TP - Introduction to Docker
ING3-IMSI – Cloud for Data Analytics
2019–2020

1 Introduction

The objective of this hands-on exercise is to get familiar with containers creation, management and distribution. You will learn how to install software into a container, create an image and make it available to other users. First of all, as it was explained in the course slides, make sure that you have docker installed in your laptop and that your Docker network is properly configured to be used at EISTI.
Once docker is installed, open a CLI (Command Line Interface) and type

```
$ docker run hello-world
```

This command will download a container. You should see a welcome message and a message explaining the process that has been followed to execute the container.

2 First contact with a container

As explained in the course slides, a container is created from a base image that uses the host kernel. On top of this image we will add additional layers that will contain our modifications. Docker takes care of the whole process.

Start an ubuntu container:

```
$ docker run -it ubuntu /bin/bash
```

Now, perform the following actions:

1. Compare the kernel version in the host (your laptop) and in the container.
2. Check the Docker image and its size.
3. Compare the container filesystem to the host filesystem. Use `df -h`. Try creating a file on the container. Does it exist in the host machine?
4. Stop the container, start it again and attach your terminal to it. Check that it still contains the modifications you made to it (like the file you created inside).
5. Finally, remove the container (but not the image!).

3 Creating an image from a customised container

Let us install an application in a container and create a new image out of it.

1. To start a GUI-based application, we need first to allow all connections to the host graphical server. To do so, execute '`xhost +`' **on the host**. Then, stop the container
2. Start a new container by exporting the DISPLAY variables:

   ```
   $ docker run -it -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
                ubuntu /bin/bash
   ```

3. Install firefox in your container. Do not forget to run first `apt-get update`.
4. Execute firefox from your container by just typing '`firefox`'.
5. Exit your container. Save your container as a new image called `myfirefoximage` using `docker commit`.
6. Delete your container with `docker rm`. Make sure that you can start a new container from your `myfirefoximage` image. You should be able to start Firefox from it.
7. You can see that there exist many applications already containerised that can be executed directly. For example, download an image of the *sublime-text-3* text editor with `docker pull jess/sublime-text-3`. Execute it. Check the difference between running the container with and without the `/bin/bash` parameter at the end.
8. Check Docker man pages and remove with a single command line all containers based on the `jess/sublime-text-3`.
9. Check the size of the `myfirefoximage` image. Then, remove it with `docker rmi`.

4 Encapsulating services on containers

In this section, we are going to install and configure a web development environment and make it accessible to users and developers.

1. From the `ubuntu` image, run an interactive container and name it `'myphpserver'` (use the option `--name`). Expose the port 80 in the container with the address 127.0.0.2 and port 8080 in your host. To do so, use the option `-p`. Check the manual pages to find out its syntax.
2. Install the Apache2 and PHP servers (packages `apache2`, `php` and `libapache2-mod-php`) in your container. Start the apache2 service.

3. Open a browser in your host and point it to `http://127.0.0.2:8080`. Check that you can access the web server. The default directory in your container from where web pages are served is `/var/www/html`. Write a `phpinfo.php` page with the following content and check from the host browser that PHP works as well:

   ```
   <?php
   phpinfo();
   ?>
   ```

4. Sometimes we will not develop our website directly on the container, but we will have already some files that we want to copy into the container. To copy files from our host to the container (and vice-versa) we can use the `docker cp` command (check the man pages). Ask your teacher for a compressed file containing a web site. Download it to your host and copy it to the container. Decompress it into the container in the right directory and make sure that it works when you access to it from the host browser.

5. At this point, we will be happy with our container. Commit it as an image with the name `myphpserverimage`. Stop the running container.

6. Whereas it is possible to copy files from and to a container, this is not always the best idea if we want to make sure that the code developed will be portable when used inside a container. The best choice would be to have a directory shared between the host and the container that allows us to edit the code that we are testing in our container server. One possible solution is to use ***shared volumes***: attached data volumes that are mounted in a directory both in the host and in the container. Any changed made to a file by the host or the container in such directory will be visible to the other one.

   Run a new container called `myphpserver2` from your `myphpserverimage` with the same options as the `myphpserver` container, but add also the option `-v` (do you remember the `-v` option in the Firefox exercise?):

   ```
   -v /myhost/source/code/directory:/mycontainer/source/code/directory
   ```

   Now you will be able to edit and develop your code on the host `/myhost/source/code/directory` directory (with any IDE of your choice), and the code will be accessible by the container on the path `/mycontainer/source/code/directory`. However, take into account that **volumes are not saved to images** after a `commit` so, if you want your code to be available in future containers spawned from your image, you must copy them from the container volume mounted directory to any different one inside the container (for example,

into `/var/www/`). We will see later that there are more flexible ways of handling this "feature" of Docker containers.

7. Commit again the `myphpserver2` container to the same image and stop the container.
8. Finally, let us suppose that we want to share the code in our container to edit it with the IDE of our choice, but this IDE is also containerised. As an example, we are going to use a container from our image `myphpserverimage` and the editor from the `jess/sublime-text-3` image.

   First create a container from your `myphpserverimage` using the `docker create` command. Call it `myphpserver3` and add the option `-v /var/www`. What is the difference between `docker run` and `docker create`?.

   Then, start a container from the `jess/sublime-text-3` image. Add the option `--volumes-from myphpserver3` so that it can mount the `/var/www` directory from `myphpserver3` as its own. Check that you can edit the files on `myphpserver3` from sublime-text.
9. Commit again the `myphpserver3` container to the same image and stop the container.

5 Exporting images

Once we are happy with the image we have committed, there are several reasons why we might want to share it: it may be useful for the community, we want to have it accessible online or we want to submit it as a deliverable to our teacher, to cite some.

1. Let us upload our image to the Docker Hub, the official repository that Docker provides to its users. Go to `https://hub.docker.com/` and follow the instructions to sign up. Once your account is validated, log on and create a new repository called `webserver` with private visibility.
2. From your host command line, add a *namespace* and a *tag* to your `myphpserverimage` image. If your login is `dockerhublogin`, run:

   `$ docker tag myphpserverimage dockerhublogin/webserver:latest`
3. Then, again from your command line, login to your Docker Hub account:

   `$ docker login --username=dockerhublogin --email=myemail@mydomain.com`
4. Finally, send your image to the Hub:

   `$ docker push dockerhublogin/webserver`

From now on, your image will be available on Docker Hub and can be downloaded by anybody with the link.

5. There exists an alternative way to save an image and store it offline as a `tar` file. Run the following command:

   `$ docker save -o webserverimage.tar myphpserverimage`

   Now you have a webserverimage.tar file in your filesystem that you can store and share offline. To load it back in Docker, use the `load` command:

   `$ docker load -i webserverimage.tar`

6 Conclusion

— Check your knowledge and the acquired skills by watching the following video: https://youtu.be/PivpCKEiQOQ