Docker Lab 1
Working with images

1 References and cheat sheets

The official reference for all Docker commands:
https://docs.docker.com/reference/cli/docker/

https://dockerlabs.collabnix.com/docker/cheatsheet/
https://devopscycle.com/blog/the-ultimate-docker-cheat-sheet/
https://spacelift.io/blog/docker-commands-cheat-sheet
https://www.geeksforgeeks.org/docker-cheat-sheet/

2 Commands covered

Searching for images	Pulling images

`docker search image-name`
`docker search --filter "key=value" image-name`
 	
https://www.tutorialworks.com/find-docker-images/
	`docker image pull image-name`
	`docker image pull -a repo-name`
	
	https://www.educba.com/docker-pull/

https://www.configserverfirewall.com/docker/search-images/	

Listing and filtering images	Getting detailed info on images

`docker image ls`
`docker image ls REPO[:TAG]`
`docker image ls --filter=reference=image1`
`docker image ls --filter "key=value"`
`docker image ls --format go-template`
 	
https://techtutorialsite.com/list-docker-images/
	`docker image inspect image-name/id`
	
	`docker image inspect -f go-template image-name/id`
	
	https://adamtheautomator.com/docker-inspect/

Deleting images	Tagging images

`docker image rm image-name / image-id / image-digest`
`docker image rm -f image-name / image-id / image-digest`
`docker image rm -f $(docker image ls -q)`
 	
https://www.digitalocean.com/community/tutorials/how-to-remove-docker-images-containers-and-volumes
 	
https://www.freecodecamp.org/news/how-to-remove-images-in-docker/
	`docker image tag` *`source-image target-image`*
	
	https://blog.atomist.com/docker-image-tags/
	
	https://www.freecodecamp.org/news/an-introduction-to-docker-tags-9b5395636c2a/

Logging into a remote registry	Pushing images
`docker login`	`docker push user-name/repo-name:tag`
https://www.techrepublic.com/article/how-to-successfully-log-in-to-dockerhub-from-the-command-line-interface/	https://www.section.io/engineering-education/docker-push-for-publishing-images-to-docker-hub/

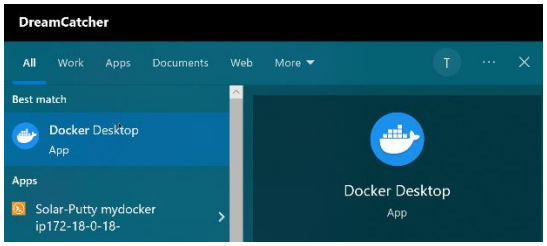
	https://jsta.github.io/r-docker-tutorial/04-Dockerhub.html

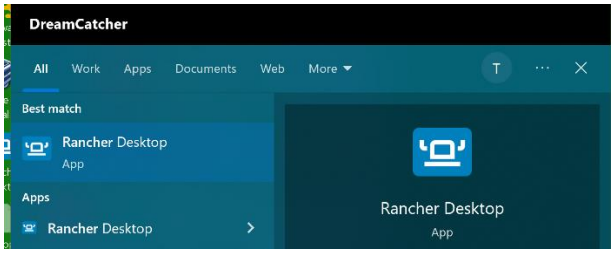
3 Lab setup

Make sure you already have an account on DockerHub (https://hub.docker.com/)

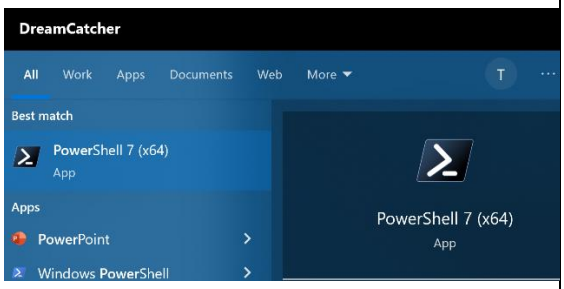
You should have an appropriate software installation to work with Docker containers on your machine: this could be either Docker Desktop, Rancher Desktop or some other suitable Docker runtime.

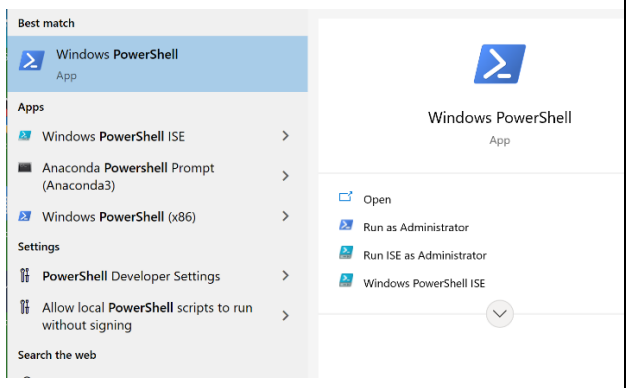
Start up any one (**but not BOTH !**) of these applications via Windows start menu.





If you are working on a Windows machine, start up either a PowerShell 7 terminal or Windows PowerShell terminal to type in the various Docker CLI commands. Although it is possible to type Docker commands via the normal command shell, this is not as ideal as it doesn't provide many of the PowerShell features that make it easier to work with Docker.





To check the Docker version and to also quickly verify that the Docker client and daemon (engine) is running type:

```
docker version
```

Normal output would look similar to below:

```
Client:
 Cloud integration: v1.0.35+desktop.5
 Version:           24.0.6
 API version:       1.43
 Go version:        go1.20.7
 Git commit:        ed223bc
 Built:             Mon Sep  4 12:32:48 2023
 OS/Arch:           windows/amd64
 Context:           default

Server: Docker Desktop 4.25.0 (126437)
 Engine:
  Version:          24.0.6
  API version:      1.43 (minimum version 1.12)
  Go version:       go1.20.7
  Git commit:       1a79695
  Built:            Mon Sep  4 12:32:16 2023
  OS/Arch:          linux/amd64
  Experimental:     false
….
…
….
```

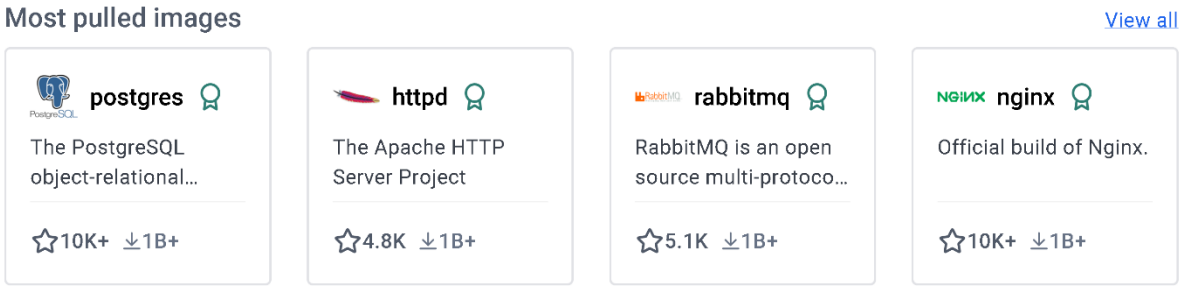
If the Docker daemon (engine) is not available, you will get an error message similar to the following:

```
error during connect: This error may indicate that the docker daemon is not running.: Get
"http://%2F%2F.%2Fpipe%2Fdocker_engine/v1.24/version": open //./pipe/docker_engine: The system
cannot find the file specified.
```

4 Searching for images on Docker Hub

Docker is configured by default to retrieve images from the public Docker Hub registry: however, you can configure it to retrieve images from your own private registry if you wish. Go to Docker Hub, and click on Explore to get a list of the images available (which will be sorted based on popular categories and trending).

Scroll down the list look for most pulled images and select View All.



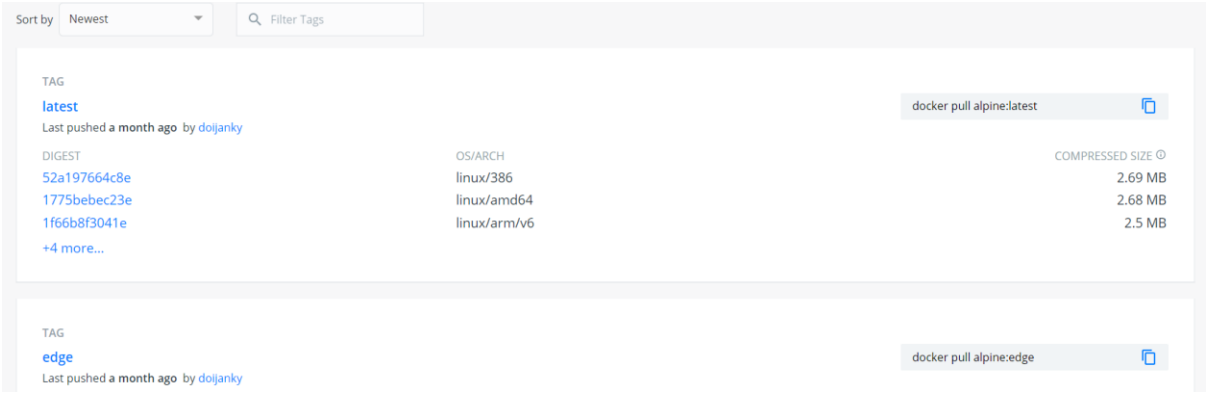
You should be directed to a page with 2 filters applied: right now you are viewing only official Docker images. These are a curated set of Docker repositories hosted on Docker Hub whose images provide essential base repositories that serve as the starting point for the majority of users.
https://docs.docker.com/trusted-content/official-images/

There are many popular Linux distro base images here such as Ubuntu, Debian, Fedora and Alpine.

Scroll down and look for Alpine and click on it.
Notice the format of the URL for the home page for this image. This indicates that it is a top-level or official repository:
https://hub.docker.com/_/alpine
You can scroll through and get more information on this particular image. Alpine linux is a very small and simple Linux distro that contains many useful packages, utilities and tools that are frequently associated with Linux-related work, but with a much smaller resource footprint compared to larger Linux images such as Ubuntu.

All Docker images are identified by image name which are essentially a combination of the repository they are located in and their corresponding tag in the format: `repo:tag`. In this case, the repo name is `alpine`.

Click on the Tags or View Available Tags link to see all the Tags available for this particular repo. In the upper right hand corner of each Tag category, you will see the docker command to download that particular image for that particular tag (for e.g. `docker pull alpine:3.20.2`). There is a listing for the different types of OS/arch supported for that particular image, and the digest for that image is also shown (this is known as the manifest list).



Typically, most official repos will have a `latest` tag which usually (but does not always) represents the latest image version in that repo.

If you click on a specific tag (for e.g. `latest`), you will get a page showing the digests for all the different images for the supported architectures, as well as the commands used to create the image layers for that particular image.

You can browse a few other popular official repos to view the naming scheme that they use for their tags. For e.g.

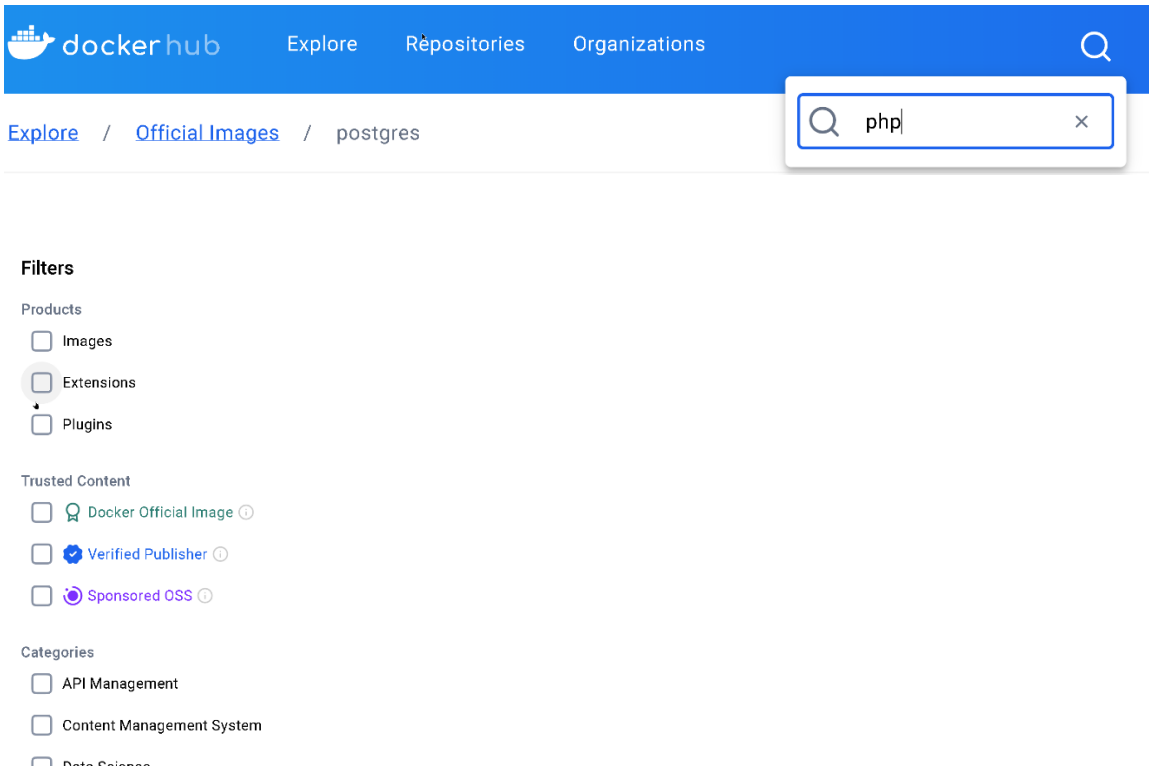
- Httpd (the Apache HTTP Server project)
- Nginx (an alternative popular web server to Httpd)
- Postgres (a popular relational DBMS)
- MySQL (another popular relational DBMS)
- Node (the runtime for executing Javascript programs server side)
- Python (the interpreter for Python programs)

You will notice that the images for applications (Httpd, Nginx, Postgres, etc) and programming language platforms (Node, Python, etc) have names like `bookworm`, `alpine`, `bullseye`, etc.

These are different Linux base images on which these applications and programming language platforms are installed on.

https://mohibulalam75.medium.com/exploring-lightweight-docker-base-images-alpine-slim-and-debian-releases-bookworm-bullseye-688f88067f4b

You can also do a general search for an image for a programming language or application that you are likely to use in your development work by typing a term in the search box at the top and then clicking on any one of the boxes on the left to further filter the search results returned.



5 Pulling images from Docker Hub

We pull images from the default registry (in this case Docker Hub) by specifying the image name (combination of `repo:tag`). Lets pull down a few light weight Linux distro images.

Search the official Alpine repository and identify a suitable image to download:
https://hub.docker.com/_/alpine/tags
For e.g. one with a suitable recent tag: 3.19

Type:

```
docker pull alpine:3.19
```

You should see an output similar to the following confirming a successful download of the image:

```
3.19: Pulling from library/alpine
46b060cc2620: Pull complete
Digest: sha256:95c16745f100f44cf9a0939fd3f357905f845f8b6fa7d0cde0e88c9764060185
```

```
Status: Downloaded newer image for alpine:3.19
docker.io/library/alpine:3.19
```

If you only specify the repo portion without a tag part for your image name, then Docker will automatically use the term `latest` for the tag. If no such tag exists on the repo, the pull attempt will fail.

Try typing just:

```
docker pull alpine
```

The output messages should confirm it is pulling down the image with the `latest` tag

```
Using default tag: latest
latest: Pulling from library/alpine
Digest: sha256:0a4eaa0eecf5f8c050e5bba433f58c052be7587ee8af3e8b3910ef9ab5fbe9f5
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

Repeat this for the following earlier images for the `alpine` and `busybox` repo:

```
docker image pull alpine:3.18.8
docker image pull alpine:3.18

docker image pull alpine:3.17.9
docker image pull alpine:3.17.8


docker pull busybox
docker pull busybox:glibc
docker pull busybox:1.36.1-glibc
docker pull busybox:1.35-musl
docker pull busybox:1.36-musl
```

Notice that in the process of downloading these image layers for these various images, Docker will be able to identify whether an image layer for an image to be downloaded already exists on the local registry. If so, it will not need to download that particular layer. This is because images can share identical layers in their construction.

You can also pull unofficial / user repos in the same way, for e.g:

```
docker pull radial/busyboxplus
```

Downloaded images from the DockerHub public registry are stored in a local registry at specific locations depending on the OS of the host machine (Windows / Linux / MacOS)
https://kodekloud.com/blog/where-docker-images-are-stored/

For Windows:
https://kodekloud.com/blog/where-docker-images-are-stored/#where-docker-images-are-stored-on-windows

6 Listing and filtering images

To get a list of all images available on the local registry, type:

```
docker images
```

The listing should appear similar to the following:

```
REPOSITORY                  TAG             IMAGE ID       CREATED         SIZE
alpine                      3.17.9          91dce2778e2d   10 days ago     7.08MB
alpine                      3.18            18f865063206   10 days ago     7.35MB
alpine                      3.18.8          18f865063206   10 days ago     7.35MB
alpine                      3.19            494edff73605   10 days ago     7.4MB
alpine                      latest          324bc02ae123   10 days ago     7.8MB……..
……..
```

To view only images from a specific repo, you could type:

```
docker images alpine
```

```
docker images busybox
```

The image ID is the actual unique identifier for a distinct image, and not the `repo:tag` form as a single image can have multiple different tags associated with it. For e.g. notice that several of the images from the `busybox` repo actually have identical image IDs.

```
REPOSITORY   TAG             IMAGE ID       CREATED         SIZE
busybox      1.36.1-glibc    65ad0d468eb1   14 months ago   4.26MB
busybox      glibc           65ad0d468eb1   14 months ago   4.26MB
busybox      latest          65ad0d468eb1   14 months ago   4.26MB
busybox      1.36-musl       615b080b9dbe   14 months ago   1.45MB
```

The different tags are basically different aliases for a single image stored in the local Docker image registry.
This is particularly common when we want to give the additional tag `latest` to an image that has been previously tagged with a version number.

You can add the `--filter` option to filter a long list of images, for e.g.

To view images with the `latest` tag:

```
docker images --filter=reference="*:latest"
```

To view all `busybox` images with the word `glibc` in their tag:

```
docker images --filter=reference="busybox:*glibc*"
```

To view all `alpine` images that are minor subversions within version 3.17:

```
docker images --filter=reference="alpine:3.17*"
```

7 Getting detailed information on images

We can get detailed information on an image with:

```
docker inspect alpine:3.17.9
```

You can alternatively also use the first 3 - 4 letters of the image ID to identify the image as well in this command (and any other commands that require you to identify a specific image), for e.g.

```
docker inspect imageid
```

Of particular interest is the `Cmd` section, which shows the commands that will be executed when a container is created and run from this particular image. We will be examining this in more detail in an upcoming lab

```
…
…
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"/bin/sh\"]"
            ],
…
…
```

You can also drill down into the output using a Go template expression that is passed to the `--format` option, as an example:

```
docker inspect --format '{{.ContainerConfig.Cmd}}' alpine:3.17.9
```

```
docker inspect --format '{{.Config.Env}}' imageid
```

8 Deleting images

Deleting an image will remove the image and all of its layers from the Docker host. However, if an image layer is shared by more than one image, that layer will not be deleted until all images that reference it have been deleted.

If the image you are trying to delete is in use by a running container you will not be able to delete it. You will have to stop and delete any containers before trying the delete operation again. We will study running containers in the next lab.

A basic image delete operation:

```
docker rmi alpine:3.17.9
```

The output messages will indicate that the delete operation starts first in an untagging of the existing tag for this image followed by the deletion of all corresponding image layers that comprise this image.

```
Untagged: alpine:3.17.9
Untagged: alpine@sha256:ef813b2faa3dd1a37f9ef6ca98347b72cd0f55e4ab29fb90946f1b853bf032d9
Deleted: sha256:91dce2778e2dd8d10b7f0788e874976006127847e156f7f40d38694225f43321
Deleted: sha256:76367d75676f0ab56722a770f40a80941396ff850244f7659bb2b2fe06b125aa
```

If you attempt to delete an image that is referenced to by multiple tags by one of its existing tags, that tag is simply untagged but the image still remains. For e.g. in this lab, the image associated with `busybox:glibc` tag is aliased with 2 other tags, so if we attempt to delete it with:

```
docker rmi busybox:glibc
```

The only message that comes out is:

```
Untagged: busybox:latest
```

If we check again with

```
docker images busybox
```

You will see the 2 other tags that are associated with that same image ID still remain:

If we now try and delete the 2 other tags:

```
docker rmi busybox:1.36.1-glibc
```

```
docker rmi busybox:latest
```

You will see that the only upon the deletion of the final tag is there an actual deletion of the corresponding image layers for that associated image ID (as shown in the output messages)

If you attempt to delete an image by its id, and that image has multiple tags associated with it, you will get an error. Let's examine the repo `alpine` again:

```
docker images alpine
```

Here we notice that there are 2 tags associated with the same image ID:

```
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
alpine       3.18      18f865063206   10 days ago   7.35MB
alpine       3.18.8    18f865063206   10 days ago   7.35MB
….
….
```

If we now attempt to delete this image using its ID with:

```
docker rmi imageid
```

We get an error response about the image being referenced in multiple repositories

We can however force delete it in this situation with:

```
docker rmi -f imageid
```

As you can see from the output, this operation will untag all the other associated tags first before deleting the actual image layers.

To remove all unused images (i.e. that are not being used by a running container) we can execute:

```
docker image prune -a
```

Check that there no more images remaining in the local repository after this with:

```
docker images
```

Lets pull a few more basic images from DockerHub to demonstrate another approach to mass deletion of images:

```
docker pull hello-world:nanoserver
docker pull hello-world:latest
docker pull hello-world:linux
```

The output from one Docker command can be based as arguments to another Docker command. Most Docker commands are capable of operating on multiple IDs (either IDs of images or containers). We can use the `$()` format to pass the evaluated results of a Docker command as arguments to another command.

For e.g.

```
docker images -q
```

gives us the listing of only the ids of all the images

Therefore, to delete all the images on the local registry we take this listing and pass it to the `docker image rm` command:

IMPORTANT NOTE: Don't run the command below if you are using Rancher Desktop, as you will end up deleting all the images that Rancher using to create the containers for its Kubernetes cluster infra.

```
docker rmi -f $(docker images -q)
```

Check that there no more images remaining in the local repository after this with:

```
docker images
```

9 Tagging images

Let's download an image again:

```
docker pull busybox:latest
```

We can use multiple `repo:tag` references to identify the same image, as we have seen earlier. The way to uniquely identify an image is through its image ID.

We can create as many alternative references for a given image as we want, for e.g.

```
docker tag busybox:latest myimage:v1

docker tag  myimage:v1 otherimage:v3
```

If you check now with:

```
docker images
```

you will see 3 different image references with the same image ID.

10 Pushing / pulling images to / from a Docker Hub user account

In order to upload (or push) an image from your local registry to Docker Hub (or some other configured remote registry), you will need to give a reference in the form of:

```
<user-name>/<repo-name>[:tag]
```

Lets create another reference to our previous image with:

```
docker tag busybox:latest dockerhub-username/coolimage:v1
```

Check that the new tag has been created correctly with:

```
docker images
```

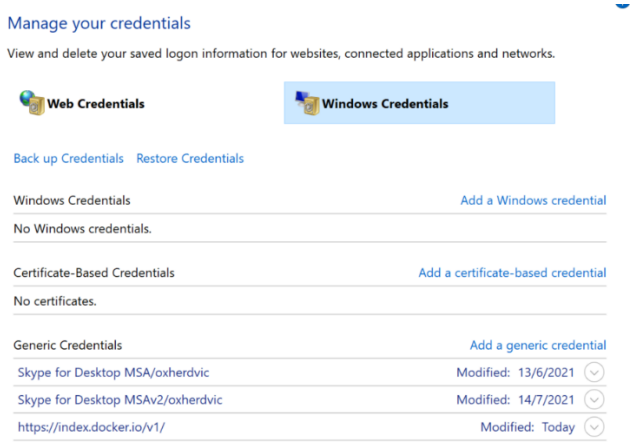
Next, login to your Docker Hub account via the CLI with:

```
docker login
```

and enter your username and password combination.

Once you have logged in successfully, Docker will automatically use the system credential manager to cache your username and password so that you do not need to repeat entering them for future push attempts.

On Windows, you can view the login credentials at: `Control Panel -> User Accounts -> Manage Windows Credentials`. In the area `Generic Credentials`, you should see some credentials related to Docker (https://index.docker.io/v1) . You will need to remove these credentials if you wish to push to a different Docker Hub account in the future from the same user account.

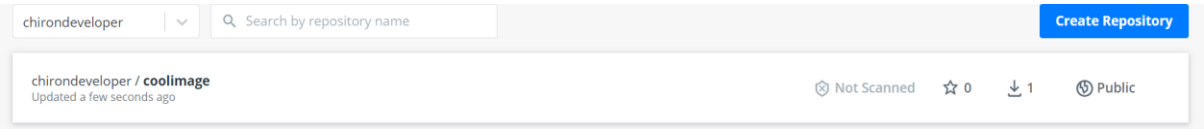


Next push the image by typing:

```
docker push dockerhub-username/coolimage:v1
```

Notice in the messages that appear that Docker is able to figure out that this image content is identical to an existing image in the DockerHub registry (`busybox:latest`) based on its SHA digest, and so it uses that image instead of actual uploading the current image on the local registry. This prevents unnecessary uploads and significantly saves bandwidth. Of course, if you were uploading a custom image that does not exist in the DockerHub registry (which we will be doing at a later point in this workshop), then the actual image will need to be uploaded.

You should now be able to see this pushed image in the `coolimage` repo on your Docker Hub account.



Lets go ahead and delete the single image on our local registry which is referenced by 4 different tags at the moment:

```
docker rmi -f imageid
```

Check that it is deleted

```
docker images
```

And then finally pull down the image that you just uploaded to your Docker Hub account:

```
docker pull dockerhub-username/coolimage:v1
```

11 Docker command categories

Type:

```
docker
```

You will see a long list which can be divided into 3 categories: common commands, management commands and commands

```
Common Commands:
  run         Create and run a new container from an image
  exec        Execute a command in a running container
  ps          List containers
…..

Management Commands:
  builder     Manage builds
  buildx*     Docker Buildx (Docker Inc., v0.14.1)
….
….

Commands:
  attach      Attach local standard input, output, and error streams
to a running container
  build       Build an image from a Dockerfile
….
…..
```

The common commands, as the name suggest, are commands that you will very commonly use in working with containers and were present since Docker was introduced. The Docker management commands represent a later logical reorganization of the original common commands from earlier versions of Docker, are used with an additional command action, for which you can get a list of by typing the first management command.

For e.g. type

```
docker container
```

You see a whole list of subcommands related to managing containers. Try getting help on one of these subcommands, for e.g:

```
docker container ls --help
```

The equivalent older common command for this is `docker ps` (you will see this listed as well as an alias for `docker container ls`). Try getting help on that by typing:

```
docker ps --help
```

You will also see `docker container ls` (and a variety of other commands) listed as aliases for this as well. Notice that the explanation and options for both these commands (`docker container ls` and `docker ps`) are identical, i.e. both of them are functionally equivalent.

In the labs for this workshop, we will be using commands from all these different categories. In particular, we will be using the common commands (`docker run`, `docker ps`, `docker images`, etc) as opposed to their newer versions (`docker container run`, `docker container ls`, `docker image ls`) since the more common commands are more popular and widely used in documentation and tutorials on the web and Youtube as opposed to their newer versions.

However, if you are in any doubt about the equivalency of commands, just run the commands with `--help` as previously demonstrated.

There is also a short list of equivalencies available here:
https://www.couchbase.com/blog/docker-1-13-management-commands/