HELLO DOCKER!

What is Docker?

- Long time ago, we installed software on an OS that is installed on a particular set of hardware devices (CPU, memory, disks…etc.)
- Then came virtualization: the ability to run multiple OS's on the same hardware by creating some form of hardware distribution and isolation. But this was still a resource-intensive process.
- Docker fills the gap between having to run a totally separate OS (like on a virtual machine) to run a software application, and to run that application on the same OS.
- It creates a lightweight level of virtualization using **containers**.
- It is an open-source, Linux-based software that can run on any hardware environment that supports Docker instances (Windows, Linux, OSX).

Containers…?

- Since the old days of UNIX, system administrators used *jails* to isolate processes in "closed environments". A closed environment is a special runtime space for the process/application to live in, where it cannot access any protected resources.
- The term "container" was popularized by the release of Solaris 10, which used "Solaris Containers" to provide even more isolation than jails. A container denies the process access to any system resource except the ones explicitly granted.
- Containers were complex and hard to set up. Accordingly, users either misconfigured them, or simply could not use them at all.
- Docker solves this problem. It uses existing container engines to build containers according to the best practices. This process does not require user intervention, putting the container technology in the reach of everyone.

Containers vs. Virtualization

- A container is not a virtual machine.
- A virtual machine is built by abstracting the necessary hardware resources to run a whole instance of an operating system. Afterwards, you can install your software on that instance.
- The problem with this model is that a lot of resources get wasted just to run the OS instance (not just the software application).
- When the software is run inside a Docker container, it runs in an isolated environment that interfaces directly with the Linux kernel of the main OS (the one you are currently running). There is no need to install a totally new instance. This saves time and reduces costs.

Docker images

- A Docker container is where the application is physically stored, together with its dependencies. This can be used to ship, copy, distribute, and run applications.
- But a container must be "filled" with something before it can be shipped. This is were Docker images come into play.
- A Docker image is the basis of containers. It is a snapshot of all the files that a program running inside a container should need.
- An image can be used to create as many containers as needed. Thus, when you want to ship an application "inside Docker", you ship the image and the recipient OS creates containers out of that image.
- To make things easier, images are stored in registries and indexes. Those may be publicly available (on the Docker website or other hosting companies). You can also host your own indexes and registries.

Common Docker use cases

Software dependencies

- You have a computer running some version of Centos
- You installed a business application that uses glibc library version 2.12
- Everything runs smoothly until the security team advises you that you must upgrade the Linux kernel to apply the latest patches.
- The kernel update naturally upgrades glibc libraries
- Once you upgrade the kernel, the business application fails to start anymore.
- You find out that the prerequisites of the application do not support glibc of a higher version than 2.12
- Unfortunately, you cannot downgrade glibc libraries without downgrading the kernel itself, which might mean a complete OS reinstallation.
- You can easily overcome this problem by installing the software application in a Docker container. This will ensure that the application will be completely isolated from your own system libraries and resources (like glibc). Now you can upgrade the kernel without worrying about breaking anything.

Software portability

- You have an excellent piece of software that solves a business problem.
- For some reason, this software runs on Linux (and only Linux).
- The companies servers are running Microsoft Windows.
- As a workaround, you install Linux on a virtual machine in one of the servers.
- Not only does this consume a lot of resources, but also creating a second and third environments (like for testing or QA) means more and more VM's and more and more wasted resources.
- Using Docker, you can run exactly the same software on any system. On a Windows or a MAC machine, Docker only needs to create one small virtual machine on which multiple containers can be built using much less resources than creating multiple VM's.

Software-running risks

- More often than not you need to install applications from "untrusted" sources. That doesn't necessarily mean that they are malicious, but you can't take the risk.
- There are many ways in which a process can misbehave. System corruption can be the least worse. It may contain bugs that allows outside attackers to compromise the system or it might leak confidential data.
- Running the application inside Docker ensures that it gains access to only the resources that you declaratively allow it to.
- Accordingly installing untrusted software for the sake of testing or other purposes is much safer when used inside Docker than when risking your system.

When *not* to use Docker?

- Throughout our discussion, we shed some light on the merits of Docker, the advantages of containers and the different scenarios where Docker can be of great value. But Docker is not a silver bullet. There are some situations where Docker cannot (or should not) by used:
  - Currently, Docker is used to run programs that are designed to run on a Linux kernel. That means that a native Windows or OSX application cannot be shipped by Docker to other systems.
  - Some applications (especially security-related ones) need to be run as a privileged user with full administrative capabilities. Docker – by definition – is used to create an isolated environment, where the account used is a limited one.

A Docker example

- It's about time to see some code running! First, install Docker on your machine. Depending on the OS you are using, the instructions will differ. In this course we'll be using Ubuntu for the examples, but you can use another Linux distro, Microsoft Windows or MAC OSX. Installation instructions can be found on this link: https://docs.docker.com/installation/
- Once installed, run the following command (if you are on a Linux or a MAC you may want to use sudo to run the command with elevated privileges): `docker run -i -t ubuntu echo 'Hello World'`
- The program will download several components, it may take some time depending on your Internet connection speed. Eventually, it prints "hello world" to the screen.

What has happened?

- Issuing Docker run will start a new container. As mentioned, a container needs an image from which to get its instructions (required files, dependencies and so on).
- The image in our example is `ubuntu` (also called repository).
- Docker searches for this image on locally on your computer. Since it doesn't find it, it contacts Docker Hub (which is a large, publicly available repository of images). When if finds that image, it starts downloading it.
- After the download completes, the image layers get installed locally on the computer.
- Eventually, Docker uses the installed image to create a new container and start the program.
- The command issued by this sample container is `echo "hello world"`. Once the command has finished, the container is automatically stopped.
- Notice that the state of the container is bundled with the state of the command: if the command is running, the container is running, if the command stops, the container stops. If the container is restarted the command is launched again.
- If you try to issue the `docker run` command again a couple of things will happen:
  - Docker will find an image locally installed on the computer so it won't download it again
  - It will create a *new* container for the command to run in. each time you run Docker run, a new container is created, which is completely isolated from other containers, although based on the same image.