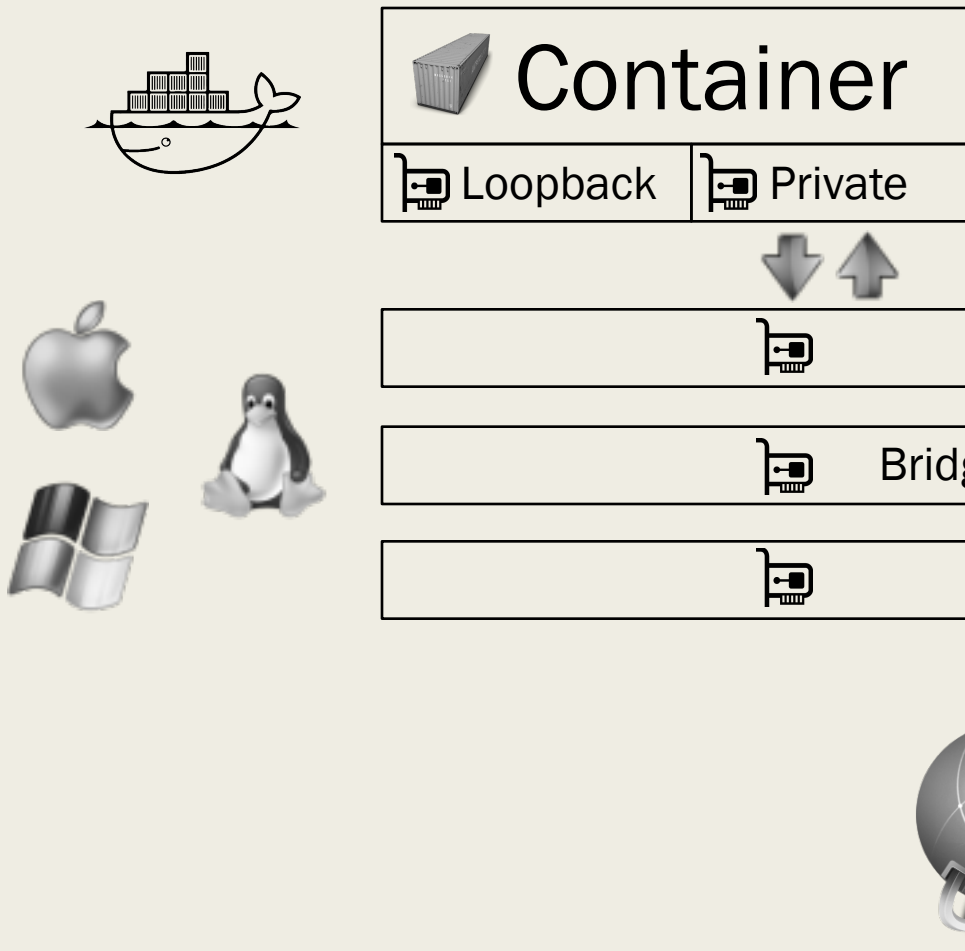
NETWORKING IN DOCKER

A networking quick refresher

- The following is just a quick refresher about important networking concepts that will be used in this section:

- A network protocol: it is an agreed-upon method of communication between two or more nodes on a network. For example, Hyper Text Transfer Protocol (HTTP), which is used for browsing web pages as well as other related tasks. Also File Transfer Protocol (FTP), which is used for uploading and downloading files.
- A network interface: the device that a given host uses to communicate with other hosts on the network. To be able to do that, an interface must have an IP address, which is a set of numbers separated by dots used to uniquely identify a device on a network. IP addresses use the Internet Protocol (IP).
- The loopback interface: when working with interfaces, you'll often be using the Ethernet interface. This is the one that will be having your IP address and you will use it to communicate over the network. But the computer has another special type of interface called the loopback. It is not connected to any any network and it has a special type of IP address (127.0.0.1). It is used by internal applications on the OS to communicate with each other using networking protocols.
- A network port: this is a unique number (per host) that identifies which application will be receiving the network traffic arriving at the network interface. For example, if you're running a web server, it – usually – operates on port 80. HTTP traffic received by the interface need to be redirected (by the OS) to the appropriate application (the web server), using it's designated port (80).
- Using the protocol, IP address (or hostname) and the network port, you can specify a meaningful *network address*. For example, to connect to a web server running on 192.168.1.100, running on port 8080 and using the HTTP protocol, you give your client application (like an Internet browser) the following address: `http://192.168.1.100:8080`

Docker network model – Illustration



How does a container access the public network?

- As you've seen in the previous illustration, Docker uses the OS networking capabilities to provide networking support for the containers.
- Each container has two interfaces:
  - *A loopback interface: this is completely isolated from any traffic outside the container and is used internally.*
  - *A public interface: this is used to communicate with the host*
- Once Docker daemon is installed on the host OS, a virtual interface is created to communicate with any containers built.
- A bridge interface is also created on the host (called docker0) that is used to bridge traffic coming from the outside network to and from the host to the container using the virtual interface.

Routing and NATing

- Because networks can get extremely large, they are divided into logical segments called “subnets”.
- In a single network subnet, all hosts can connect and communicate with each other without a problem. But when one host in a subnet want to connect to another host in another subnet, an intermediary device (called router) must be used.
- The router contains “routing tables”. Those are information about how to reach different hosts on the networks to which the router is connected. The router is responsible for connecting hosts on different subnets.
- Within Docker’s network topology, docker0 interface is like the router of an outside network; it is used to connect containers with each others in an internal network, it is also used to connect those containers to the outside world.
- Docker provides four different ways to build and connect containers. In the following we introduce each of them so that you can make an educated decision when choosing which model to use while creating your containers.

Closed container

- This container does not have any access to the outside world.
- It uses the loopback interface of the host to allow programs inside the container to communicate internally.
- This provides the maximum level of network protection; as the container will not be reachable from outside the host.
- It suffers the inability of the containers to connect to the outside network. This may not be the best option if you need network (or Internet) connection. For example, if the container needs to download extra packages before it can function correctly.
- In order to create a closed container, you use the --net none option in the creation command. For example, `docker run --rm --net none busybox ifconfig -a`
- The possible use cases of this model is where applications require the maximum security level, or where network access is not a requirement. Lets say you want to run the Python script used in one of the previous labs to convert CSV files to SQL statements, this program does not require outside network access so it should be created as a closed container.

Bridged containers

- This is the default type of networks in Docker containers.
- It reduces the level of network isolation in favor or providing outside connectivity.
- In this mode, the container has two interfaces: the loopback interface, and another private interface that is connected to the host's virtual interface and the bridge (docker0).
- All bridged containers can communicate with each other as they are connected to the same virtual network.
- All bridged containers can communicate with the outside network (and the Internet depending on the network configuration) through the host's bridge interface.
- This is the most common type of network-models in Docker. It will receive the most emphasis in this section.

Running a bridged container

- The bridged network model is the default in Docker. It is the model that is automatically chosen for you if you did not specify --net flag.
- In a bridged container, applications have access to two network interfaces: the loopback device, which is used for internal communication, and a private interface connected to the virtual interface of the host. This is the one used to connect to the outside network. It uses the host's bridge device to connect to the outside network.
- You can run a container in the bridged-networking mode by either running the container without any flags, or specifying --net bridge. For example:
  ```
  docker run --rm --net bridge busybox
  ```
- A container in the bridged mode can connect to other containers on the same host using the internal network. It can also connect to the outside network (and the Internet, depending on the configuration) using the host's bridge device.
- But all communication so far is done through the IP addresses. If you want the container to be identified on the network by the hostname, you will have to modify the creation command as we'll see next.

Containers' name resolution

- In any given network, if you want to be able to communicate with other nodes using their hostnames other than the IP addresses, you have to use a service/server called the DNS (Domain Name Service). It's role is to translate hostnames to IP addresses and vice versa.
- To assign a hostname to the container while creating it, you have a couple of options:
  - *Specify the hostname to the container explicitly using the –hostname flag. For example:*
    `docker run --rm --hostname helloDocker busybox nslookup helloDocker`
    Notice here that Docker overrides the DNS by manually assigning a hostname to the container. Also note that this hostname is used by the container internally; no other containers can connect to this container by it's hostname as it is not known to them.
  - *Specify one or more DNS servers that will be used in name resolution. For example:*
    `docker run --rm --dns=8.8.4.4 --dns=8.8.8.8 busybox nslookup google.com`
- Using the DNS option has a number of considerations to take care of:
  - *It must be specified in the form of an IP address. Which is natural.*
  - *The --dns can be set while the daemon is starting up. This way you needn't specify a DNS server for new containers; as they will automatically be assigned one. However, if you changed that DNS server and restarted the daemon to use the new DNS server, only neew containers will be affected by that change. Containers that were already running before the DNS was changed will continue to use the old DNS. You will have to manually assign them the new DNS address.*
- You can also instruct the containers to use a domain name by default if they want to resolve a hostname that does not contain the domain part. For example, myserver.example.com is said to be a *fully qualified domain name*. If it was written like this: myserver only, it may not be resolvable unless the client is instructed to append example.com by default to containers that do not have FQDNs. This can be done in Docker using the --dns-search. For example:
  `docker run --rm –dns-search google.com busybox nslookup mail`
  This container will correctly resolve mail.google.com to its IP address using the nslookup command, although the hostname used was only mail. You can ad more than one search domain during container creation. This option can also be configured on daemon startup.
- Another useful feature in name resolution provided by Docker is the ability to manually assign hostnames to IP addresses. This can e done using the –add-host flag. For example:
  docker run --rm –add-host www.google.com:127.0.0.1 busybox ping www.google.com
  now whenever any application tries to access www.google.com, the answer will come from the localhost loopback address 127.0.0.1. This technique is used often when you need to block some outside connections or when you want to re-route unsecure traffic through a secure channel like SSH. This flag cannot be set at daemon startup.

Controlling connections to the container

- So far we have seen how we can configure and control traffic leaving our container. Let's have a look at controlling network traffic arriving to the container. This is done through port-forwarding.
- By default, bridged containers cannot be reached from outside the host. This is a security measure for protecting the container. However, you can allow specific ports to be accessible from the outside network using port forwarding, which is done through the –p flag.
- The –p flag has four possible use-cases:
  - *-p N where N is the port container's port number. It is bound to a random port on all the host's interfaces.*
  - *-p N:n where n (small n) is the host's port where the container's N port is bound to. For for example, -p 8080:80 means that any traffic arriving at the host's port 80 will be automatically routed through Docker to the container listening at port 8080.*
  - *-p ip_address::N this will bind the container's port N to a random port on the host but only on the interface specified by the IP address.*
  - *-p ipaddress:n:N this will bind the container's port N to port n on the host but only on the interface indicated by the IP address.*
- You can opt to assign random host ports to all the ports exposed by a given container by using the –P flag. For example, `docker run –P httpd` will bind port 80 exposed by the webserver to a random host port.
- To determine which ports on the container are mapped to which ports on the host you can use `docker ps`, `docker inspect`, or `docker port` commands.

Joined containers

- It simply refers to more than one container sharing the same network interface.
- It is done in a similar way when we shared volumes. For example:
  ```
  docker run –d –name server1 –net none busybox nc –l
  127.0.0.1:8888
  docker run –it --rm  --net container:server1  busybox
  netstat –al
  ```
  notice that `netstat –al` when run on the second container will show you port 8888 listening on the local interface 127.0.0.1.
- Both containers still maintain network isolation concepts previously discussed.
- This model can be used if you have containers that need to communicate with each other but you don’t want any external access (even from the host). It can also be used in the absence of network discovery tools like DNS.
- Caution must be taken when running programs that have the same port numbers on two joined containers; as conflicts may arise.

Open containers

- This is the most unprotected sort of network models in Docker.
- Open containers have full access to the host's interface. They can use ports that are numbered lower than 1024.
- You should use open containers only when it is the only valid option. To create one you issue a command like this:
  docker run –it --net host busybox ifconfig –a
  the output of this command will show you all the network interfaces on the host, including the bridge interface.
- There is no isolation performed on this type of containers.

How do containers "know" about each other?

- So far we have seen how containers use different network models to communicate with each other and to communicate with the outside network. But given a container running a webserver, how can another container (like in a bridged network model) connect to that other container?
- A number of possible options exist: it can make a simple network scan to see which nodes on the network are listening on port 80. Alternatively you can use a local DNS server, and make the necessary configuration so that containers register themselves the moment they start. But Docker provides a much easier option: linking.
- We've seen linking before since the start of this course. When you link two or more containers, an entry will be added to the DNS override list (/etc/hosts) specifying the hostname and the IP address of the target machine to be easily discovered. For example:
  `docker run -d --name webserver1 httpd`
  `docker run -it --rm --link webserver1:web busybox wget web`
  what the second command did is that it created a second docker container that will jave a link to the container named webserver1. It will internally refer to it as web. Once the link is established, a command like wget (used to download files from webservers) can be used to download the index.html page from the webserver container.
- You should be aware – though – that there is nothing in Docker that will automatically link a container to other containers serving its required services. For example, you may be running an application that is expecting to find a webserver named http101. You will have to consult the documentation to do the necessary linking with the correct alias.

Environment variables creation

- Consider the following example:
  ```
  docker run –d --name baseserver --expose 8080 –expose 9090 busybox nc
  –l 0.0.0.0:8080
  docker run --rm –it --link baseserver:server busybox env
  ```
  Running the above commands will produce a list of environment variables that were automatically created when the container was linked.
- There will be one environment variable starting with the alias followed by 'NAME' (SERVER_NAME), then the container's name and the link alias separated by a slash.
- The rest of the environment variables can be classified as follows:

– *ALIAS_**PORT**_PORT_NUMBER_PROTOCOL_PORT*

– *ALIAS_**PORT**_PORT_NUMBER_PROTOCOL_ADDR*: this refers to the interface IP address on which the port is bound.

– *ALIAS_**PORT**_PORT_NUMBER_PROTOCOL_PROTO:* the name of the protocol (TCP or UDP).

– *ALIAS_**PORT**_PORT_NUMBER_PROTOCOL: this will have all the previous information encoded in one URL.*