Docker

Docker is a tool that packages all your project's code and dependencies in an executable container that can be saved and reused. Docker allows you to run your application from anywhere as long as you have docker installed on that machine. Developers use Docker to eliminate the "works on my machine" problem when sharing code with co-workers.

Docker containers run within an isolated environment on top of the host's operating system. Operating systems like Ubuntu, Mac, Windows consist of an OS kernel and a set of software. The OS kernel interacts with the underlying hardware, whereas the software is what makes the Operating systems different. Docker requires a linux OS kernel, thus in order to run Docker an a non-linux system (Windows) a lightweight virutal machine (VM) with a mini-Linux Guest OS is required.

Docker containers are more lightweight (less resources) than VM. However, virtual machines are more isolated (do not share the OS and run independently of the host's OS). Thus, often a combination of both Docker and VM is used.

DockerHub

On Dockerhub you can share and download Docker images in order to run containers on your machine.

Basic Terminology

Virtual Machine: VMs run software applications inside a guest Operating System, which runs on virtual hardware powered by the server's host OS.

Container: An isolated environment which contains all the required dependencies for you application to run. A container is created based on a container image. Containers are not persistent. They can include data, however the data is also not persistent. To make data persistent you have to use volumes.

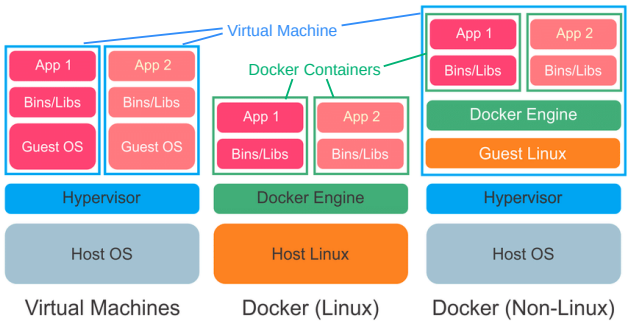
Docker Image: An image is a blueprint for building a container. It contains everything you need to recreate the original environment (code, runtime, system tools, system libraries, settings).

Dockerfile: a yaml-based file that is used to build your image

Volume: Local data that is linked inside a container

Docker Engine: The system that lets you create and run Docker containers.

Docker Compose: Lets you run mulit-container images



Installation on Ubuntu

There are three different ways to install Docker. In the following the recommended approach is shown. More info on https://docs.docker.com/engine/install/ubuntu/

Docker is not in the official Ubuntu 18.04 repositories, thus you have to manually create one.

```
sudo apt-get remove docker docker-engine docker.io \
   containerd.io
```

Unistall older versions of Docker

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl \
   gnupg-agent software-properties-common
```

Updatde the package list and install some necessary packages

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
   sudo apt-key add -
```

Add Docker's official GPG key

```
sudo apt-key fingerprint 0EBFCD88
```

Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

```
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) stable"
```

Set up the stable repository

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Update the apt package index, and install the latest version of Docker Engine and containerd

```
sudo docker run hello-world
```

Verify that Docker Engine is installed correctly

Uninstall Docker Engine

```
sudo apt-get purge docker-ce docker-ce-cli containerd.io
```

Uninstall the Docker Engine, CLI and Containerd packages

```
sudo rm -rf var/lib/docker
```

Delete all Images, containers and volumes

Basic Docker Commands

```
docker --version
```

Print the version of docker

```
docker run -it --rm -p 8888:8888 jupyter/scipy-notebook
```

Run a Jupyter notebook within a container (just copy the URL into a browser). Remove the container after exiting. The -p 8888:8888 is required for jupyter notebook.

```
docker ps -a
```

Lists all running and exited docker containers

```
docker images
```

Lists all images that you have offline on your computer

```
docker exec -it [container_ID] bash
```

This lets you open up a bash shell in an already running container (run this command in a separate window).

```
docker start -i [container_ID_or_name]
```

Restart a container that has been exited and that have not yet been removed. The -i stands for interactive mode.

```
docker stop [container_ID_or_name]
```

Stops running container.

Remove Images and Containers

```
docker rm [ID_or_Name]
```

Remove specified container

```
docker rm $(docker ps -a -f status=exited -q)
```

List and remove all exited containers (the images are not removed)

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

Stop and remove all containers

```
docker rmi [ID]
```

List all images and remove the specified images

Dockerfile

The dockerfile is the “recipe” that contains everything to build an images. A Dockerfile is literally a file called “Dockerfile” on your system. Most often you start a Dockerfile based on another container (eg. FROM ubuntu)

First create a new folder on your system and inside this folder create a new file called Dockerfile (no .txt or anything).

```
# Dockerfile

# use anaconda3 as base image
FROM continuumio/anaconda3:latest
```

Use Anaconda3 as the base image. The part after the colon tells docker which version should be used (in this case the latest one).

```
# copy requirements file
COPY requirements.txt /project/
```

Copy the requirments file from the current directory on the local machine into the directory project inside the container.

```
# set the working directory (inside the container)
WORKDIR /project/
```

Set the working directory to project

```
# install dependencies
RUN pip install -r requirements.txt
```

Install the python dependencies

```
# copy the project into the container
COPY . /project/
```

Copy the project from the local machine into the project folder of the container

```
CMD jupyter-notebook --ip=0.0.0.0 --allow-root
```

The CMD indicates that the default command to run in the container should be jupyter notebook. This will create a link to open a jupyter notbook inside the local browser.

Build Docker Image

Go to into the folder which contains the Dockerfile.

```
# bash terminal (host machine (the system that is running the container))
# pull the base image from docker hub
docker pull continuumio/anaconda3
```

Before building the image we can first pull (from Docker Hub) the base image used in our docker file. This is not absolutely necessary.

```
# build the image
docker build -t [name_of_image] .
```

Create a new image with the tag [name_of_image].

The “.” tells Docker to use the Dockerfile from the current directory

The dot at the end just tells Docker to look in the current directory for the Dockerfile.

Run a Python Script

How to run a python script inside a container: First create the Dockerfile (inside the directory that contains the python script), then build the image, and finally run the container. In the following different possibilites are described.

In order to use data from your machine inside the container you need to connect a volume to the container using the -v parameter.

```
# mount volume (-v) and directly run script inside the container (run python and the script)
docker run -it --rm -v `pwd`:/project [name_of_image] python [script_name]
```

This creates a container from the image [name_of_image] and runs it in interactive mode (-it). Python is run and script called [script_name] is executed.

Additionally, the current folder on the local machine is mounted onto the container using the -v flag. Like this, if files are changed, deleted or added inside the container, the changes are also visible on the local machine.

Without the -v [host_path]:[container_path] the local file system is not affected by changes inside the container.

The --rm flag tells Docker to remove the container when exiting it. Without this flag the container can be restarted (continue where you stopped).

```
# directly run script inside container (run python and the script) without mounting volume
docker run -it [name_of_image] --name [container_name] python [script_name]
```

Create a container from the image called [name_of_image] and run the container in interactive mode (-it). Run python and execute the script called [script_name].

The container is not removed when exited (no –rm).

The name [container_name] is assigned to the container using the --name flag.

Changes are not saved to the local file system (no -v [host_path]:[container_path])

```
# start bash session from which the script can be run
docker run -it --rm [name_of_image] /bin/bash
```

The container is run in interactive mode and a bash shell is opened. Exit the container with the command exit.

```
# bash terminal (host machine)
# open a new terminal inside the container
docker exec -it [container_ID_or_name] bash
# stop a container
docker stop [container_ID_or_name]
# restart a containers
docker start -i [container_ID_or_name]
# remove a container
docker rm [container_ID_or_name]
```

Some basic command to (1) open a terminal, and to (2) stop, (3) start and (4) remove a container.

Run a Development Session

Open a Jupyter notebook. Additionally, we do also need to add a data Volume to the container in order to keep the changes that we make.

You can also open a bash terminal inside the container to pip install further packages. If install additional packages don’t forget them to add them to the requirements.txt

```
# bash terminal (host machine)
# run a jupyter notebook
docker run -it -p 8888:8888 -v `pwd`:/project --name [container_name] [image_name]
```

Build a container from the image [image_name] and connect a volume to the container. The data in the current folder (`pwd`) will be available inside the container at /project (due to the -v flag). The -v is used as follows: -v [host_path]:[container_path]. Changes will affect the data outside the container.

The host directory (eg. on local machine) can represent a git repository. You can commit and push code either from the host or from within your container.

The container is called [container_name].

The -p 8888:8888 option specifies that we want to map port 8888 in the container to port 8888 on the host.

Just copy the URL to open the Jupyter notebook: there you can add new files or remove or change existing files.

```
# bash terminal (host machine)
# open a bash shell inside the container
docker exec -it [container_name] bash
```

Opens a bash shell inside the container, so that you can pip install new packages

```
# bash terminal (container)
# install tensorflow
pip install tensorflow==2.2.0
```

Pip install version 2.2.0 of tensorflow inside the container. If the container is stopped and started again, tensorflow is still available inside the container.

However, if the container is removed, and a new container is created, tensorflow needs to be installed again. Thus, we need to add tensorflow to the requirements file.

```
# bash terminal (container)
# update requirements file
pip freeze > requirements.txt
```

This saves all dependencies of the running container in the requirements.txt file. As we used the -v flag, the requirements.txt is also updated on the local machine.

The make the changes permanent the image need to be rebuild, in order to contain the new dependencies

```
# bash terminal (local machine)
docker build -t [name_of_image] .
```

Build a new image. Containers created from the new image will contain the new dependencies.

GPU for Docker

In order to run GPU accelerated Docker containers you need to have installed the **NVIDIA driver** and **Docker 19.03** on your Linux distribution. You do not need to install the CUDA toolkit on the host, but the driver needs to be installed.

Singularity: Docker requires root privileges and is thus often not possible on HPC. Singularity is a tool designed to run containers on HPC. It supports Docker containers and images. Singularity limits user privileges and access from within the container, making it safe for users to bring their own containers.

Installation

In order to run GPU accelerated Docker containers you need to have installed the **NVIDIA driver** and **Docker 19.03** on your Linux distribution. You do not need to install the CUDA toolkit on the host, but the driver needs to be installed.

```
# bash terminal (host machine)
# Verify Nvidia driver installation on host
nvidia-smi
```

Get the nvidia driver version.
You can install the nvidia dirver with **sudo apt install nvidia-xxx** (xxx is the version).

```
# Add the package repositories
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -

curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | sudo tee /etc/apt/sources.list.d/nvidia-docker.list

# install nvidia container toolkit
sudo apt-get update && sudo apt-get install -y nvidia-container-toolkit
# restart docker
sudo systemctl restart docker
# test nvidia-smi
docker run --gpus all nvidia/cuda:10.1-base-ubuntu18.04 nvidia-smi
```

Install nvidia container toolkit and check if docker has access to the GPU

Dockerfile (for GPU access)

In order to run GPU accelerated Docker containers you need to have installed the **NVIDIA driver** and **Docker 19.03** on your Linux distribution. You do not need to install the CUDA toolkit on the host, but the driver needs to be installed.

```
# use ubuntu18.04 with CUDA10.1 as base image
FROM nvidia/cuda:10.1-base-ubuntu18.04

# install anaconda3 (dockerfile from continuumio/anaconda3)
ENV LANG=C.UTF-8 LC_ALL=C.UTF-8
ENV PATH /opt/conda/bin:$PATH

RUN apt-get update --fix-missing && apt-get install -y wget bzip2 ca-certificates \
    libglib2.0-0 libxext6 libsm6 libxrender1 \
    git mercurial subversion

RUN wget --quiet https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh -O ~/anaconda.sh && \
    /bin/bash ~/anaconda.sh -b -p /opt/conda && \
    rm ~/anaconda.sh && \
    ln -s /opt/conda/etc/profile.d/conda.sh /etc/profile.d/conda.sh && \
    echo ". /opt/conda/etc/profile.d/conda.sh" >> ~/.bashrc && \
    echo "conda activate base" >> ~/.bashrc

RUN apt-get install -y curl grep sed dpkg && \
    TINI_VERSION=`curl https://github.com/krallin/tini/releases/latest | grep -o "/v.*\"" | sed 's:^..\(.*\).$:\1:'` && \
    curl -L "https://github.com/krallin/tini/releases/download/v${TINI_VERSION}/tini_${TINI_VERSION}.deb" > tini.deb && \
    dpkg -i tini.deb && \
    rm tini.deb && \
    apt-get clean

ENTRYPOINT [ "/usr/bin/tini", "--" ]

# copy requirements file
COPY requirements.txt /project/
# set working directory
WORKDIR /project/
# install dependencies
RUN pip install -r requirements.txt
# copy the project
COPY . /project/
# open jupyter notebook
CMD jupyter-notebook --ip=0.0.0.0 --allow-root
```

Use base image with CUDA10.1. Then install anaconda3 and setup the environment.

Run GPU-accelerated container

```
# bash terminal (host machine)
# build image
docker build -t [name_of_image] .
# run container
docker run --gpus all -it --rm -p 8888:8888 [image_name]

# open bash terminal inside the container
docker exec -it [container_name] bash
```

Build the image and run the container using all available gpus

```
# optional
# bash terminal (container)
# install pytorch inside the container
pip install torch==1.5.0+cu101 torchvision==0.6.0+cu101 -f https://download.pytorch.org/whl/torch_stable.html

# start python inside container
python
>> import torch
>> torch.cuda.is_available()
```

Verify that GPUs are available inside the docker. The toch.cuda.is_available() should

return TRUE.

https://data-ken.org/docker-for-data-scientists-part3.html#workflow3

Install docker-compose (Ubuntu)

https://docs.docker.com/compose/install/

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.26.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

download the current stable release of Docker Compose

```
sudo chmod +x /usr/local/bin/docker-compose
```

Apply executable permissions to the binary

docker-compose up