UnixUtils
Published by: unixutils.com | Author: Vijayendar Gururaja | Email: admin@unixutils.com

Domain 2: Image Creation, Management, and Registry (20% of exam)

Describe the use of Dockerfile

Note: Dockerfile is a text file containing sequence of command instructions that is run in a sequence in order to build an image. Dockerfile can contain comments and instructions in the following pattern.

```
$ cat /path/to/Dockerfile
# Comment
INSTRUCTION arguments
```

`"docker build"` command is used for building an image out of the Docker file. "docker build" command requires a Dockefile and a Context (a set of files in a directory path or URL). The URL can contain a git source or plain text files or prepacked tarball. If a file named `.dockerignore` file is placed in this context directory containing files or directories (can contain patterns too), they are ignored from context.

```
$ cat /path/to/context-directory/.dockerignore
# comment
*/temp*
*/*/temp*
temp?
```

1. Build a docker image.
   ```
   # docker build .
   ```
   Note: This command expects the Dockerfile to be present in the current directory. The current directory is the context. '.' Represents the current working directory. One can also specify a different location that contains the files and directories needed for building the image.

   cli reference – docker build

2. Build docker image by passing the docker file.
   ```
   # docker build -f /path/to/Dockerfile /path/to/context/directory
   ```
   Note: This command uses the Dockerfile passed instead of searching it in current directory.

3. Build docker image and tag it to a couple of repositories.
   ```
   # docker build -t reponame/tagname:1.0.0 -t
      unixutils/myapp_dev_release:latest .
   ```
   Note: Tags are used to reveal some useful info about the image. An existing local image that is already built can also be tagged using "`docker tag image registryserver/image:tag`"

4. Build docker image using a git URL
   ```
   # docker build
      https://github.com/myaccount/dockerexample.git#BranchName:DirectoryName
   ```
   Note: The above example builds from context directory 'DirectoryName' from branch 'BranchName' out of the git URL specified.

```
# docker build http://server/context.tar.gz
```

Note: The tar is downloaded on the host the Docker daemon is running on, (not necessarily the same host from which the build command is being issued) and is used as context for build.

5. Build docker image using STDIN plain text Dockerfile from file path or URL to Dockerfile.

```
# docker build - < Dockerfile # example with STDIN Dockerfile
# docker build https://path/to/Dockerfile # example with URL to Dockerfile
```

Note: If you use STDIN or specify a URL pointing to a plain text file, there is no context. The Dockerfile cannot use a context since it would not work.

When you build an image with docker, let's say you have specified an invalid URL for a file download or may be an invalid command, this returns a non-zero exit value. The docker build fails when a non zero exit value is returned by one of its instructions.

Describe options, such as add, copy, volumes, expose, entry point

The following are different instructions that can be added to Dockerfile to build an image.

- *FROM* – Select the base image to build the new image on top of.

  Let's say you want your application to run with a centos base image, you begin the docker file with the below lines to choose a centos 7 base image. Example,

  ```
  #Download base image centos 7
  FROM --platform=LINUX centos:7
  ```

  RUN, CMD, ENTRYPOINT *are the three instructions that are used to execute a command*.

  There are two methods that can be used to execute a command in Dockerfile, which are the shell method and the exec method.

  shell method syntax: INSTRUCTION *some command*
  The above is run as: /bin/sh -c "some command"

  exec method syntax: INSTRUCTION ["executable", "param1", "param2", ...]
  The above is run as: executable param1 param2

- *RUN* - This is executed only during image build time and not during run time. (i.e not when the container is started). Example,

  ```
  #install RPM
  RUN yum install myapp.rpm
  ```

- *ENTRYPOINT*: can be used to specify command to run when container starts. Example, start a service. Example,

  ```
  #Call service script
  ENTRYPOINT ["/opt/myapp/bin/myapp_service", "--operation"]
  ```

- CMD: can be used to specify command to run when container starts OR it can also be used to specify an argument which will get passed to an ENTRYPOINT command. Example, Start a service or provide some arguments to ENTRYPOINT. Example,

  ```
  CMD start
  ```

  # "start" is an argument that gets passed as "/opt/myapp/bin/myapp_service --operation start" . This way of having arguments passed to ENTRYPOING is possible, only when ENTRYPOINT instruction uses *exec* form.

  "exec" method is recommended for ENTRYPOINT. If using "shell" method for ENTRYPOINT, it is recommended to specify keyword *exec* before the command. This will ensure that docker stop command successfully terminates the process. Example usage,

  ```
  ENTRYPOINT exec systemctl start myapp.service
  ```

  Recommended read: Understand how ENTRYPOINT and CMD interact

  For the above reasons, the RUN instruction keyword can appear multiple times in dockerfile whereas the CMD and ENTRYPOINT instruction keyword can appear only ones. Either CMD or ENTRYPOINT command should be present at least once.

- *SHELL*– Useful to run commands in shell form by overriding the default shell. Example,

  ```
  SHELL ["/bin/ksh", "-c"]
  # Instructions using shell for after above declaration will run as /bin/ksh -c "some command"
  ```

  - *USER* - Define the default User all commands will be run as within any Container created from your Image. Example,

    ```
    USER admin:somegroup
    # one can also use UID and GID instead.
    ```

- *VOLUME* - Creates a mount point within the Container and links it back to file systems accessible by the Docker Host. Example,

  VOLUME /var/www/html

  Volumes are better managed with docker-compose or "docker run" commands. Due to technical limitations path of volume in host cannot be mapped to a path in host through Dockerfile, however can be done with docker run command. Example, map a container volume to a path in host:

  ```
  # docker volume create MyVol
  Note: /var/lib/docker/volumes/TEST is created in docker host

  # docker run -it -v MyVol:/path/in/container <image-id> /bin/bash
  Note: MyVol is mounted in container, along with any contents from
  host.

  # docker run -it -v /data/MyVol:/path/in/container <image-id>
  /bin/bash
  Note: in this example we bindmount /data/MyVol in container instead
  of mounting a volume object created from "docker volume create"
  ```

- *ADD* – allows you to use a URL or a path to a file from Docker host as the source of a file to be copied to a path in container. When a URL is provided, a file is downloaded from the URL and copied to the destination specified in container. Example,

  ```
  ADD http://example.com/image.jpg /tmp/myappimage.jpg
  ADD /path/*.tar.gz /tmp/mypath/
  ADD --chown=user:group /data/loc2 /tmp/loc2
  ```

  When URL is passed, the contents are downloaded and placed in destination. When tar files are sourced, its contents are extracted to destination automatically. --chown is supported only in Linux containers.

- *COPY* – allows you copy local files only. URL is only supported in ADD. Also, tarfiles are not extracted like in ADD.

  COPY /path/*.txt /tmp/mypath/

- *WORKDIR* – Define the default working directory for the commands defined in the "ENTRYPOINT" or "CMD" instructions. Example,

  WORKDIR /home

- *EXPOSE* – Define which Container ports to expose

  EXPOSE 80/tcp

```
EXPOSE 80/udp
```

This only exposes the port in container, but does not publish the port on host. To publish port, it must be done with docker run command.

```
# docker run -p 80:80/tcp -p 80:80/udp -d nginx
```

- *ENV* – sets environmental variables in container

```
ENV key value
#allows single key value pair per instruction. Anything after key including whitespaces are considered as value. Quotes will be removed if not escaped.

ENV key1=value1 key2=value2 key3="some value"
#allows multiple key value pairs per instruction
```

Environmental variables can also be used during build time in an instruction. Variables can be referenced as $variable_name or ${variable_name}. Example,

```
RUN echo ${key}
#this translates to echo "value"
```

Recommended read: Environment replacement

- *ARG – define variables which can be used during build time. These variables cannot be referenced in the container (i.e during runtime)*

```
ARG appuser=nginxadmin
RUN useradd ${appuser}
```

# ARG is also the only instruction that can precede FROM instruction. Example,

```
ARG  image_version=latest
FROM alpine:${image_version}
```

Scope of ARGs:
ARG can have key value pairs or just keys with no value. Example,

```
ARG appuser
ARG appversion=1.0
```

- An instruction can use an ARG only after it has been defined in the docker file. Instructions that appear before ARG declaration will get an empty string.
- If an instruction is using an ARG that only has a key and no value, will get an empty string
- An ARG declared in Dockerfile can also be passed as an argument with a different value to “docker build” command. Example,

  ```
  # docker build --build-arg version=1.2 .
  ```

  In this case, the ARG value passed as argument using `--build-arg` takes priority. Thus, an ARG declared in Dockerfile acts as default that gets used only when no argument for that ARG is passed using `--build-arg`.

- If the same variable is defined in ARG and ENV, ENV takes priority. Example,

  ```
  ENV name=name-env
  ARG name=name-arg
  RUN echo ${name}
  #Will yield “echo name-env” during build time.
  ```

Predefined ARGS:

Predefined ARGS can be directly used without having to declare them. Recommended read: Predefined ARGS

- *ONBUILD* – Used to specify instructions that gets triggered on builds that uses this as baseimage and not during the build of the current Dockerfile itself.

  ```
  ONBUILD ARG appuser
  ONBUILD RUN useradd ${appuser}
  ```

- *STOPSIGNAL*– Used to specify signal call that will be sent to container to exit. Example,

  ```
  STOPSIGNAL SIGTERM
  #or
  STOPSIGNAL SIGKILL
  ```

- *HEALTHCHECK*– Used to specify a CMD instruction that will be triggered for health check. Example,

```
HEALTHCHECK --interval=5m --timeout=3s CMD systemctl status sshd.service || exit 1
```

The health check command should constructed with the following rules.
# 0: success - the container is healthy and ready for use
# 1: unhealthy - the container is not working correctly
# 2: reserved - do not use this exit code

- *LABEL*– Labels are Key Value pairs defined in DockerFile used to add metadata to an image. Example,

```
LABEL image_version="1.0.0"
LABEL image_provider="com.unixutils"
```

Identify and display the main parts of a Dockerfile

Following is an example of Docker file with the different instructions that was discussed in the previous section.

```
# cat /path/to/my/Dockerfile
#use centos base image
FROM --platform=LINUX centos:7

#metadata
LABEL image_provider="com.unixutils"
LABEL author="vijay"
ONBUILD RUN ["/bin/sh", "-c", "echo Built from baseimage by UnixUtils> /tmp/test"]

#default user to run commands
USER root

#environmental variables
ENV app_binary_path="/opt/webapp/bin"

#default working directory to run commands
ARGS app_home="/opt/webapp"
WORKDIR ${app_home}

#install httpd
RUN ["yum", "install", "httpd", "-y"]

#Create volume in container
VOLUME /var/www/html

#Place files into container.
ADD http://example.com/image.jpg /var/www/html/images/
ADD /path/*.tar.gz /tmp/mypath/files/
COPY /my/config/web.conf /etc/webconfig/web.conf

#open port 80 on container
EXPOSE 80

#start httpd when the container is started
CMD ["-D", "FOREGROUND"]
ENTRYPOINT ["/usr/sbin/httpd"]
```

Describe and demonstrate how to create an efficient image via a Dockerfile

Following are the recommendations & best practices for creating an efficient image via Dockerfile

- *Create ephemeral containers* – Containers that can be easily stopped, destroyed & recreated with minimal setup and configuration.
- *Build context* – Exclude unnecessary files from build context. Use a *.dockerignore* file to add a list of files and directories to exclude from build.
- *Use Multistage builds* – As we know, every line of instruction in a Dockerfile is built as an individual layer in the image. When rebuilding a Dockerfile, only the changes are built again, and the rest is pulled from cached layers. Multistage-build feature in Dockerfile allows usage of multiple FROM instructions to leverage using this image cache. Every time a FROM instruction is encountered in a Dockerfile, a new stage of build is started. This is beneficial since a new image can be built based on a previous one. Example,

```
cat /multistage-build-example/Dockerfile
# STAGE1 – Build your artifacts for production and run test cases

FROM centos:latest as DevImage
WORKDIR /app
COPY dependencies/* /app
RUN ./configure
RUN make
RUN && make install
RUN tests/run_post_build_tests.sh --binary_path=/app/build/MyNewBinaryFile

# STAGE2 – Prod image creation – (this is the final image that contains only the
artifact that was built in STAGE1 and not the dependencies used for build)

FROM centos:latest as ProdImage
COPY --from=DevImage /app/build/MyNewBinaryFile /app/bin/MyBinaryFile
```

- *Don’t install unnecessary packages*
- *Decouple applications* – Example, have separate containers for different modules such as front-end-web-container, DB-container etc. It is recommended to have one process

per container (not mandatory). One can also have a single process with child worker processes like in apache/HTTPD.

- *Minimize no.* of layers – less layers makes image build process performant.
  NOTE: RUN, COPY, ADD instructions create separate layers when encountered.
- *Sort multi-line arguments passed to a command alphanumerically, for easy maintenance and readability*
- *Leverage Build Cache –* Docker by default uses Build cache. In Dockerfile, if parent image already exists, for every instruction, docker checks if an image layer exists that matches the instruction then the layer is used. *--no-cache=true* disables this behavior. For ADD and COPY instructions since it involves files, the source file's checksum is compared to validate and reuse the later.
- *Use only official images as base images in FROM.* (docker recommends Alpine)
- *Use LABEL to organize images*
- *Do not major updates after starting container.* Example, yum update. Instead, use containers with updated packages. Example, "RUN yum update" will not run if an existing image layer exists for this instruction
- *Split long commands into multiline using backslashes (\)*
- *Use ENV to update $PATH*. Example, ENV PATH /myapp/bin:$PATH.
- *Group commands working towards same goal in a single instruction*. Example,
  RUN export version=1.0 && unset version instead of having the unset in a separate RUN instruction. (because the var is unset the next image layer and would still persist in previous layer)
- *Use separate COPY or ADD instructions for different files* - If copying files use separate COPY instructions so that when individual files are update, only the layer for that specific file would be rebuilt and rest will be picked up from cache.
- *Use VOLUME for any mutable data storage such as DB, config files etc –* non static data should VOLUME.
- *Use absolute path for WORKDIR*
- *Images built with ONBUILD instructions in it should get a separate tag.* Example: ruby:1.9-onbuild or ruby:2.0-onbuild

Recommended read: Best practices for building docker images using Dockerfile

Describe and demonstrate how to use CLI commands to manage images, such as list, delete, prune, rmi

Following are the CLI commands used to manage docker images

```
1. Build docker image
# docker image build -f /path/to/dockerfie /path/to/context
```

cli reference – docker image build

```
2. Show image history
# docker image history <image>
```

cli reference – docker image history

```
3. Import a docker image from a local tar file or URL to tar file.
# docker image import </local/file.tar.gz OR https://url/file.tar.gz>
```

cli reference – docker image import

```
4. Load an image from a tar archive or STDIN
# docker image load -i </local/file.tar.gz OR STDIN>
```

cli reference – docker image load

LOAD vs IMPORT:

`Load` is used to load multiple images from a tar file. It loads all the image layers including base image and versions. This type of a tar file can be created by using `docker image save` command
`IMPORT` is used to import a docker filesystem image from a tar file. This type of a tar file can be created by using `docker container export`

```
5. Get detailed information about an image
# docker image inspect <image>
```

cli reference – docker image impect

```
6. List all images
# docker image ls
```

cli reference – docker image ls

```
6. Remove unused docker images.
# docker image prune
```

cli reference – docker image prune

```
7. pull an image or repo from registry
# docker image pull <repo:tag>
```

cli reference – docker image pull

```
8. push an image or repo to registry
# docker image push <repo:tag>
```

cli reference – docker image push

```
9. Remove docker image
# docker image rm <image>
```

cli reference – docker image rm

```
10. Save an image
# docker image save <image> -o MyImage.tar.gz
NOTE: This creates a tar file with all layers including baseimage and
versions.
```

cli reference – docker image save

```
11. Tag an image
# docker image tag <image> <repo:tag>
OR
# docker image tag <repo1:tag1> <repo2:tag2>
NOTE: An existing image ID or <repo:tag> can be tagged into a new
<repo:tag>
```

cli reference – docker image tag

Describe and demonstrate how to inspect images and report specific attributes using filter and format

Using --filter option to filter view images with specific attributes

```
# docker images --filter "label=com.example.version"
```
NOTE: filter images containing specified labels

```
# docker images --filter "dangling=true"
```
NOTE: filter unused dangling images

```
# docker images --filter "label=com.example.version=1.0"
```
NOTE: filter by image labels using label's key/values

```
# docker images --filter "before=image1"
```
NOTE: filter all images created before the specified image

```
# docker images --filter "since=image3"
```
NOTE: filter all images since the specified image was created

```
# docker images --filter reference='centos*:*'
```
NOTE: filter by pattern for repo and tag

Using --format, show only specific columns while listing images. Example, show only column ID or Repository names etc.

```
# docker images --format "{{.ID}}"

# docker images --format "{{.Repository}}"
# docker images --format "{{.Tag}}"

# docker images --format "{{.Digest}}"

# docker images --format "{{.CreatedSince}}"

# docker images --format "{{.CreatedAt}}"

# docker images --format "{{.Size}}"

# docker images --format "{{.ID}} {{.Size}}"
```
NOTE: can also specify multiple columns like in this example

```
# docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"
```
NOTE: use table formatting in output.

Describe and demonstrate how to tag an image.

Example of image name with hostname of private registry:
Example: `hostname:5000/reponame:tag`

Example if image name without private registry hostname:port:
Example: `reponame:tag`
If hostname is not provided, it defaults to docker's public registry.

Every image also has an imageID. An image can be referenced with the imagename or image-ID. The following output shows image "`centos:latest`" with image ID `7e6257c9f8d8`. (`centos` being the *reponame* and `latest` being the *tag* separated by :)

```
# docker image ls centos:latest
REPOSITORY       TAG             IMAGE ID         CREATED          SIZE
centos           latest          7e6257c9f8d8     2 weeks ago      203MB
```

One can reference the image with "`centos:latest`" OR "`7e6257c9f8d8`". One can pass either of these to `docker image ls` and yield the same output.

With that, *tags* can be understood as aliases to the images. (An alias with naming convention to convey useful information about the image.)

An existing image can be tagged multiple times. Which means, one can create multiple tags (aliases) to same image and call them.

Examples:
```
# docker tag centos:7 centos_custom:dev
# docker tag 7e6257c9f8d8 centos_custom:test
```

Tagging can also be done to a private repository, by prefixing the private registry server's hostname and port. Example

```
# docker tag centos:7 myregistryhost:5000/centos_custom:prod
```

As seen in the following output, when docker images are listed, same original image centos:7 with image-ID 7e6257c9f8d8 is tagged with new name centos_custon:dev, centos_custom:test, myregistryhost:5000/centos_custom:prod

```
# docker image ls | grep centos
centos                     7              7e6257c9f8d8       2 weeks ago        203MB
centos_custom                    dev            7e6257c9f8d8       2 weeks ago        203MB
centos_custom                    test           7e6257c9f8d8       2 weeks ago        203MB
myregistryhost:5000/centos_custom   prod           7e6257c9f8d8       2 weeks ago
203MB
```

Describe and demonstrate how to apply a file to create a Docker image

`docker image load` and `docker image import` are the two commands that allows using a tar file to create a docker image.

See the following sections which has already been discussed:

docker image import
docker image load
load vs import

Describe and demonstrate how to display layers of a Docker image

docker image inspect command and docker history command can be used to list the layers of the command.

```
# docker image inspect <image>
# docker image inspect <image> --format "{{.RootFS.Layers}}"
```

NOTE: --format can be used to filter the json and display a specific section or subsection.

```
# docker history --no-trunc <image>
NOTE: this will display the layer, size and the command/instruction
that created the specific layer.
```

Describe and demonstrate how to modify an image to a single layer (multi-stage build, single layer)

An image can be made to contain a single layer by leveraging the multistage build option explained in the following section that has been discussed already.

Multistage Build

Describe and demonstrate registry functions

Docker registry scalable and stateless application used for storing and distributing docker images. This is useful when one does not want to store images in docker hub and to take control of the image distribution pipeline.

The docker registry application is available as a docker image which can be pulled and run as a service.

1. Run docker registry application.

```
# docker run -d -p 5000:5000 --name MyPrivateRegistry registry:latest
```

NOTE: Standalone docker container running registry

```
# docker service create --replicas 2 --name myprivateregistry --publish
    published=5000,target=5000 registry:latest
```

# docker registry application running as a service in a swarm

2. Tag, push and pull operations

```
# docker tag centos:7 myprivateregistry:5000/centos_custom:prod
```

NOTE: Tags local image 'centos:7' to private registry

```
# docker push myprivateregistry:5000/centos_custom:prod
```

NOTE: this pushes the tagged image to registry server that the image is tagged to.

```
# docker pull myprivateregistry:5000/centos_custom:prod
```

NOTE: Pulls the image from private registry

Deploy a registry

As we know now, Registry is nothing but an instance of the registry image available on docker hub, which can be run as a standalone container or in swarm mode.

Below are a few key considerations for deploying a registry in a standalone container.

- If you'd like to test *registry* instance: deploy a standalone container of registry accessible via localhost only. You may also have multiple local registry instances by publishing different port numbers for each instance.
- Start the registry automatically by setting the restart policy to "always"
- Customizing storage: By default registry contents are found under `/var/lib/registry` inside container. This is stored as a volume externally on docker host. The path where volumes are store on host is

`/var/lib/docker/volumes`, which cannot be changed. However if you'd like to use a file system with SAN or other storage options such as `/opt/my-san-storage/registry` etc, you can use bind mount, which allows mapping any filesystem path to container.

- Externally accessible registry (i.e outside localhost):
  1. register a domain for registry server
  2. whitelist port 443 for external access on docker host running registry server container instance. Configure DNS and routing as well for access.
  3. create a directory and place certificate and private key in docker host that you want to use in registry instance. Example, `/ssl/mydomain.crt`, `/ssl/mydomain.key`.
  4. Include intermediate-certificates if needed. Example,
     ```
     # cat /ssl/mydomain.crt /myintermediate-certs/intermediate-certificates.pem > /ssl/mydomain.crt
     ```
  5. Bind mount this directory to instance.

Running a registry container - Example,

```
# docker run -d \
--restart=always \
--name registry \
-v /opt/my-san-storage/registry:/var/lib/registry
-v /ssl/certs:/certs \
-e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
-p 443:443 \
registry:2
```

NOTE: In the above example, certificate and key is mounted as volume in container. However sensitive information can should be stored as *secrets* object. But this is available only when running a service in swarm mode.

Running *registry* as a service - Example,

Instead of bind mounting the directory containing cert and key, *secret* object can be created and attached to service.

```
# docker secret create mydomain.crt /ssl/mydomain.crt
# docker secret create mydomain.key /ssl/mydomain.key
```

Update swarm node labels (supported only in swarm mode), to be used as a constraint while creating the service.

```
# docker node update --label-add registry=true node1
```

```
$ docker service create \
  --name registry \
  --secret domain.crt \
  --secret domain.key \
  --constraint 'node.labels.registry==true' \
  --mount type=bind,src=/opt/my-san-
storage/registry,dst=/var/lib/registry \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
  -e
REGISTRY_HTTP_TLS_CERTIFICATE=/run/secrets/mydomain.crt \
  -e REGISTRY_HTTP_TLS_KEY=/run/secrets/mydomain.key \
  --publish published=443,target=443 \
  --replicas 1 \
  registry:2
```

Note: that secrets attached to service are mounted under /run/secrets/<secret-name> in the container.

Registry instance uses Docker registry HTTP API protocol to facilitate image distribution. Example,

```
# curl -I -IX GET http://myregistryhost:5000/v2/
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json; charset=utf-8
Docker-Distribution-Api-Version: registry/2.0
X-Content-Type-Options: nosniff
Date: Thu, 03 Sep 2020 15:47:11 GMT

# curl -X GET http://myregistryhost:5000/v2/_catalog
{"repositories":["centos_custom"]}
```

- HTTP header considerations:
    1. For responses to requests under the “/v2/” Docker-*Distribution-API-Version* header should be set to “registry/2.0”, even for a 4xx response, to ensure proper resolution authentication realms and fallback to v1 by docker engine.
    2. Ensure that registry returns HTTP response “401” if credentials are not used.
    3. Ensure Health check commands do see 401 response as unhealthy.
    4. Load balancers like amazon LB does not allow changing healthy response code, in which case request can be directed to “/” to ensure response “200 OK” is received.

- Load balancing Considerations: The following should be the same for all instances running registry, to ensure there is no issues in processing service requests.
    1. Storage Driver
    2. `HTTP Secret (to ensure uploads are coordinated)`
    3. `Redis Cache (if configured)`
    4. `All registry instances should Same share filesystem (on same machine)`

- Restrict access to registry using *Auth:*
  Authentication with clear text is not supported. TLS must be configured to make it work.

    1. Basic auth (using htpasswd) – This is one of the ways to restrict access. In linux htpasswd uses flatfiles to store and manage usernames and passwords in linux. A htpasswd file can be created and mounted as a volume (`example, /auth/htpasswd`) in container and used to implement authentication by passing the following arguments while running the container.

       ```
       -e REGISTRY_AUTH=htpasswd \
       -e REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm
       \
       -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
       ```

       Post this change, image push/pull operations would work only after login, example,

       ```
       # docker login myregistryhost:5000
       ```

    2. Use proxies in front of registry if needed.
    3. Registry also supports delegated authentication where in auth requests are redirected to a different trust token server. (useful if companies want to have full control over authentication)

- Considerations for *Air-gapped registries* (registries not connected physically in local network)

    1. Build your local registry's data volume on a host where docker pull of remote images would work, and then migrate volume to the air-gapped network.
    2. Non distributable layers of an image cannot be pushed to private registries unless it is configured in daemon.json. Example,

       ```
       {
       ```

```
  "allow-nondistributable-artifacts":
["myregistrydomain.com:5000"]
}
```

Recommended read: deploying a registry

Log into a registry

We have discussed restricting registry access in the previous section. The following provides insight on logging into docker registry.

Except for a remote docker daemon such as docker-machine, to login to registry, on must be root or use sudo to login.

```
1. Login to docker registry
# docker login myregistrydomain.com:5000
NOTE: This will prompt for username and password

2. Login to docker registry with credentials as arguments
# docker login myregistrydomain.com:5000 -u admin -p password

3. Login to docker registry with stdin password.
# cat ~/mypassword | docker login myregistrydomain.com:5000 -u admin
```

Docker stores the credentials (base64 encoded) in `$HOME/.docker/config.json`. This is not recommended and instead one of the following credential-helper programs that interacts with external credential store such as OS keychain, can be used.

1. D-Bus Secret Service
2. Apple macOS keychain
3. MS windows credential manager
4. Pass or secretservice (linux)

The config.json shoud be updated with helper program’s executable suffixed in the following format. Also the executable should be available/exported in $PATH on registry host.

```
"docker-credential-<credential-helper-program's executable>":
        {
            "credsStore": "<credential-helper-program's executable>"
        }
```

Once you run `# docker Logout` and log back in, plain text credentials should be removed and the credential helper program should take effect.

For linux, *pass* or *secretservice* are the credential helper programs. If the binaries are not available, it fallsback to using the base64 encoded credentials stored in config.json

Credential helpers: Credential helper program can be written in any programming language as long as it follows the credential helper protocol.

The credential helper protocol via a script or program accepts 3 arguments namely: store, get, erase all of which accepts a json payload.

1. *store* - Json payload for *store* requires server address for which credential is to be associated, username and password OR an identify token which is just the username.
2. *get* - Json payload for *get* accepts the server address only and responds with username and password
3. *erase* – Json payload for erase accepts the server address only and removes corresponding username and password

Recommended read: Credential helpers

Utilize search in a registry

It is possible to search for images from docker registry by passing the string to search with.

```
1. search for “centos” from docker registry
# docker search centos
```

cli reference: docker search

Push an image to a registry

An image can be pushed to docker registry for distribution. We have already seen examples of this previously.

Reference to previous section: docker image – tag/push operation
cli reference: docker push

docker commit command allows committing the state of the container to an image. Example,

```
# docker commit 7e6257c9f8d8 my-custom-centos-image

# docker tag customimage myregistryserver:5000/customimage:1.0

# docker push myregistryserver:5000/customimage:1.0
```

Sign an image in a registry

A tag for a specific image can be trust signed.

```
1. Trust sign an image
# docker trust sign myimage:1.0

2. Trust sign a local image
# docker trust sign myimage:1.0 –local

3. Inspect/list trust signatures for all tags of an image
# docker trust inspect myimage --pretty

4. Inspect/list trust signatures for a specific tag of an image
# docker trust inspect myimage:2.0 --pretty
```

Pull and delete images from a registry

images can be pulled from registry either as single image i.e a specific tag or all images i.e the entire repository.

```
1. pull centos latest image from registry
# docker pull centos
NOTE: when tag is not specified, the latest tag is pulled

2. pull a specific tag(version) of centos
# docker pull centos:7

3. pull all tags of centos
# docker pull centos -a

cli reference: docker pull
```

By default, Docker daemon pulls three layers of an image at a time. This number can be modified by changing `--max-concurrent-downloads` in /etc/docker/daemon.json. Similarly `--max-concurrent-uploads` controls the image push operation.

Images in registry can be deleted using by issuing http DELETE request.

`HTTP request format: DELETE /v2/<name>/manifests/<reference>, in which,`

- <*name*> refers to image name. Example "`myimage`"
- <reference> refers to image digest. Example, "`sha256:fe2347002c630d5d61bf2f28f21246ad1c21cc6fd343e70b4cf1e5102f8711a9`".

`Example HTTP request,`

```
# curl -X DELETE http://myregistryserver:5000/v2/centos_custom
/manifests/sha256:fe2347002c630d5d61bf2f28f21246ad1c21cc6fd343e70b4c
f1e5102f8711a9
```