the root user by default. Any RUN instructions we put in a Dockerfile starting from ubuntu will be run as root. This has the advantage that all folders are accessible, and we won't get errors about permissions when installing anything. However, it is unsafe as all instructions will run with full permissions. The USER Dockerfile instruction allow us to change the user in the image. Any following instructions will be run as the user set by the USER instruction. It can be used multiple times, and the latest instruction will determine the user executing the following instructions.

## Changing the user in a container

The USER instruction changes the user with which the following instructions in the image are run. The last USER instruction in a Dockerfile will also control the user in any containers started from the image of this Dockerfile.

# Variables in Dockerfiles

## Variables with the ARG instruction

First, we will look at the ARG instruction. The ARG instruction allows us to set variables inside a Dockerfile and then use that variable throughout the Dockerfile. It is followed by a space then the name of the variable we want to create, an equal sign and the value of the variable. Later commands can then reference this variable using a dollar sign followed by the variable name. However, it can only be used in the Dockerfile, the variable won't be accessible after the image is built. In other words, if you define a variable with ARG in a Dockerfile, build an image from that Dockerfile, and then start a container from that image, that variable will not exist inside the container.

## Use-cases for the ARG instruction

Typical use cases for the ARG instruction are to define a version we need in multiple places throughout the Dockerfile. Like in the first example on the slide, we specify a version of Python called bionic compiled for Ubuntu. Defining a path to a project or user directory is also helpful as an ARG. This allows us to make any instructions using this path less verbose and makes it more apparent at a glance that all files are going to the same folder.

## Setting ARG variables at build time

The ARG instruction can also be set in the docker build command, giving us even more flexibility. At the top of the slide, you see the same example Dockerfiles as on the previous slide. By using the build dash arg flag when running 'docker build', we can set another value for the project-folder variable, which overrides the original value during that build.

## Variables with ENV

The second way to define variables in Dockerfiles is by using the ENV instruction. The syntax is identical to the ARG instruction, but unlike the ARG instruction, variables set with ENV are still accessible after the image is built. While variables set with ARG are used to change the behavior of Dockerfiles during the build, variables set with ENV are used to change behavior at runtime.

## Use-cases for the ENV instruction

Typical use cases are setting variables used by applications when they are starting, like database directories or users - or setting an application to production or development mode. Unlike ARG variables, it is not possible to override ENV variables at build time. However, it is possible to override ENV variables when starting a container from an image. This can be done using the dash env parameter of the docker run command. For example, in the official postgres image, there are several ENV variables available to configure the container.

## Secrets in variables are not secure

Both ENV and ARG variables seem convenient for adding passwords or other secrets to a docker image at build or runtime. However, both are not secure to use for secrets. Anyone can look at variables defined in a Dockerfile after the image is built with the docker history command. This command shows all the steps that were done to build an image. If, instead, we pass variables at build or start time, they can be found in the bash history of the host or image. The bash history is a list of all shell commands executed by a user. Keep in mind that if we use secrets to create our image without using more advanced techniques to hide them, they will be shared with anybody we share the image with.

# Creating Secure Docker Images

## Inherent Security

Docker inherently provides more security over running applications locally because there is an extra layer of isolation between the application and our operating system. This makes it much safer to open an application or archive from an unknown source in a container in comparison to doing the same on your local machine. However, that doesn't mean it is 100% safe to do so. A malicious payload can escape the container's isolation and infect the host.

## Making secure images

Attackers breaking out of a container to the host operating system is the main risk of using containers. Docker and other container providers spend extensive resources on making their containers as secure as possible. Additionally, there are several things we, the creators and users of images and containers, can do to make both more secure. The safety measures we'll be discussing next might seem like they won't do much if we're just sharing images with colleagues or using them to run workloads locally. However, a widespread use case for images is running them on remote machines and allowing external access. For example, to run a database or a pipeline in a production environment. It is in those scenarios that the following safety measures become critical.

## Images from a trusted source

The first step to creating a secure image is choosing the right image to start from. Anybody on the Internet can provide images for us to use or build on top of. However, using images from an untrusted source is a security risk. The official Docker Hub registry provides thousands of images and allows the filtering of Trusted Content in three different ways. All three Trusted Content filters will give us images we consider safe for the most use-cases.

## Keep software up-to-date

Even images downloaded from the official Docker Hub Repository aren't always up-to-date. Applications release updates all the time, and even operating system updates aren't incorporated into images the minute of their release. In the slide, you can see the extremely popular Docker Official Images Ubuntu and Mariadb, which were updated two weeks and a month ago. While it could be the case no safety-related updates have been made to anything installed in these images since then, best practice is to update the software to its latest version in images ourselves.

## Keep images minimal

What's better than ensuring all software in our image is updated? Having less of it. There is no safer piece of software than one we haven't installed. When creating a secure image, ensure you only install the software you need for its current use case. This also means we will have to keep less software up to date.

## Don't run applications as root

All previous measures are of little use if we allow anybody who gets access to a container to install anything they want. The solution is not to leave the user in our images as root. Often it is needed to install and configure applications as root; after that, the user in our image should be changed to a user with fewer permissions. If, for example, we change the user before the CMD instruction that starts our pipeline, we ensure that any malicious code in the pipeline does not have root access in our container.

# Summary

## Theoretical foundation

In chapter one, we saw that containers are portable computing environments containing everything needed to run a workflow or application. They provide security, portability, and reproducibility. Then we dove deeper into Docker specifically by learning about Docker Engine, which is everything we need to create, run and manage containers. Wrapping up the chapter, we saw why the lightweight nature of containers made them gain popularity over virtual machines. To understand why this comparison is valuable, we first had to learn about virtualization, which allows us to run software isolated from each other but on the same hardware.

## The Docker CLI

Chapter two is where we finally got our hands dirty. We went from starting our first container to running containers in several different ways, looking at container logs, managing several containers, and cleaning everything up again. Once we knew how to work with containers, we could learn more about images, where to get them, how they are versioned, and how we can share them with others.

## Dockerfiles

After seeing how to manage images, it was time to build our own in chapter three. Creating Docker images is done using Dockerfiles and specific instructions made for exactly this goal. We saw all the essential instructions allowing us to specify an image to start from, run shell commands, copy files, and more! At the end of chapter three, we looked in more detail at Docker layers, giving us insight into how Docker creates images and how we can optimize our images and keep them small.

## Security and Customization

In this last chapter, we learned about four Dockerfile instructions that allow us to configure our Dockerfile and images in more complex ways, for example, by setting a user or creating variables that can be configured either while building an image or starting a container.

To wrap up, we went over some best practices of container security, start from a trusted image, keep images up to date, only install the software you need, and don't run applications with the root user.

## What more is there to learn?

Docker is part of a larger ecosystem of tools around containers and there is much more to learn. There are several more less-used Dockerfile instructions like ENTRYPOINT and HEALTHCHECK, among others, which are still very useful in the right circumstances. Still, in the context of Docker Engine, there is even more to learn, like how to start an image from scratch instead of continuing from an existing image or multi-stage builds, allowing you to create Dockerfiles built on top of multiple other images at once. Other topics not touched upon in this course are networking and volumes, respectively, allowing you to connect containers to a network and access local or saved files in a new way. And last but not least, there are several tools, like Kubernetes and docker-compose, to orchestrate containers. docker-compose allows you to define how a few containers interact, for example, a python container running a pipeline together with a database image the pipeline can connect to. While Kubernetes allows us to start, stop and do anything else imaginable from code.