SOFTWARE PACKAGING

LAB 01: Packaging a sample application

- In this lab, you are going to build a very basic application that will just create a new file. To achieve this, we are going to do the following: create a container from an existing image, modify the filesystem of the container to fit our needs, and finally commit those changes. Once done, you'll be able to use this image to create new containers.

```
docker run --name base_container busybox touch /welcome.txt
docker commit base_container
base_imagesha256:b3452858612206 86b0cebc038e98e6ced1b297f6ba
5286c5f3eb4857aaee01f2
docker rm -vf base_container
base_container
docker run --rm base_image ls -l /welcome.txt
-rw-r--r--    1 root     root             0 Aug  6 08:49
/welcome.txt
```

- The procedure is very simple: you run a container out of a busybox image, passing in a command that will create a new file called /welcome.txt. Then you use the commit command to create an image out of this container. This image will have – by default – a file called /welcome.txt whenever a container is created out of it.

LAB 02: Example package: installing Python

- In this lab we'll move a few steps deeper. Using the previous method, we are going to create an image that has Python installed in it.
  `docker run --name python_base ubuntu /bin/bash`
  This is pull an Ubuntu latest image and run a container. Then it will immediately drop you to the bash shell.
- Inside the shell type the following commands:
  `apt-get update`
  `apt-get install python`
  The first one updates the software sources used by Ubuntu OS and the second actually installs Python.
- You can make sure that the installation was successful by running `python --version`, which should print the current install Python version (2.7 at the time of writing this).

Determining changes

- In the previous lab, we stopped at installing Python necessary packages. Now what if several days later we want to determine what changes have been made to this container before committing it to an image?
- Docker provides the `docker diff` command for that purpose. You can run it as follows:
  `docker diff base_container`
- The resulting output contains all the file that were changed in the container. You will find one of the following letters on the left of each file:
  A : Added
  D : Deleted
  C : Changed
- You can repeat the LAB 01 in this section, and before committing to a new image you can do all sorts of file manipulation (adding, deleting, and modifying) and see the output of `docker diff` command before committing the container to a new image to see those changes.

Committing to a new image

- Let's revisit the `docker commit` command. It can do more than just creating a new image out of the modified container.
- You can (and should) use –m to specify a message describing the changes that have been made to the container before creating a new image.
- You may also want to add your own signature using –a option.
- Let's use those options and commit our changes to create an Ubuntu image that contains Python:
  ```
  docker commit -a "@afakharany" -m "Installed Python 2.7"
  base_container ubpython
  ```
- You can test your new image using the following command:
  ```
  docker run –it --rm ubpython python --version
  ```
  it should report Python 2.7

The entrypoint configuration

- In the previous example, you successfully created an Ubuntu image that contains a Python installation by default.
- Now how would your users benefit from it? If they simply run the docker run command without any commands following it, the container will exit as soon as it starts.
- This is because the default entrypoint command for Ubuntu is to execute /bin/bash and exit when the command returns successfully.
- We need our container to do something more useful by modifying the entrypoint of the container. Consider the following:

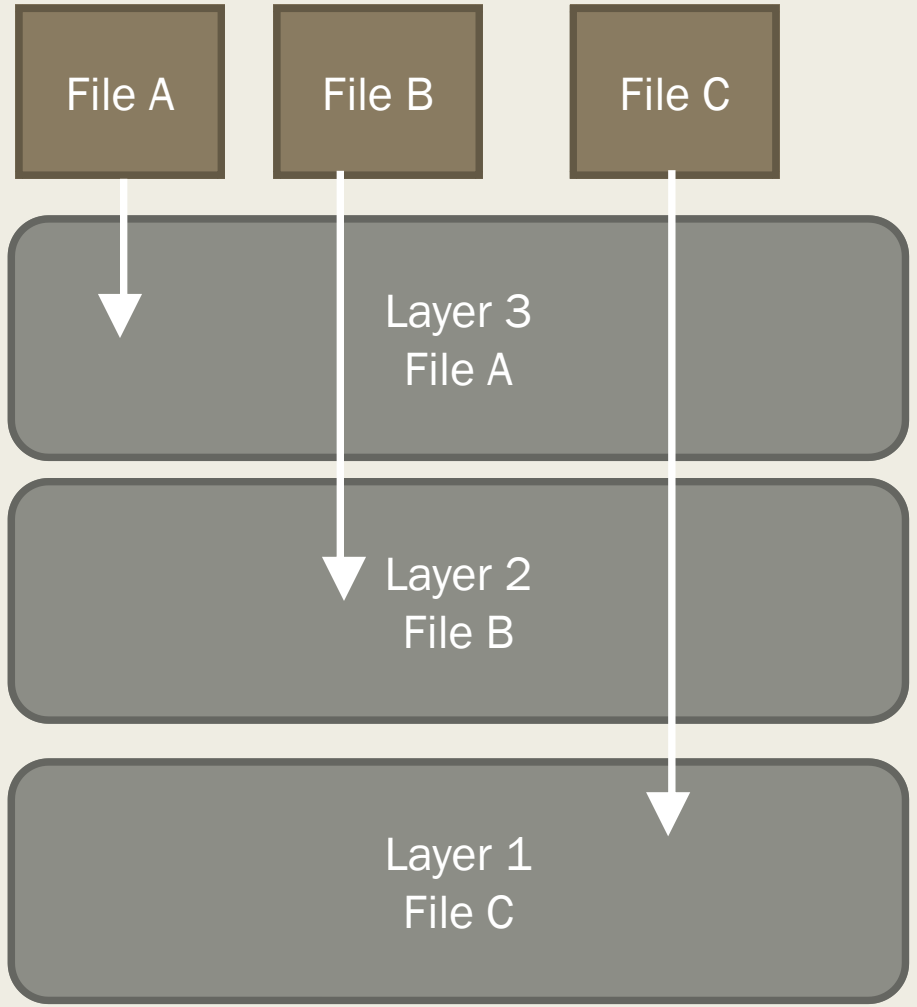
```
docker run --name python-int --entrypoint python ubpython
docker commit -m "Change entrypoint" -a "@afakharany" python-
int ubpython
docker run -it --rm ubpython
[Inside Python interpreter]
print "Hello Docker!"
```

The commit command options

- When committing a container to an image, you are not just transferring the union file system, but also environment variables, the current working directory, any exposed ports, volume definitions, the entrypoint, and the command and its arguments.
- If you do not explicitly set those values, they will be inherited from the basic image.
- The following example sets some of those variables at creation time:
```
docker run -it --name shared -e MYVAR=Hello busybox
docker commit shared test-image
docker run --rm -it --name cont1 test-image echo MYVAR
```
You should see "Hello" as the result of printing the MYVAR environment variable.
- Now let's add some more options to the image by creating a second container and committing the changes it makes:
```
docker run --name cont2 --entrypoint "/bin/sh" test-image
docker commit cont2 test-image
docker run -it --rm test-image
# echo $MYVAR
```
- In the previous example, we created a second container, cont2 that was based on the modified busybox image, test-image. We altered it's entry point to be /bin/sh. Then we committed this container to test-image, which added up to its inherited options. To prove that we run echo $MYVAR in the shell prompt to display the value of the inherited environment variable.

The UFS (Union File System) revisited

- We briefly discussed the UFS when we introduced volumes. A UFS works by representing several layers of files to the client. But how does it work? Consider the following figure:
  - *When a file us created, it is placed on the top most layer in UFS.*
  - *As more and more files are added, more and more layers contain files.*
  - *When you need to access a file, the system searches UFS from the top-most layer downwards until it finds the file and serves it.*
  - *When more than one file is requested, they get served from different layers at the same time. That is, the file system "unions" the files from different layers, hence the name.*



Understanding file operations in UFS

- As shown in the previous figure, files are dispersed on layers in UFS. But what happens when a file is changed or deleted?
- If a file is deleted, that deletion occurs in the top most layer hiding any other version of that file in a deeper layer. For example, let's delete /etc/localtime from a test container and see what docker diff brings:

```
docker run  --name test busybox rm /etc/localtime
docker diff test
C /etc
D /etc/localtime
```

  A change happened in /etc and a deletion happened on /etc/localtime.
- Let's try changing the file instead of deleting it:

```
docker run  --name test2 busybox touch /etc/localtime
docker diff test2
C /etc
C /etc/localtime
```

  This time, a change occurred on /etc then another change on /etc/localtime
- Union File Systems applies changes to file using a technique called "copy-on-write". This means that when a file located in some read-only layer (not the top-most one) gets changed, this file is copied first to the writeable layer before any changes can be applied. Of course this negatively affects the image size as well as performance,

The commit command and UFS

- Any changes made to the UFS file system are written to the top-most layer, in addition to the metadata of that layer.
- The `commit` command saves any changes done to the image (like when we added the Python package), the top layer is saved in a way so that it can be identified.
- A layer metadata contains the unique identifier, the identifier of the layer below it, and information about the container that was used to create the layer.
- The group of layers combined together (according to dependencies) form the image from which a container can be created and run.
- The image id is also the id of the top most layer. This is also the ID that gets printed when the `commit` command is used.
- This image id is used to create and run containers. But since they hard for humans to read, repositories are used for this purpose.

Image tags

- Repositories have been already discussed in a previous section. You can use them to easily search for the image that serves your needs by name and tag.
- Let's have a second look at the command that was used to write create the ubpython image:

```
root@DockerDemo:~# docker commit -m "install python" -a "@afakharany"
python-int ubpython
sha256:13972d204c2ec93db486addfa064ad42a75c67053061a9630623cc034793f2
fb
root@DockerDemo:~# docker commit -m "install python" -a "@afakharany"
python-int afakharany/ubpython
sha256:a015dc6866a8f94a4a19aab77362f14001c044e12304ef9797c7ecc666fac8
7a
root@DockerDemo:~# docker commit -m "install python" -a "@afakharany"
python-int afakharany/ubpython:v1
sha256:54ed4d035c33012cde0598f5342220157a69371b1e6fdf33cae13311c540c5
0b
```

- As you can see, the image id changed when we changed the image name to be inside a repository (afakharany), and when we added a tag. Actually the ID will change whenever any change happens to the image; as this change will always be reflected to the outermost layer.
- Tags can be added to the image at built time, or when committing changes, or even afterwards using the docker tag command. For example:

```
docker tag afakharany/ubpython afakharany/ubpython:v1
```

Image layer sizes and limits

- Image layers have a very interesting property: they are immutable. This means they cannot be modified.
- When you make any change to an image and commit that change, a new layer is added to the top of UFS containing that change. The positive side of this is that layers can be shared easily. The negative size of this, however, is that the image size keeps increasing. Accordingly, care must be taken when making changes to images.
- For example, let's say that you want to modify the image that we previously created by removing Python from it. This can be done as follows:

```
docker tag afakharany/ubpython:latest afakharany/ubpython:v1
docker run -it --name base_container  afakharany/ubpython:latest /bin/bash
# apt-get remove python
# apt autoremove
# python --version
 bash: /usr/bin/python: No such file or directory
exit
docker commit base_container afakharany/python
docker images | grep python
afakharany/ubpython       latest              b8246cc93bf7        6 seconds ago       197.9 MB
afakharany/ubpython       v1                  fdd6ff55abc3        2 minutes ago       197.6 MB
```

- Notice from the previous output how the image size increased after we remove Python. This is because UFS does actually delete files; it places other files on the top layer that shadows files in layers underneath.
- Another important limitation to take care of is the layer limit. This is a number indicating the maximum amount of layers that can be created on an image. You can have a look at the available layers using the command `docker history`. This command shows the layer id (abbreviated), its age, the command that used to create it, and the total file size of the layer.
- A possible solution of this limit, and in order not to make a huge image size, you can "branch" your images.
- Branching simply means that you select one version of the image, run a container out of it and commit to a new image name (or different tag). This will ensure that the image size will not get unnecessarily high (because not all the changes are committed to the same image), and also you will avoid the layer limit problem.

Working with flat file systems

- You can work directly with the underlying UFS file system of any image without having to create a container specifically for that by importing and exporting the image file system.
- Consider the following example:

```
docker run --name myContainer afakharany/ubpython:latest
docker export --output myContainer.tar myContainer
docker rm export-test
tar -tvf myContainer.tar
```

- The above commands will create a TAR file containing the contents of the image. You can also output the contents of that TAR ball to the standard output if you want to chain it to other shell commands by omitting --output (-o for short).
- You can import a flat-file image into a new image by using docker import command. It can accept data from a web URL or from the local file system. It can also accept tarred and compressed files (.tar, .tar.gz among others). For example you can import that above image using a command like the following:

```
docker import – myImage < myContainer.tar
```

notice here the use of the dash (-), instructing Docker to accept data from the standard input.