Environment Variables
printenv: prints all env variables

PATH: list of directories separated by a colon, OS looks for these directories for the execution of a program or command

export ENV_NAME=env variable value
above command sets the value of the env variable for the current terminal session

.bashrc: write permanent env variables in this file.
$ source ~/.bashrc: to refresh the .bashrc file

echo $ENV_VARIABLE_NAME: prints the value of the env variable

Processes
1.Process: running instance of a program

2. ps: command to see the running processes

3. To run a process in the background, add an '&' sign after the process name
e.g. $ gedit &

4. kill pid: kills the process

Managing Users
1. useradd or adduser
2. usermod
3. userdel

For adding a user, first, add a group to which the user will be added later.
addgroup <group-name>

Then add the user with the same name as the primary group.
adduser -S -G <group-name/user-name> <group-name/user-name>

--system/S: system user

File Permissions
ls -l: to see the file permissions.

 - - - - - - - - - -:

total 10 characters
1. if the first letter is 'd' then it is a directory, if the first letter is '-' then it is a file
2. next 9 characters are divided into groups of three characters, 'r': read, 'w': write, and 'x': execute.
3. the first group represents the user that owns the file or directory

4. the second group represents the group that owns the file or directory
5. the third group represents permissions for everyone else

chmod: to change the permissions of a file or directory
$ chmod u+x file/directory name: adding execute permission for the user
$ chmod u-x file/directory: removing execute permission of the user
similarly, 'o': others and 'g':groups can be used instead/along with 'u'

Docker Intro

Docker helps to build, run, and ship applications consistently.

$docker-compose up
Installs all the dependencies required by an application

$docker-compose down --rmi all
Removes all the dependencies required by an application

Docker architecture

1. Containers are similar to processes running on the OS.
2. But containers do not have full-blown OS but share the kernel of the underlying OS which manages resources such as the CPU and the main memory.

Development workflow using docker

1. Develop an application
2. Write a dockerfile
    Dockerfile: a text file containing instructions that are required to build the docker image
    Docker image: Contains everything required to run an application such as an OS, a runtime environment(Node/python), application files, 3rd party libraries, environment variables, etc.
    Container: Process having its own file system provided by the docker image, applications run inside a docker container having an isolated environment.
    docker-hub: Repository of docker images available for use.
3. push the docker image on docker-hub from the development machine
4. pull the same docker image from docker-hub on the test/prod machine

Docker images & containers

Docker image contains all the files and configuration settings required to run an application.

An image includes:
A cut-down OS
3rd party libraries
application files
env variables

Containers :
provide an isolated environment
can be stopped and restarted

it is a process having its own file system provided by the docker image
each container is an isolated environment and it is invisible to other containers(although it is possible to share the data between the containers).

Dockerfile

1. FROM is a Dockerfile Instruction used to specify Docker Image Name and start the build process
2. CMD is Dockerfile Instruction used to execute a command in a Running container, There should be one CMD in a Dockerfile.
3. RUN is a Dockerfile Instruction used to execute any commands on top of the current Docker Image
   RUN executes the command when you are building an Image.
4. LABEL in Dockerfile Instruction is used to specify metadata information of Docker Image
5. EXPOSE in Dockerfile Instruction is used to specify the Network port for the Docker container
6. ENV in Dockerfile Instruction is used to set Environment Variables with key and value.
7. ADD: Copies a file and directory from your host to a Docker image, however, can also fetch remote URLs, extract TAR/ZIP files, etc. It is used to download remote resources and extract TAR/ZIP files.
8. COPY in Dockerfile Instruction used to Copy a file or directory from your host to the Docker image, It is used to simply copy files or directories into the build context.
9. WORKDIR: specifying the working directory for the project. All the commands after this command will be executed in the current working directory set-up using this command.
10. ENTRYPOINT in Dockerfile Instruction is used to configure a container that you can run as an executable.
ENTRYPOINT specifies a command that will execute when the Docker container starts.
11. VOLUME in Dockerfile Instruction is used to create or mount the volume to the docker container.
12. ARG in Dockerfile Instruction is used to set Environment variables with key and value during the image build.
13. .dockerignore in Dockerfile Instruction is used to prevent copying local modules and other unwanted files from being copied into Docker Image.

https://www.fosstechnix.com/dockerfile-instructions/

Docker command-line

1. build: to build an image from dockerfile
$docker build -t <tag-of-the-image> <build-context/root-directory-of-the-project>
build context:
*root directory of the project
*docker engine refers to this directory as the base directory while building a docker image; it is unaware of anything outside this directory
*Dockerfile must be present in this directory
*most probably, as we run the build command inside the root directory of our project, the build context is usually '.' (pwd)

2. Run: start a container
```
$docker run <name of image>
$docker run -it <name/tag of image>, -it: to run the container in the interactive mode.
$docker run -it <name/tag of image> <command/s>
```

3.ps: list containers
```
$docker ps
$docker ps -a
```

4. stop: stop a container
```
$docker stop <name/id-of-container>
```
   Start: start a container
```
$docker start <name/id-of-container>
```

5. rm: remove a container
```
$docker container prune(removes stopped containers)
$docker rm -f <name/id-of-container> (-f: forcefully stop a container)
```

6. images: list images
```
$docker images
```

7. rmi: remove an image
```
$docker image prune(removes dangling images)
$docker rmi <image-name>
```

8. pull: download an image
```
$docker pull <image-name>
```

9. Append a command
```
$docker run <container-name> sleep 5
```

10. exec: execute a command inside a running container
```
$docker exec <container-name> <command>
e.g. docker exec my-container cat /etc/hosts
```

11. Run: attach and detach
```
$docker run <container-name>,  container runs in foreground, press ctrl-c to exit
$docker run -d <container-name>,   container runs in a detach mode i.e. in background
$docker attach <container-name>,   bring the container to the foreground.
```

12. inspect: extra details
```
$docker inspect <container-name>
```

13. Container logs
```
$docker logs <container-name>
```

14. Tagging images
```
$docker build -t <image-name>:<tag-of-the-image> <build-context-directory>
```

$docker image tag <source-image-name:old-tag/source-image-id> <source-image-name:new-tag>

15. naming containers
$docker run -d --name <custom-container-name> <image-name:tag>

Optimization

1. Docker image is a collection of several underlying layers:
   base image(maybe made up of several layers), adding user&grp may modify some files which form another layer, installing dependencies forms another layer, etc
2. So without optimization, docker will again reconstruct all the layers irrespective of whether the layers are changed or not.
3. Optimization: docker will not rebuild the layer if the corresponding instruction is not changed, it will reuse it from the cache.
4. So instead of copying all the application files at once, first copy those files which are required for installing the dependencies, install the dependencies, and then copy the remaining application files.

5. So, in the dockerfile, the instructions that don't change frequently (stable instructions) should be at the top while the instructions that change frequently (changing instructions) should be at the bottom of the dockerfile.

Pushing docker images

1. Tag the latest(updated version, not the latest tag) image with the name:
   $docker image tag <dockerhub-username>/<image-name:tag>
2. $docker login
3. $docker push <dockerhub-username>/<image-name:tag>

Saving and loading docker images

$docker image save -o <filename-for-saving-image>.tar <image-name:tag>

$docker image load -i <filename>.tar

Mapping host ports to the container ports

$ docker run -d -p <host-port>:<container-port> <image-name:tag>

Executing commands in the running container

$ docker exec -it <container name/id> <command>
  -it: interactive shell
exit: to exit from the prompt, it does not stop the container.

Docker run vs docker exec

Docker run: start a new container and then run a command

Docker exec: execute a command in a running container

Docker Volume

Volume: storage outside the container(either on the host or in the cloud)
Create:
$docker volume create <volume-name>
$docker run -d -p <host-port:container-port> -v <volume-name>:<absolute-path>
<image-name:tag>

Copying files between the container and the host

$docker cp <src> <dest>
Both commands should be run outside the container

Container --> host
$docker cp <container-id/name>:<full-absolute-path-to-file-or-directory>
<host-directory-path>
Host -- > container
$docker cp <host-directory-path <container-id/name>:<full-absolute-path-to-file-or-directory>

Mapping the host directory to a container directory

This is to reflect the changes done inside the host directory into the container directory without having to build the image repetitively.

$docker run -d -p <host-port>:<container-port> -v $(pwd):<container-directory-path>
<image-name:tag>

Cleaning workspace

$ docker container rm -f $(docker container ls -aq)

$ docker image rm -f $(docker image ls -aq)