Docker is a tool that allows developers, sys-admins etc. to easily deploy their applications in a sandbox (called containers) to run on the host operating system i.e. Linux. The key benefit of Docker is that it allows users to package an application with all of its dependencies into a standardized unit for software development. Unlike virtual machines, containers do not have high overhead and hence enable more efficient usage of the underlying system and resources.

The Docker platform runs natively on Linux (on x86-64, ARM and many other CPU architectures) and on Windows (x86-64). Docker Inc. builds products that let you build and run containers on Linux, Windows and macOS.

One of the most important enhancements is that Docker can now run Linux containers on Windows (LCOW), using Hyper-V technology

Docker Desktop for Windows is Docker designed to run on Windows 10. Docker Desktop for Windows uses Windows-native Hyper-V virtualization and networking and is the fastest and most reliable way to develop Docker apps on Windows

Containers provide most of the isolation of virtual machines at a fraction of the computing power.

Containers offer a logical packaging mechanism in which applications can be **abstracted** from the environment in which they run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop. This gives developers the ability to create predictable environments that are isolated from the rest of the applications and can be run anywhere.

Containers also give more granular control over resources giving your infrastructure improved efficiency. Companies like Google, Facebook, Netflix and Salesforce leverage containers to make large engineering teams more productive and to improve utilization of computer resources. In fact, Google credited containers for eliminating the need for an entire data center.

Cross-checking docker:

```
docker run hello-world
```

We are interested in building a busybox container. BusyBox is a software suite that provides several Unix utilities in a single executable file (https://en.wikipedia.org/wiki/BusyBox)

```
docker pull busybox
```

docker images

docker run busybox

Apparently, nothing happened. Behind the scenes, a lot of stuff happened. When you call run, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run *docker run busybox*, we didn't provide a command, so the container booted up, ran an empty command and then exited.

docker run busybox echo "hello from busybox"

docker ps

docker ps –a

We want to run more than one command:

docker run –it busybox sh [In this case, you will bash into the linux container. Running the run command with the -it flags attaches an interactive tty in the container. Now we can run as many commands in the container as we want.]

Enter ls

Enter uptime

Enter rm –rf bin

Enter ls

Enter uptime

Enter exit

docker run –it busybox sh

docker run --help

docker rm <container id1> <container id2>

docker container prune [delete all stopped containers with a status of exited]

docker stop id/name

docker rm id/name

docker rmi

docker run -it docker/whalesay sh

Webapps with Docker

The first thing we're going to look at is how we can run a dead-simple static website. We're going to pull a Docker image from Docker Hub, run the container and see how easy it is to run a webserver.

docker run --rm prakhar1989/static-site [--rm automatically removes the container when it exits]

Nginx is running...

Okay now that the server is running, how to see the website? What port is it running on? And more importantly, how do we access the container directly from our host machine? Hit Ctrl+C to stop the container.

Well, in this case, the client is not exposing any ports so we need to re-run the docker run command to publish ports. While we're at it, we should also find a way so that our terminal is not attached to the running container. This way, you can happily close your terminal and keep the container running. This is called detached mode.

docker run -d -P --name static-site prakhar1989/static-site

In the above command, -d will detach our terminal, -P will publish all exposed ports to random ports and finally --name corresponds to a name we want to give. Now we can see the ports by running the docker port [CONTAINER] command

docker port static-site

```
C:\Users\tmahmood>docker run -d -P --name static-site prakhar1989/static-site
ae07af01da512915ef4a86e333de726d88b3123c0931bb5e12b449c5a7690c9e

C:\Users\tmahmood>docker ps
CONTAINER ID   IMAGE                  COMMAND          CREATED         STATUS         PORTS
                 NAMES
ae07af01da51   prakhar1989/static-site   "./wrapper.sh"   9 seconds ago   Up 5 seconds   0.0.0.0:49154->80/tcp, 0.0.0.0:
49153->443/tcp   static-site

C:\Users\tmahmood>docker port static-site
443/tcp -> 0.0.0.0:49153
80/tcp -> 0.0.0.0:49154
```

You can open http://localhost:49154 in your browser.

The type of network a container uses, whether it is a bridge, an overlay, a macvlan (container wants to be connected to the main network independently) network, or a custom network plugin, is transparent from within the container. From the container's

point of view, it has a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details (every container will be behaving like a separate PC).

By default, when you create or run a container using docker create or docker run, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to Docker containers which are not connected to the container's network, use the --publish or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host to the outside world.

Flag value	Description
`-p 8080:80`	Map TCP port 80 in the container to port 8080 on the Docker host.
`-p 192.168.1.100:8080:80`	Map TCP port 80 in the container to port 8080 on the Docker host for connections to host IP 192.168.1.100.
`-p 8080:80/udp`	Map UDP port 80 in the container to port 8080 on the Docker host.
`-p 8080:80/tcp -p 8080:80/udp`	Map TCP port 80 in the container to TCP port 8080 on the Docker host, and map UDP port 80 in the container to UDP port 8080 on the Docker host.

specify a custom port to which the client will forward connections to the container.

docker run –p 8887:80 prakhar1989/static-site [I could also map to 8888 but Jupyter is already running on this port]

To stop a detached container, run docker stop by giving the container ID. In this case, we can use the name static-site we used to start the container

docker stop static-site

docker images

The TAG refers to a particular snapshot of the image and the IMAGE ID is the corresponding unique identifier for that image.

An image is akin to a git repository - images can be committed with changes and have multiple versions. If you don't provide a specific version number, the client defaults to latest.

docker search mongodb

- Base images are images that have no parent image, usually images with an OS like ubuntu, busybox or debian.
- Child images are images that build on base images and add additional functionality
- Official images are images that are officially maintained and supported by the folks at Docker. These are typically one word long. In the list of images above, the python, ubuntu, busybox and hello-world images are official images.
- User images are images created and shared by users like you and me. They build on base images and add additional functionality. Typically, these are formatted as user/image-name.

Create an image that sandboxes a simple Flask application to display cats

git clone https://github.com/prakhar1989/docker-curriculum.git

cd docker-curriculum/flask-app

As mentioned above, all user images are based on a base image. Since our application is written in Python, the base image we're going to use will be Python 3

A Dockerfile is a simple text file that contains a list of commands that the Docker client calls while creating an image. It's a simple way to automate the image creation process. The best part is that the commands you write in a Dockerfile are almost identical to their equivalent Linux commands. This means you don't really have to learn new syntax to create your own dockerfiles.

The application directory does contain a Dockerfile but since we're doing this for the first time, we'll create one from scratch. To start, create a new blank file in our favorite text-editor and save it in the same folder as the flask app by the name of Dockerfile.

We start with specifying our base image. Use the FROM keyword to do that -

```
FROM python:3
```

The next step usually is to write the commands of copying the files and installing the dependencies. First, we set a working directory and then copy all the files for our app.

```
# set a directory for the app

WORKDIR /usr/src/app
```

```
# copy all the files to the container
COPY . .
```

Now, that we have the files, we can install the dependencies.

```
# install dependencies
RUN pip install --no-cache-dir -r requirements.txt
```

The next thing we need to specify is the port number that needs to be exposed. Since our flask app is running on port 5000, that's what we'll indicate.

```
EXPOSE 5000
```

The last step is to write the command for running the application, which is simply - python ./app.py. We use the CMD command to do that -

```
CMD ["python", "./app.py"]
```

The primary purpose of CMD is to tell the container which command it should run when it is started. With that, our Dockerfile is now ready.

The docker build command does the heavy-lifting of creating a Docker image from a Dockerfile.

```
docker build –t compadrejaysee/catimages .
```

```
docker run -p 8889:5000 compadrejaysee/catimages
```

You will have to cross-check that ports are not previously held by other applications.

```
docker login
```

```
docker push compadrejaysee/catimages
```

- The Dockerfile is a text file that (mostly) contains the instructions that you would execute on the command line to create an image.
- A Dockerfile is a step by step set of instructions.

- Docker provides a set of standard instructions to be used in the Dockerfile, like FROM, COPY, RUN, ENV, EXPOSE, CMD just to name a few basic ones.
- Docker will build a Docker image automatically by reading these instructions from the Dockerfile.

You'll be basically describing the build steps of your environment in the Dockerfile

This also implies that understanding Dockerfile instructions is not enough to create your Dockerfile, because you need to also understand the context of the technology you are building for. If, for example, you are building a Dockerfile to be used in a PHP project, you'll need to dive into PHP specific knowledge, like configuration methods, PHP extensions, environment settings and such.

The good news is that you can save a lot of time when starting out experimenting with a new technology, because you can use an image prepared by someone else, without understanding the details immediately. Once you are up for some more complex stuff you can start adding to the knowledge that you can extract and learn from other people's Dockerfiles.

Create a separate folder trial and access it

touch Dockerfile

Let's create a custom image from Alpine that has git, vim and curl included.

Every Dockerfile must start with the FROM instruction. The idea behind is that you need a starting point to build your image. You can start FROM scratch, scratch is an explicitly empty image on the Docker store that is used to build base images like Alpine, Debian and so on.

FROM alpine:3.4

Please add the lines to install vim and curl like this:

FROM alpine:3.4

RUN apk git

RUN apk add vim

RUN apk add curl

docker build -t compadrejaysee/alpinejc:1.0 .

note the . (dot) at the end of the line. You need to specify the directory where docker build should be looking for a Dockerfile. Therefore . tells docker build to look for the file in the current directory.

docker images

docker run --rm -ti compadrejaysee/alpinejc:1.0 /bin/sh

vim --v

curl –version

With every step in the build process Docker will create an intermediary image for the specific step. This means that Docker will take the base image (alpine:3.4), then execute RUN apk update and then Docker will add the resulting files from that step as another layer on top of the base image.

This means that the final Docker image consist of 4 layers and the intermediary layers are also available on your system as standalone images. This is useful because Docker will use the intermediary images as image cache, which means your future builds will be much faster for those Dockerfile steps that you do not modify.

docker images –a (list all intermediary images as well)

Only RUN, COPY and ADD instructions create layers to improve build performance.

The main advantage of image layering lies in image caching.

If you build your Dockerfile again now with the same command docker build -t compadrejaysee/alpinejc:1.0 ., you'll notice that the build was almost instantaneous and the output for every step says that the build was done from cache.

This behavior makes our lives a lot easier. Since image layers are built on top of each other Docker will use images cache during the build process up to the line where the first change occurs in your Dockerfile. Every later step will be re-built.

Change the Dockerfile as follows:

FROM alpine:3.4

RUN apk update

RUN apk add vim

RUN apk add git

The first three steps will be executed very quickly due to image caching

If you change an early step in the Dockerfile, for example you add one line after apk update like this:

FROM alpine:3.4RUN apk updateRUN apk add curlRUN apk add vimRUN apk add git

In this case every step after the change will be re-built. Which means that the steps to install curl, vim and git will be run from scratch, no caching will be available beyond the point where the change occurred.

docker images --filter "dangling=true"

image that we built with curl is still hanging around and it does not have a proper tag or name right now

docker system prune

```
FROM alpine:3.4
RUN apk update && \
    apk add curl && \
    apk add vim && \
    apk add git
```

```
FROM alpine:3.4
RUN apk update && \
    apk add curl && \
    apk add git && \
    apk add vim
```

```
FROM alpine:3.4
RUN apk update &&
    apk add \
    curl \
```

```
    git \
    vim
```

Start your Dockerfile with the steps that are least likely to change

- Install tools that are needed to build your application.
- Install dependencies, libraries and packages.
- Build your application.

The directory where you issue the docker build command is called the build context. Docker will send all of the files and directories in your build directory to the Docker daemon as part of the build context. If you have stuff in your directory that is not needed by your build, you'll have an unnecessarily larger build context that results in a larger image size.

You can remedy this situation by adding a .dockerignore file that works similarly to .gitignore. You can specify the list of folders and files that should be ignored in the build context.

If you want to have a look at the size of your build context, just check out the first line of your docker build output.

It is in your best interest to design and build Docker images that can be destroyed and recreated/replaced automatically or with minimal configuration.

Which means that you should create Dockerfiles that define stateless images.

One container should have one concern

Think of containers as entities that take responsibility for one aspect of your project. So design your application in a way that your web server, database, in-memory cache and other components have their own dedicated containers.

You'll see the benefits of such a design when scaling your app horizontally.

- FROM - every Dockerfile starts with FROM, with the introduction of multi-stage builds as of version 17.05, you can have more than one FROM instruction in one Dockerfile.
- COPY vs ADD - these two are often confused
- ENV - setting environment variables is important.
- RUN - run commands.

- VOLUME - another source of confusion, what's the difference between Dockerfile VOLUME and container volumes?
- USER - when root is too mainstream.
- WORKDIR - set the working directory.
- EXPOSE - get your ports right.
- ONBUILD - give more flexibility to your team and clients.

FROM

every Dockerfile must start with the FROM instruction in the form of FROM <image>[:tag]. This will set the base image for your Dockerfile, which means that subsequent instructions will be applied to this base image.

The tag value is optional, if you don't specify the tag Docker will use the tag latest and will try and use or pull the latest version of the base image during build.

ADD vs COPY

Both ADD and COPY are designed to add directories and files to your Docker image in the form of ADD <src>... <dest> or COPY <src>... <dest>. Most resources, including myself, suggest to use COPY.

The reason behind this is that ADD has extra features compared to COPY that make ADD more unpredictable and a bit over-designed. ADD can pull files from url sources, which COPY cannot. ADD can also extract compressed files assuming it can recognize and handle the format. You cannot extract archives with COPY.

The ADD instruction was added to Docker first, and COPY was added later to provide a straightforward, rock solid solution for copying files and directories into your container's file system.

If you want to pull files from the web into your image use RUN and curl and uncompress your files with RUN and commands you would use on the command line.

ENV

ENV is used to define environment variables. The interesting thing about ENV is that it does two things:

1. You can use it to define environment variables that will be available in your container. So when you build an image and start up a container with that image you'll find that the environment variable is available and is set to the value you specified in the Dockerfile.

2. You can use the variables that you specify by ENV in the Dockerfile itself. So in subsequent instructions the environment variable will be available.

RUN

RUN will execute commands, so it's one of the most-used instructions. I would like to highlight two points

You'll use a lot of apt-get type of commands to add new packages to your image. It's always advisable to put apt-get update and apt-get install commands on the same line. This is important because of layer caching. Having these on two separate lines would mean that if you add a new package to your install list, the layer with apt-get update will not be invalidated in the layer cache and you might end up in a mess

RUN has two forms; RUN <command> (called shell form) and RUN ["executable", "param1", "param2"] called exec form. Please note that RUN <command> will invoke a shell automatically (/bin/sh -c by default), while the exec form will not invoke a command shell.

VOLUME

You can use the VOLUME instruction in a Dockerfile to tell Docker that the stuff you store in that specific directory should be stored on the host file system not in the container file system. This implies that stuff stored in the volume will persist and be available also after you destroy the container.

In other words it is best practice to create a volume for your data files, database files, or any file or directory that your users will change when they use your application.

The data stored in the volume will remain on the host machine even if you stop the container and remove the container with docker rm. (The volume will be removed on exit if you start the container with docker run --rm, though.)

You can also share these volumes between containers with docker run --volumes-from.

You can inspect your volumes with the docker volume ls and docker volume inspect commands.

You can also have a look inside your volumes by navigating to Docker volumes in your file system.

The difference between the VOLUME instruction in Dockerfile and starting your container with docker run -v ... is this:

VOLUME in Dockerfile will create a new empty directory for your files under the standard Docker structure, i.e. /var/lib/docker/volumes. docker run -v ... can do more, you can mount existing directories from your host file system into your container and you can also specify the path of the directory on the host.

USER

Don't run your stuff as root, be humble, use the USER instruction to specify the user. This user will be used to run any subsequent RUN, CMD AND ENDPOINT instructions in your Dockerfile

WORKDIR

A very convenient way to define the working directory, it will be used with subsequent RUN, CMD, ENTRYPOINT, COPY and ADD instructions. You can specify WORKDIR multiple times in a Dockerfile.

If the directory does not exists, Docker will create it for you.

EXPOSE

An important instruction to inform your users about the ports your application is listening on. EXPOSE will not publish the port, you need to use docker run -p... to do that when you start the container.

CMD and ENTRYPOINT

CMD is the instruction to specify what component is to be run by your image with arguments in the following form: CMD ["executable", "param1", "param2"…].

You can override CMD when you're starting up your container by specifying your command after the image name like this: $ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...].

You can only specify one CMD in a Dockerfile (OK, physically you can specify more than one, but only the last one will be used).

It is good practice to specify a CMD even if you are developing a generic container, in this case an interactive shell is a good CMD entry. So you do CMD ["python"] or CMD ["php", "-a"] to give your users something to work with.

So what's the deal with ENTRYPOINT? When you specify an entry point, your image will work a bit differently. You use ENTRYPOINT as the main executable of your image. In this case whatever you specify in CMD will be added to ENTRYPOINT as parameters.

ENTRYPOINT ["git"]CMD ["--help"]

This way you can build Docker images that mimic the behavior of the main executable you specify in ENTRYPOINT.

ONBUILD

You can specify instructions with ONBUILD that will be executed when your image is used as the base image of another Dockerfile. :)

This is useful when you want to create a generic base image to be used in different variations by many Dockerfiles, or in many projects or by many parties.

So you do not need to add the specific stuff immediately, like you don't need to copy the source code or config files in the base image. How could you even do that, when these things will be available only later?

So what you do instead is to add ONBUILD instructions. So you can do something like this:

ONBUILD COPY . /usr/src/appONBUILD RUN /usr/src/app/mybuild.sh

ONBUILD instructions will be executed right after the FROM instruction in the downstram Dockerfile.

Pick the right base image

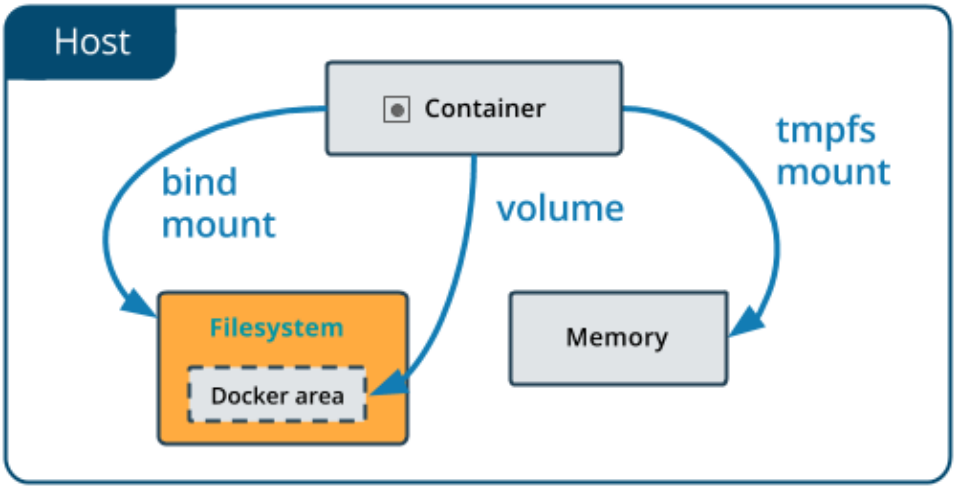
Go to shell and build your environment (start manually executing the steps in the container and see how things work out)

Add the steps to your Dockerfile and build your image

Repeat above two points

Find more here: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Find more here: https://docs.docker.com/engine/reference/builder/



Docker Exercises:

For more help, see the following URLs:

- http://tutorials.jenkov.com/docker/dockerfile.html
- https://intellipaat.com/blog/tutorial/devops-tutorial/docker-tutorial/
- https://stackify.com/docker-tutorial/

Exercise 1:

Start 3 containers from image that does not automatically exit, such as nginx, detached.

Stop 2 of the containers leaving 1 up.

Submitting the output for docker ps -a is enough to prove this exercise has been done.

Exercise 2:

Clean the docker daemon from all images and containers.

Submit the output for docker ps -a and docker images

Exercise 3:

Start image devopsdockeruh/pull_exercise with flags -it like so: docker run -it devopsdockeruh/pull_exercise. It will wait for your input. Navigate through docker hub to find the docs and Dockerfile that was used to create the image.

Read the Dockerfile and/or docs to learn what input will get the application to answer a “secret message”.

Submit the secret message and command(s) given to get it as your answer.

Exercise 4:

Start image devopsdockeruh/exec_bash_exercise, it will start a container with clock-like features and create a log. Go inside the container and use tail -f ./logs.txt to follow the logs. Every 15 seconds the clock will send you a “secret message”.

Submit the secret message and command(s) given as your answer.

Exercise 5:

Start a ubuntu image with the process sh -c 'echo "Input website:"; read website; echo "Searching.."; sleep 1; curl http://$website;'

You will notice that a few things required for proper execution are missing. Be sure to remind yourself which flags to use so that the read actually waits for input.

Note also that curl is NOT installed in the container yet. You will have to install it from inside of the container.

Test inputting helsinki.fi into the application. It should respond with something like

```
<html>
<head>
  <title>301 Moved Permanently</title>
</head>
<body>
  <h1>Moved Permanently</h1>
  <p>The document has moved <a href="http://www.helsinki.fi/">here</a>.</p>
</body>
</html>
```

This time return the command you used to start the process and the command(s) you used to fix the ensuing problems.

This exercise has multiple solutions, if the curl for helsinki.fi works then it's done. Can you figure out other (smart) solutions?

Exercise 6:

Create a Dockerfile that starts with FROM devopsdockeruh/overwrite_cmd_exercise and works only as a clock.

The developer has poorly documented how the application works. Passing flags will open different functionalities, but we'd like to create a simplified version of it.

Add a CMD line to the Dockerfile and tag it as "docker-clock" so that docker run docker-clock starts the application and the clock output.

Return both Dockerfile(s) and the command you used to run the container(s)

Exercise 7:

Make a script file for echo "Input website:"; read website; echo "Searching.."; sleep 1; curl http://$website; and run it inside the container using CMD. Build the image with tag "curler".

Run command docker run [options] curler (with correct flags again, as in 1.5) and input helsinki.fi into it. Output should match the 1.5 one.

Return both Dockerfile(s) and the command you used to run the container(s)

Exercise 8:

In this exercise we won't create a new Dockerfile. Image devopsdockeruh/ports_exercise will start a web service in port 80. Use -p flag to access the contents with your browser.

Submit your used commands for this exercise.

Exercise 9:

Create Dockerfile for an application in any of your own repositories and publish it to Docker Hub. This can be any project except clones / forks of backend-example or frontend-example.

For this exercise to be complete you have to provide the link to the project in docker hub, make sure you at least have a basic description and instructions for how to run the application in a README that's available through your submission.

Exercise 10:

Create an image that contains your favorite programming environment in it’s entirety.

This means that a computer that only has docker can use the image to start a container which contains all the tools and libraries. Excluding IDE / Editor. The environment can be partially used by running commands manually inside the container.

Explain what you created and publish it to Docker Hub.