Docker Basics

Images and containers	A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files. A **container** is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, `docker ps`, just as you would in Linux.
DockerFile	It is a text file that has all commands which need to be run for building a given image.
Hypervisor	A hypervisor is a software that makes virtualization happen because of which is sometimes referred to as the Virtual Machine Monitor.
Difference between virtualization and containerization	Containers provide an isolated environment for running the application. Whereas in Virtualization, hypervisors provide an entire virtual machine to the guest(including Kernel).
Docker Architecture	Docker Architecture has a Docker engine which is basically a client-server application. It has 3 major components which are: A server or a kind of long-running program, known as a **daemon process**. (The Docker Command) A **REST API** usually specifies the interfaces that can be used by the programs to talk with daemons and to instruct what to do. A **CLI (Command Line Interface) client**. It uses REST API to interact with or control the Daemon. It is done through CLI commands or scripting.
Docker namespace	A namespace is basically a Linux feature that ensures OS resources partition in a mutually exclusive manner. This forms the core concept behind containerization as namespaces introduce a layer of isolation amongst the containers. In docker, the namespaces ensure that the containers are portable and they don't affect the underlying host. Examples for namespace types that are currently being supported by Docker – **PID, Mount, User, Network, IPC**.
Docker Compose	
Docker components	Docker Client: This component performs “build” and “run” operations for the purpose of

	opening communication with the docker host. Docker Host: This component has the main docker daemon and hosts containers and their associated images. The daemon establishes a connection with the docker registry. Docker Registry: This component stores the docker images. There can be a public registry or a private one. The most famous public registries are Docker Hub and Docker Cloud.
	A container MUST be in the stopped state before we can remove it.
Differentiate between COPY and ADD commands that are used in a Dockerfile	
Can a container restart by itself	available policies: Off On-failure Unless-stopped Always
Diff between **docker Image and Layer**	
docker volumes stored in docker	`/var/lib/docker/volumes/`
most commonly used instructions in Dockerfile	FROM: LABEL: RUN: CMD:
Daemon Logging and Container Logging	• In docker, logging is supported at 2 levels and they are logging at the Daemon level or logging at the Container level. • Daemon Level: This kind of logging has four levels- Debug, Info, Error, and Fatal. - Debug has all the data that happened during the execution of the daemon

	process. - Info carries all the information along with the error information during the execution of the daemon process. - Errors have those errors that occurred during the execution of the daemon process. - Fatal has the fatal errors that occurred during the execution. • Container Level: - Container level logging can be done using the command: `sudo docker run -it <container_name> /bin/bash` - In order to check for the container level logs, we can run the command: `sudo docker logs <container_id>`
the best way of deleting a container	We need to follow the following two steps for deleting a container: - docker stop <container_id> - docker rm <container_id>
lifecycle of Docker Container	**Created**: This is the state where the container has just been created new but not started yet. **Running**: In this state, the container would be running with all its associated processes. **Paused**: This state happens when the running container has been paused. **Stopped**: This state happens when the running container has been stopped. **Deleted:** In this, the container is in a dead state.
How will you ensure that a container 1 runs before container 2 while using docker compose	`depends_on:` `  - db`

Docker concepts	Images and containers A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.

	A **container** is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, `docker ps`, just as you would in Linux.
Container	
Define a container with Dockerfile	`Dockerfile` `# syntax=docker/dockerfile:1` `FROM ubuntu:18.04` `COPY . /app` `RUN make /app` `CMD python /app/app.py` Each instruction creates one layer: • `FROM` creates a layer from the `ubuntu:18.04` Docker image. • `COPY` adds files from your Docker client's current directory. • `RUN` builds your application with `make`. • `CMD` specifies what command to run within the container.
Dockerfile instructions	FROM LABEL RUN The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the `Dockerfile`. `RUN apt-get update && apt-get install -y \` `package-bar \` `package-baz \` `package-foo \` `&& rm -rf /var/lib/apt/lists/*` CMD The `CMD` instruction should be used to run the software contained in your image, along with any arguments. `CMD` should almost always be used in the form of `CMD ["executable", "param1", "param2"…]`. Thus, if the image is for a service, such as Apache and Rails, you would run something like `CMD ["apache2","-DFOREGROUND"]`.

The main purpose of a `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified with the JSON array format.

EXPOSE

The `EXPOSE` instruction indicates the ports on which a container listens for connections

ENV

To make new software easier to run, you can use `ENV` to update the `PATH` environment variable for the software your container installs.

```
ENV PG_MAJOR=9.3
ENV PG_VERSION=9.3.4
```

ADD or COPY

Although `ADD` and `COPY` are functionally similar, generally speaking, `COPY` is preferred. That's because it's more transparent than `ADD`. `COPY` only supports the basic copying of local files into the container, while `ADD` has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for `ADD` is local tar file auto-extraction into the image, as in `ADD rootfs.tar.xz /`.

ENTRYPOINT

The best use for `ENTRYPOINT` is to set the image's main command, allowing that image to be run as though it was that command (and then use `CMD` as the default flags).

Let's start with an example of an image for the command line tool `s3cmd`:

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

Now the image can be run like this to show the command's help:

```
$ docker run s3cmd
```

	## VOLUME The `VOLUME` instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. The value can be a JSON array, `VOLUME ["/var/log/"]`, or a plain string with multiple arguments, such as `VOLUME /var/log` or `VOLUME /var/log /var/db` ## USER If a service can run without privileges, use `USER` to change to a non-root user. Start by creating the user and group in the `Dockerfile` with something like `RUN groupadd -r postgres && useradd --no-log-init -r -g postgres postgres`. ## WORKDIR
CMD vs Entrypoint	• CMD sets default command and/or parameters, which can be overwritten from command line when docker container runs. • Use CMD if you want the user to override the command. • ENTRYPOINT command and parameters will not be overwritten from command line. Instead, all command line arguments will be added after ENTRYPOINT parameters. Can be overridden by --entrypoint flag. • Use ENTRYPOINT if you want the container to be used as an executable and pass parameters via cmd line.
Services	In a distributed application, different pieces of the app are called “services”. For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on. `version: "3"` `services:` `  web:` `  # replace username/repo:tag with your name and image details` `  image: username/repo:tag` `  deploy:` `  replicas: 5` `  resources:` `  limits:` `  cpus: "0.1"`

	<pre> memory: 50M restart_policy: condition: on-failure ports: - "4000:80" networks: - webnet networks: webnet: docker service create --name="myservice" ubuntu</pre>
Docker compose	Using Compose is basically a three-step process: 1. Define your app's environment with a `Dockerfile` so it can be reproduced anywhere. 2. Define the services that make up your app in `docker-compose.yml` so they can be run together in an isolated environment. 3. Run `docker-compose up` and Compose starts and runs your entire app. A `docker-compose.yml` looks like this: <pre>version: '3' services: web: build: . ports: - "5000:5000" volumes: - .:/code - logvolume01:/var/log links: - redis redis: image: redis volumes: logvolume01: {}</pre> Compose has commands for managing the whole lifecycle of your application: • Start, stop, and rebuild services ……

	• View the status of running services • Stream the log output of running services • Run a one-off command on a service

Docker Volumes

	Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops: *volumes*, and *bind mounts*. If you're running Docker on Linux you can also use a *tmpfs mount*. Types: Volumes Bind Mounts tmfs mounts
Volumes	are stored in a part of the host filesystem which is *managed by Docker* (`/var/lib/docker/volumes/` on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker. `$ docker volume create my-vol` `$ docker volume ls` `$ docker volume rm my-vol` `$ docker run -d \` `  --name devtest \` `  --mount source=myvol2,target=/app \` `  nginx:latest` `or` `$ docker run -d \` `  --name devtest \` `  -v myvol2:/app \` `  nginx:latest`

Bind mounts	may be stored *anywhere* on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time. `$ docker run -d \` `  -it \` `  --name devtest \` `  --mount type=bind,source="$(pwd)"/target,target=/app \` `  nginx:latest` `$ docker run -d \` `  -it \` `  --name devtest \` `  -v "$(pwd)"/target:/app \` `  Nginx:latest`
`tmpfs` mounts	are stored in the host system's memory only, and are never written to the host system's filesystem. It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information. `$ docker run -d \` `  -it \` `  --name tmptest \` `  --mount type=tmpfs,destination=/app \` `  nginx:latest` `$ docker run -d \` `  -it \` `  --name tmptest \` `  --tmpfs /app \` `  nginx:latest`

Docker network

Network drivers	`bridge`: The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate. are best when you need multiple containers to communicate on the same Docker host.

	`host`: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated. `overlay`: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services. `macvlan`: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address. `none`: For this container, disable all networking. Usually used in conjunction with a custom network driver. `none` is not available for swarm services. Network plugins: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. ● **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host. ● **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated. ● **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services. ● **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address. ● **Third-party network plugins** allow you to integrate Docker with specialized network stacks.
Differences between user-defined bridges and the default bridge	**User-defined bridges provide better isolation and interoperability between containerized applications.** Containers connected to the same user-defined bridge network automatically expose all ports to each other, and no ports to the outside world. **User-defined bridges provide automatic DNS resolution between containers**. Containers on the default bridge network can only access each other by IP addresses

	Containers can be attached and detached from user-defined networks on the fly. **Each user-defined network creates a configurable bridge**. **Linked containers on the default bridge network share environment variables**. Originally, the only way to share environment variables between two containers was to link them using the `--link` flag. This type of variable sharing is not possible with user-defined networks. However, there are superior ways to share environment variables. A few ideas: • Multiple containers can mount a file or directory containing the shared information, using a Docker volume. • Multiple containers can be started together using `docker-compose` and the compose file can define the shared variables. • You can use swarm services instead of standalone containers, and take advantage of shared secrets and configs.
Manage a user-defined bridge	`$ docker network create my-net` `$ docker network rm my-net` `$ docker create --name my-nginx \` `  --network my-net \` `  --publish 8080:80 \` `  nginx:latest` `$ docker network connect my-net my-nginx` `$ docker network disconnect my-net my-nginx`
Configure the default bridge network	To configure the default `bridge` network, you specify options in `daemon.json`. Here is an example `daemon.json` with several options specified. Only specify the settings you need to customize. `{` `  "bip": "192.168.1.5/24",` `  "fixed-cidr": "192.168.1.5/25",` `  "fixed-cidr-v6": "2001:db8::/64",`

	<pre> "mtu": 1500, "default-gateway": "10.20.1.1", "default-gateway-v6": "2001:db8:abcd::89", "dns": ["10.20.1.2","10.20.1.3"] }</pre>
Use overlay networks	The `overlay` network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely. When you initialize a swarm or join a Docker host to an existing swarm, two new networks are created on that Docker host: • **an overlay network called `ingress`**, which handles control and data traffic related to swarm services. When you create a swarm service and do not connect it to a user-defined overlay network, it connects to the `ingress` network by default. • **a bridge network called `docker_gwbridge`**, which connects the individual Docker daemon to the other daemons participating in the swarm. . . . .
Use host networking	start a `nginx` container which binds directly to port 80 on the Docker host. <pre>docker run --rm -d --network host --name my_nginx nginx</pre>
Disable networking for a container	<pre>$ docker run --rm -dit \ --network none \ --name no-net-alpine \ alpine:latest \ ash</pre>

Configure containers

Start containers automatically	Use a restart policy: To configure the restart policy for a container, use the `--restart` flag when using the `docker run` command. The value of the `--restart` `No` `On-failure` `Always (`If it is manually stopped, it is restarted only when Docker daemon restarts or the container itself is manually restarted`)` `Unless-stopped (`Similar to `always`, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.`)`
Keep containers alive during daemon downtime	Use the following JSON to enable `live-restore`. `{` `  "live-restore": true` `}`
Runtime metrics	Docker stats `$ docker stats redis1 redis2`
Control groups	To figure out where your control groups are mounted, you can run: `$ grep cgroup /proc/mounts` `You can look into /proc/cgroups to see the different control group subsystems known to the system, the hierarchy they belong to, and how many groups they contain.` `You can also look at /proc/<pid>/cgroup to see which control groups a process belongs to. The control group is shown as a path relative to the root of the hierarchy mountpoint. / means the process has not been assigned to a group, while /lxc/pumpkin indicates that the`

	`process is a member of a container named pumpkin.`
Find the cgroup for a given container	Putting everything together to look at the memory metrics for a Docker container, take a look at `/sys/fs/cgroup/memory/docker/<longid>/`.

Specify a container's resources

`-m` or `--memory=`	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is `4m` (4 megabyte).
`--memory-swap`*	The amount of memory this container is allowed to swap to disk. See `--memory-swap` details.
`--memory-reservation`	Allows you to specify a soft limit smaller than `--memory` which is activated when Docker detects contention or low memory on the host machine. If you use `--memory-reservation`, it must be set lower than `--memory` for it to take precedence. Because it is a soft limit, it does not guarantee that the container doesn't exceed the limit.
`--cpus=<value>`	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set `--cpus="1.5"`, the container is guaranteed at most one and a half of the CPUs. This is the equivalent of setting `--cpu-period="100000"` and `--cpu-quota="150000"`. Available in Docker 1.13 and higher.

HEALTHCHECK	The `HEALTHCHECK` instruction has two forms: • `HEALTHCHECK [OPTIONS] CMD command` (check container health by running a command inside the container) • `HEALTHCHECK NONE` (disable any healthcheck inherited from the base image) healthcheck `starting healthy unhealthy` The options that can appear before `CMD` are: • **`--interval`**`=DURATION` (default: `30s`) • **`--timeout`**`=DURATION` (default: `30s`) • **`--start-period`**`=DURATION` (default: `0s`) • **`--retries`**`=N` (default: `3`) `HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost/ || exit 1` Health check in docker file ``` FROM nginx:1.13 HEALTHCHECK --interval=30s --timeout=3s \ CMD curl -f http://localhost/ || exit 1 EXPOSE 80 ``` Specify health check with docker run ``` docker run --name=nginx-proxy -d \ --health-cmd='stat /etc/nginx/nginx.conf || exit 1' \ nginx:1.13 ```
Prune images	`$ docker image prune` `WARNING! This will remove all dangling images.` `Are you sure you want to continue? [y/N] y` To remove all images which are not used by existing containers, use the `-a` flag: `$ docker image prune -a`

	`WARNING! This will remove all images without at least one container associated to them.` `Are you sure you want to continue? [y/N] y`
Prune containers	`$ docker container prune` `WARNING! This will remove all stopped containers.` `Are you sure you want to continue? [y/N] y` `$ docker container prune --filter "until=24h"`
Prune volumes	`$ docker volume prune` `WARNING! This will remove all volumes not used by at least one container.` `Are you sure you want to continue? [y/N] y` By default, all unused volumes are removed. You can limit the scope using the `--filter` flag. For instance, the following command only removes volumes which are not labelled with the `keep` label: `$ docker volume prune --filter "label!=keep"`
Prune networks	`$ docker network prune` `WARNING! This will remove all networks not used by at least one container.` `Are you sure you want to continue? [y/N] y`
Prune everything	`$ docker system prune` `WARNING! This will remove:` `- all stopped containers` `- all networks not used by at least one container` `- all dangling images` `- all build cache`

	Are you sure you want to continue? [y/N] y If you are on Docker 17.06.1 or higher and want to also prune volumes, add the `--volumes` flag: $ docker system prune --volumes WARNING! This will remove: - all stopped containers - all networks not used by at least one container - all volumes not used by at least one container - all dangling images - all build cache Are you sure you want to continue? [y/N] y

Docker main points to cover:
healthchecks
docker services
common commands

Run your app in production directory in notes