By now you already know the terminologies related to Docker Network and the various Network types. Let's learn more about Networking by performing some hands-on !

Finding IP Address of the Container Using *inspect*

- `docker inspect` command

```
root@docker:~# docker run -d --name nginx nginx
63dfc086a93a9170fba5fb377c469011e0a5bcdec9e5ba10d3dd038240ecce6d
root@docker:~# docker inspect --format '{{ .NetworkSettings.IPAddress }}' nginx
172.17.0.4
root@docker:~#
```

Finding IP Address of the Container Using *exec*

- `docker exec` command

```
root@docker:~# docker exec -it nginx ip add | grep global
    inet 172.17.0.4/16 scope global eth0
root@docker:~# _
```

Exposing a Container Port on the Host

Ports are exposed in the container so that the container can be using the container IP. We can connect to the http address of that port.

How can we expose a Container Port?

- We are going to direct the port that is listening to http on a container to an underlying port on our host
- We can redirect the exposed ports to the host ports

There are two ways of exposing the ports – by using commands: P or p

- P: Any ports that are exposed by the container, any random port between 32768 and 65000 will be available on the host machine. These are the ports available for Docker to pick randomly.
- To expose the Port, use the below command:
    - `$ docker run - d --name = edurekanginx - P nginx:latest`

Container Port – Exposed

```
root@docker:~# docker run -d --name=edureka-nginx -P nginx
4b2804f50c49dc4bfd0dd0f92b27d961997ec0753d67931ff30c68204b3be710
root@docker:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS                                              NAMES
4b2804f50c49        nginx               "nginx -g 'daemon ..."   23 seconds ago      Up 22 seconds
     0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp      edureka-nginx

root@docker:~# elinks http://localhost:32769
```

-P : exposes to available ports

Exposed Ports

Welcome to nginx!

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

This page shows we are successfully connected to the ports and container ports are available through local host

Getting and Binding Ports

```
root@docker:~# docker port edureka-nginx $CONTAINERPORT
443/tcp -> 0.0.0.0:32768
80/tcp -> 0.0.0.0:32769
root@docker:~# _
```

Variable to be passed

Getting the container port, where port 443 goes to all local address through port 32468

- What if you want to bind it to a specific port on your server rather than any value, which is picked in a range, such as 8080

```
root@docker:~# docker run -d -p 8080:80 --name=nginx-edureka nginx
e2ae33bafb0131195a9feaa19a0a59307491a1a340c32448b82c487602472773
root@docker:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS                                   NAMES
e2ae33bafb01        nginx               "nginx -g 'daemon ..."   14 seconds ago      Up 12 seconds
     443/tcp, 0.0.0.0:8080->80/tcp           nginx-edureka
```

Local host port is 8080 and container port is 80

I have a web-application which uses Web Server and Database Server that resides in two separate containers. Is it possible to provide a secure channel between them for their communication?

Solution – Linking Container in Docker

- Linking containers provide a secure channel via which Docker containers can communicate to each other.
- Linking Docker Container implies the following in this scenario:
  - Launch a Docker container that will be running the Database Server.
  - Launch the second Docker container (Web Server) with a link flag to the container launched in Step 1.

This way, the Web Server will be able to talk to the Database Server via the link name.

Linking Container in Docker

1.
2.
3.
4.
5.
6.
7.

- Pull down the redis image from the repository
- Launch a Redis container (named **redis1**) in detached mode as follows:

```
root@docker:~# docker run -d --name redis1 redis
45ee1444549ab502b938324b213caf04545e84b5a2125e4566cec9fbcc28950b
root@docker:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS               NAMES
45ee1444549a        redis               "docker-entrypoint..."   7 seconds ago       Up 7 seconds
     6379/tcp            redis1
55a6dc515b7e        mysql               "docker-entrypoint..."   21 minutes ago      Up 20 minutes
     3306/tcp            database
a114b1a59ff2        tomcat              "catalina.sh run"        52 minutes ago      Up 52 minutes
     8080/tcp            edureka
root@docker:~#
```

Linking Container in Docker

1
2
3
4
5
6
7

- Running another container, a **busybox** container as shown below
- The value provided to the — link flag is

  `link` **`SourceContainerName:ContainerAliasName`**.

- We have chosen the value *redis1* in the **SourceContainerName,** since that was the name that was given to our first container that we launched earlier.
- The **ContainerAliasName** has been selected as *redis* and it could be any name of your choice.
- Launch of container *(redisclient1)* will lead you to the shell prompt.

```
root@docker:~# docker run -it --link redis1:redis --name redisclient1 busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
7520415ce762: Pull complete
Digest: sha256:32f093055929dbc23dec4d03e09dfe971f5973a9ca5cf059cbfb644c206aa83f
Status: Downloaded newer image for busybox:latest
```

Linking Container in Docker

1
2
3
4
5
6
7

- Let us observe first what entry has got added in the `/etc/hosts` file of the `redisclient1` container:
- Notice an entry at the end, where the container **redis1** has got associated with the **redis** name.

```
/ # cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.4      redis 45ee1444549a redis1
172.17.0.5      0d8496a86194
/ # _
```

Linking Container in Docker

1
2
3
4
5
6
7

- Ping by the host name i.e. alias name (**redis**)

```
/ # ping redis
PING redis (172.17.0.4): 56 data bytes
64 bytes from 172.17.0.4: seq=0 ttl=64 time=0.466 ms
64 bytes from 172.17.0.4: seq=1 ttl=64 time=0.085 ms
64 bytes from 172.17.0.4: seq=2 ttl=64 time=0.073 ms
64 bytes from 172.17.0.4: seq=3 ttl=64 time=0.115 ms
64 bytes from 172.17.0.4: seq=4 ttl=64 time=0.074 ms
64 bytes from 172.17.0.4: seq=5 ttl=64 time=0.073 ms
64 bytes from 172.17.0.4: seq=6 ttl=64 time=0.073 ms
64 bytes from 172.17.0.4: seq=7 ttl=64 time=0.075 ms
64 bytes from 172.17.0.4: seq=8 ttl=64 time=0.074 ms
```

Linking Container in Docker

1
2
3
4
5
6
7

- Lets print out the environment variables :

```
/ # set
HISTFILE='/root/.ash_history'
HOME='/root'
HOSTNAME='0d8496a86194'
IFS='
'
OPTIND='1'
PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
PPID='0'
PS1='\w \$ '
PS2='> '
PS4='+ '
PWD='/'
REDIS_ENV_GOSU_VERSION='1.7'
REDIS_ENV_REDIS_DOWNLOAD_SHA1='6780d1abb66f33a97aad0edbe020403d0a15b67f'
REDIS_ENV_REDIS_DOWNLOAD_URL='http://download.redis.io/releases/redis-3.2.8.tar.gz'
REDIS_ENV_REDIS_VERSION='3.2.8'
REDIS_NAME='/redisclient1/redis'
REDIS_PORT='tcp://172.17.0.4:6379'
REDIS_PORT_6379_TCP='tcp://172.17.0.4:6379'
REDIS_PORT_6379_TCP_ADDR='172.17.0.4'
REDIS_PORT_6379_TCP_PORT='6379'
REDIS_PORT_6379_TCP_PROTO='tcp'
SHLVL='1'
TERM='xterm'
_='redis'
/ #
```

- You can see that various environment variables were auto-created for you to help reach out to the **redis1** server from the **redisclient1**.

Linking Container in Docker

1
2
3
4
5
6
7

- Exit Redis container and launch a container based on the redis image
- But this time instead of the default command that will launch the redis server, we will simply go into the shell so that all the redis client tools are ready for us.
- Note that the redis1 server (container) that we launched is still running.

```
root@docker:~# docker run -it --link redis1:redis --name client1 redis sh
# ping redis
PING redis (172.17.0.4): 56 data bytes
64 bytes from 172.17.0.4: icmp_seq=0 ttl=64 time=0.132 ms
64 bytes from 172.17.0.4: icmp_seq=1 ttl=64 time=0.114 ms
64 bytes from 172.17.0.4: icmp_seq=2 ttl=64 time=0.077 ms
64 bytes from 172.17.0.4: icmp_seq=3 ttl=64 time=0.076 ms
^C--- redis ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.076/0.100/0.132/0.024 ms
```

Linking Container in Docker

1
2
3
4
5
6
7

- Next thing is to launch the redis client (**redis-cli**) and connect to our redis server (running in another container and to which we have linked) as given below:
- We have been able to successfully connect to the redis server via the alias name that we specified in the — **link** flag while launching the container.
- If we were running the Redis server on another port (other than the standard 6379) we could have provided the **-p** parameter to the **redis-cli** command and used the value of the environment variable over here (**REDIS_PORT_6379_TCP_PORT**).

```
# redis-cli -h redis
redis:6379> PING
PONG
redis:6379> set myvar DOCKER
OK
redis:6379> get myvar
"DOCKER"
redis:6379> _
```

Linking Container in Docker

1
2
3
4
5
6
7

- Exit the previous container and launch another client (client2) that wants to connect to the same Redis Server that is still running in the first container (containing string key/ value pair)

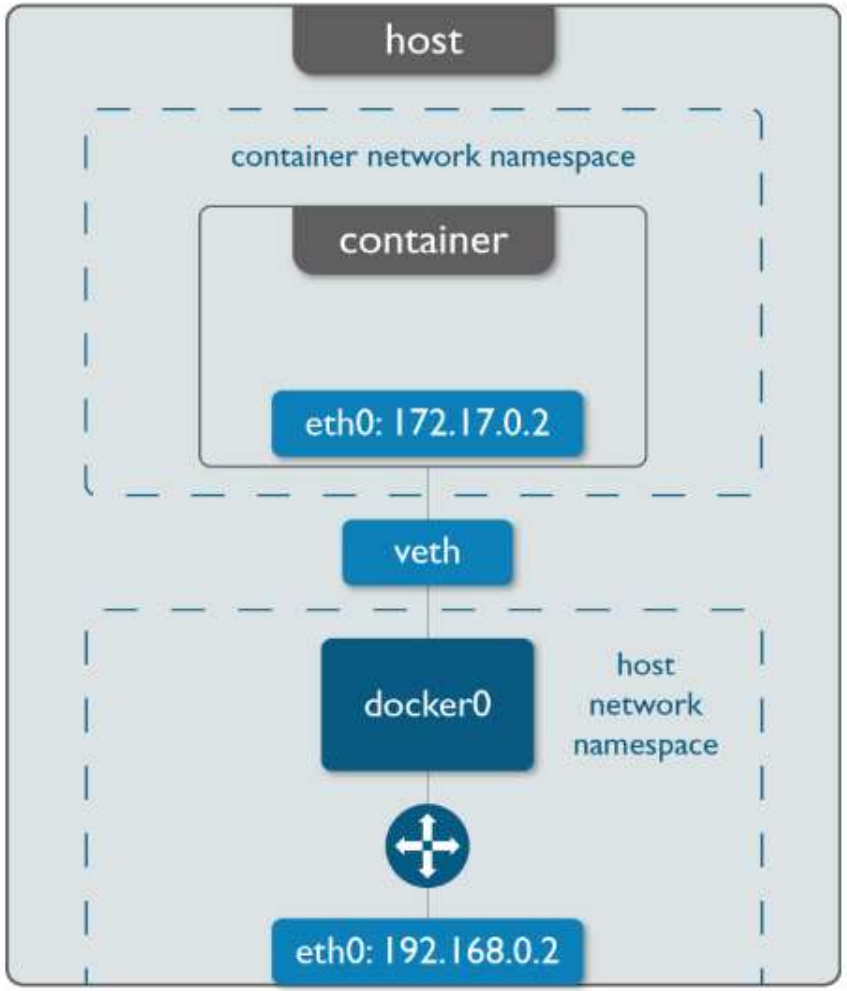
```
# redis-cli -h redis
redis:6379> PING
PONG
redis:6379> set myvar DOCKER
OK
redis:6379> get myvar
"DOCKER"
redis:6379> exit
# exit
root@docker:~# docker run -it --link redis1:redis --name client2 redis sh


# redis-cli -h redis
redis:6379> get myvar
"DOCKER"
redis:6379>
```

- With this *linking between the containers* is now established.

CONTAINER NETWORK : NAMESPACE

What is Network Namespace?

A network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices.



Choosing a Container Network Namespace

- Before starting a container, you should choose a specific network namespace.
- Certain applications that run in containers, may require a different network setup than the default bridge networking, or may not need any networking at all.
- When you start a container with defaults of `docker run` command, the container gets attached to a linux bridge and creates a appropriate network namespace.
- The container can be started with a different type of networking, such as:
  - host networking namespace,
  - no networking namespace,
  - or the networking namespace of another container by using the --net option of docker run

Starting A Container With Network Namespace

- Start a container on a Docker host *without any networking namespace* by using `--network=none`:

```
root@docker:~# docker run -it --network=none ubuntu
root@6030f64efa84:/# _
```

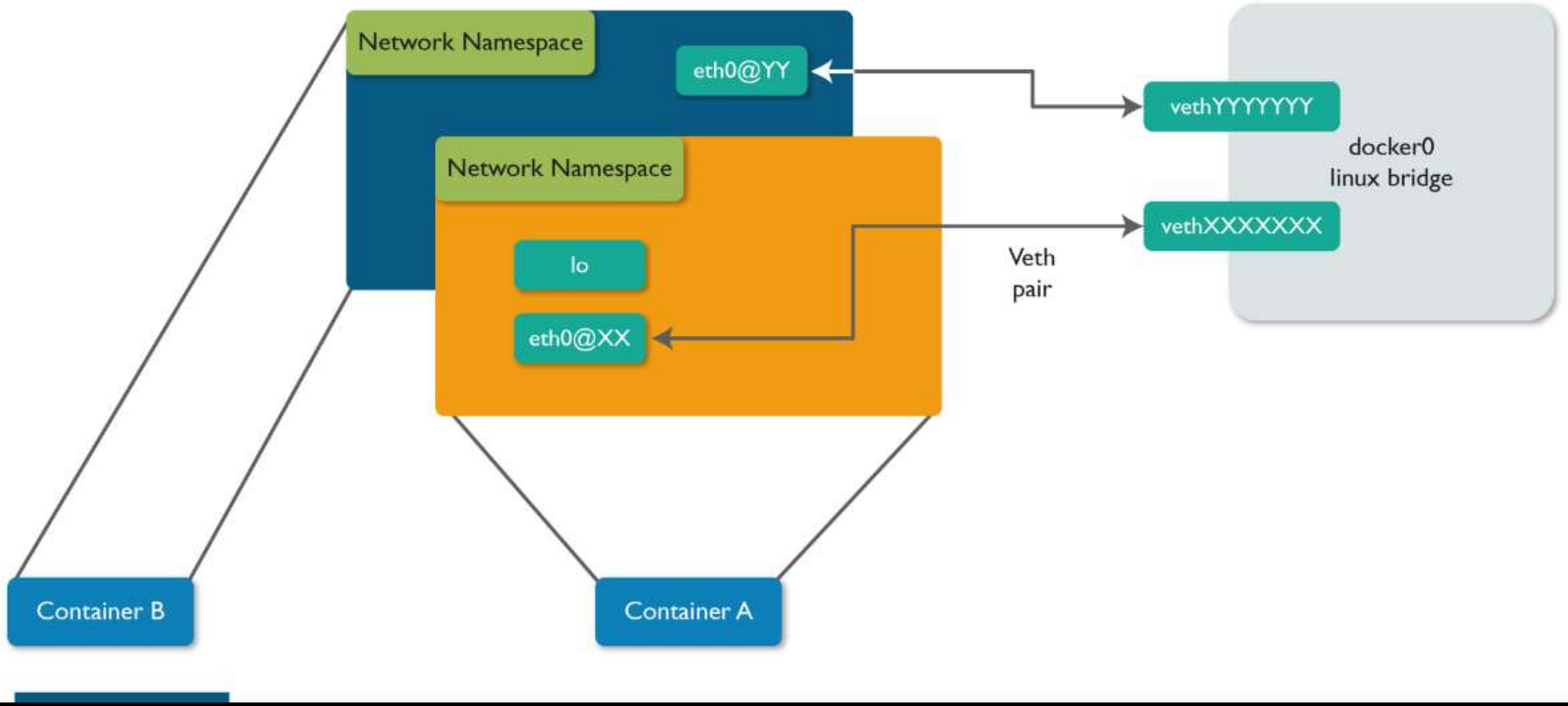
- Start a container on Docker host *with networking namespace* by using `--network=host`

```
root@docker:~# docker run -it --network=host ubuntu
root@docker:/#
```

- Start a container on Docker host with *networking namespace bridge* by using `--network=bridge`

```
root@docker:~# docker run -itd --network=bridge busybox
bdeba72bdff0f1472c59a3ec2d8d4e54bac23fc585c36f7bf67a28d94719db5f
root@docker:~#
```

Network Namespace



How to Access Network Namespace for a given container?

1. Step 1
2. Step 2
3. Step 3

- Get container process id.
- Either run `docker inspect` and look for the Pid under state section or use the following command to extract the Pid field explicitly.

```
root@docker:~# docker run --name box -itd busybox
1afb78d2f33299568ee46ca56c94bb04f19c6ce789fb1271e05db765afc8667f
root@docker:~# pid="$(docker inspect -f '{{.State.Pid}}' "box")"
root@docker:~# echo $pid
9163
```

How to Access Network Namespace for a given container?

1. Step 1
2. Step 2
3. Step 3

- Soft link (symlink) the network namespace of the process from the `/proc` directory into the `/var/run` directory as shown below.
- Note that you may need to create the directory netns under `/var/run` before symlinking the process network namespace.

```
root@docker:~# mkdir -p /var/run/netns
root@docker:~# ln -s /proc/$pid/ns/net /var/run/netns/box
```

How to Access Network Namespace for a given container?

1. Step 1
2. Step 2
3. Step 3

- The network namespace can be listed and accessed using the command in the following sequence:
  - `ip netns`
  - `ip netns exec (netns_name) (command)`

```
root@docker:~# ip netns
box (id: 5)

root@docker:~# ip netns exec box ip a
```

Summary – Accessing Network Namespace for a Given Container

```
root@docker:~# docker run --name box -itd busybox
1afb78d2f33299568ee46ca56c94bb04f19c6ce789fb1271e05db765afc8667f
root@docker:~# pid="$(docker inspect -f '{{.State.Pid}}' "box")"
root@docker:~# echo $pid
9163
root@docker:~# mkdir -p /var/run/netns
root@docker:~# ln -s /proc/$pid/ns/net /var/run/netns/box
root@docker:~# ip netns
box (id: 5)

root@docker:~# ip netns exec box ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
30: eth0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:07 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.7/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:7/64 scope link
       valid_lft forever preferred_lft forever
root@docker:~# _
```

Docker Container Networking

Communicating to the Outside World

- Whether a container can talk to the world is governed by two factors:
  - The first factor is whether the host machine is forwarding its IP packets.
  - The second is whether the host's iptables allow this particular connection.

Configuring the IP Tables and IP Forwarding Settings – Query

I don't want the default Docker Daemon to turn on the IP forwarding and modify my IP tables. How can I get more control on how the traffic flows in the host, between the containers and the outside world?

Configuring the IP Tables and IP Forwarding Settings – Answer

Docker Behavior is customizable when you start the Docker daemon with the --ipforward=false, --iptables=false options. This recipe shows you to make those customizations. To try this, stop the Docker daemon on the host that you are using. On Ubuntu systems, edit */etc/default/docker* and set the options to false

Configuring the IP Tables and IP Forwarding Settings

- With this configuration, traffic on the Docker bridge *docker0* will not be forwarded to the other networking interfaces and the post-routing masquerading rule will not be present.
- This means that all outbound connectivity from your containers to the outside world will be dropped.

```
root@test01:~# service docker stop
docker stop/waiting
root@test01:~# sudo su
root@test01:~# echo DOCKER_OPTS=\"--iptables=false --ip-forward=false\">> /etc/def
ault/docker
root@test01:~# service docker restart
stop: Unknown instance:
docker start/running, process 5146
root@test01:~# iptables -t nat -D POSTROUTING 1
root@test01:~# echo 0 > /proc/sys/net/ipv4/ip_forward
root@test01:~# service docker restart
docker stop/waiting
docker start/running, process 6071
root@test01:~# docker run -it --rm ubuntu
WARNING: IPv4 forwarding is disabled. Networking will not work.
```

Configuring the IP Tables and IP Forwarding Settings – Manually

- To re-enable communication to the outside manually, enable IP forwarding and set the post-routing rule on the Docker host like so:

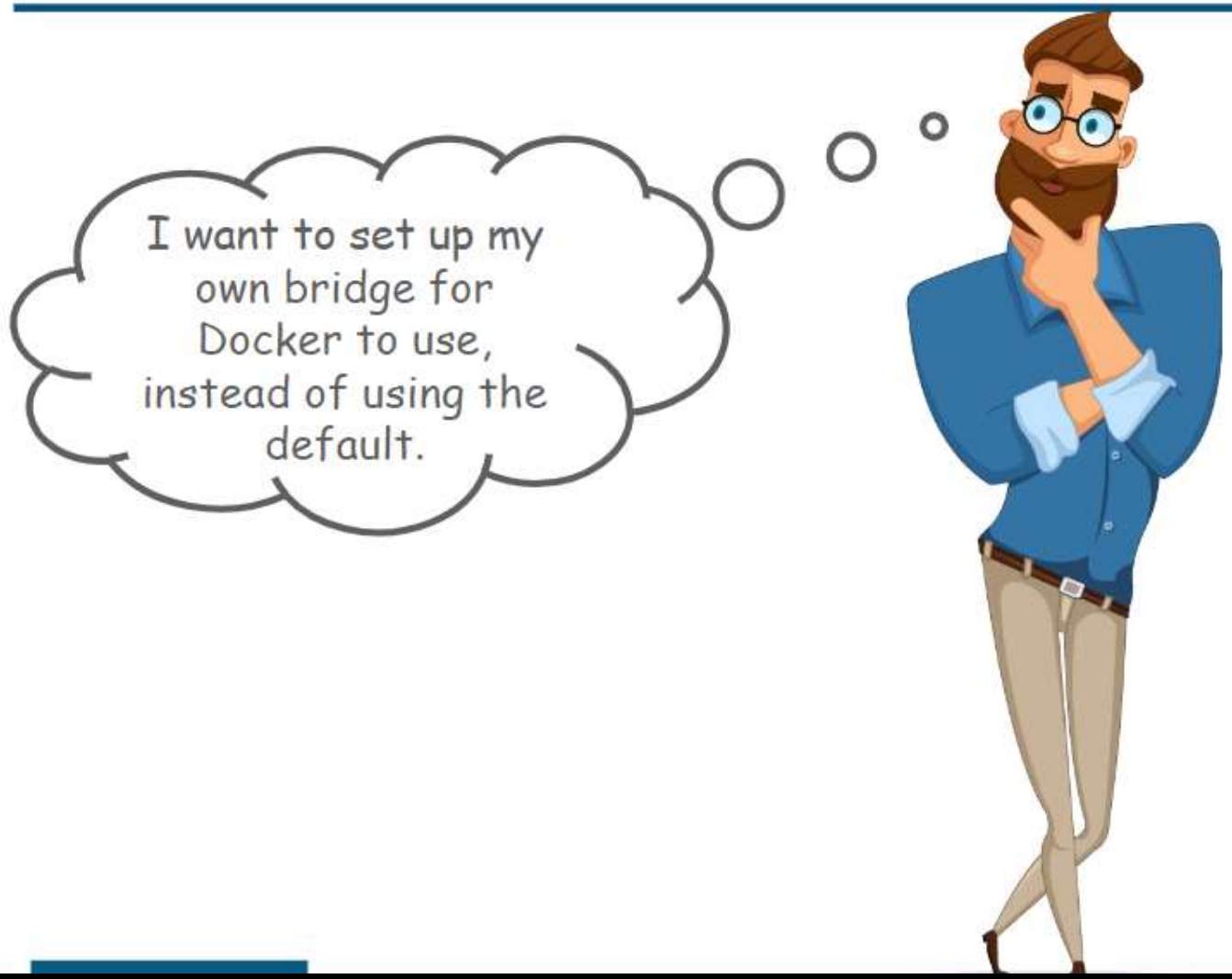
```
root@test01:~# echo 1 > /proc/sys/net/ipv4/ip_forward
root@test01:~# iptables -t nat -A POSTROUTING -s 172.17.0.0/16 -j MASQUERADE
```

Note:

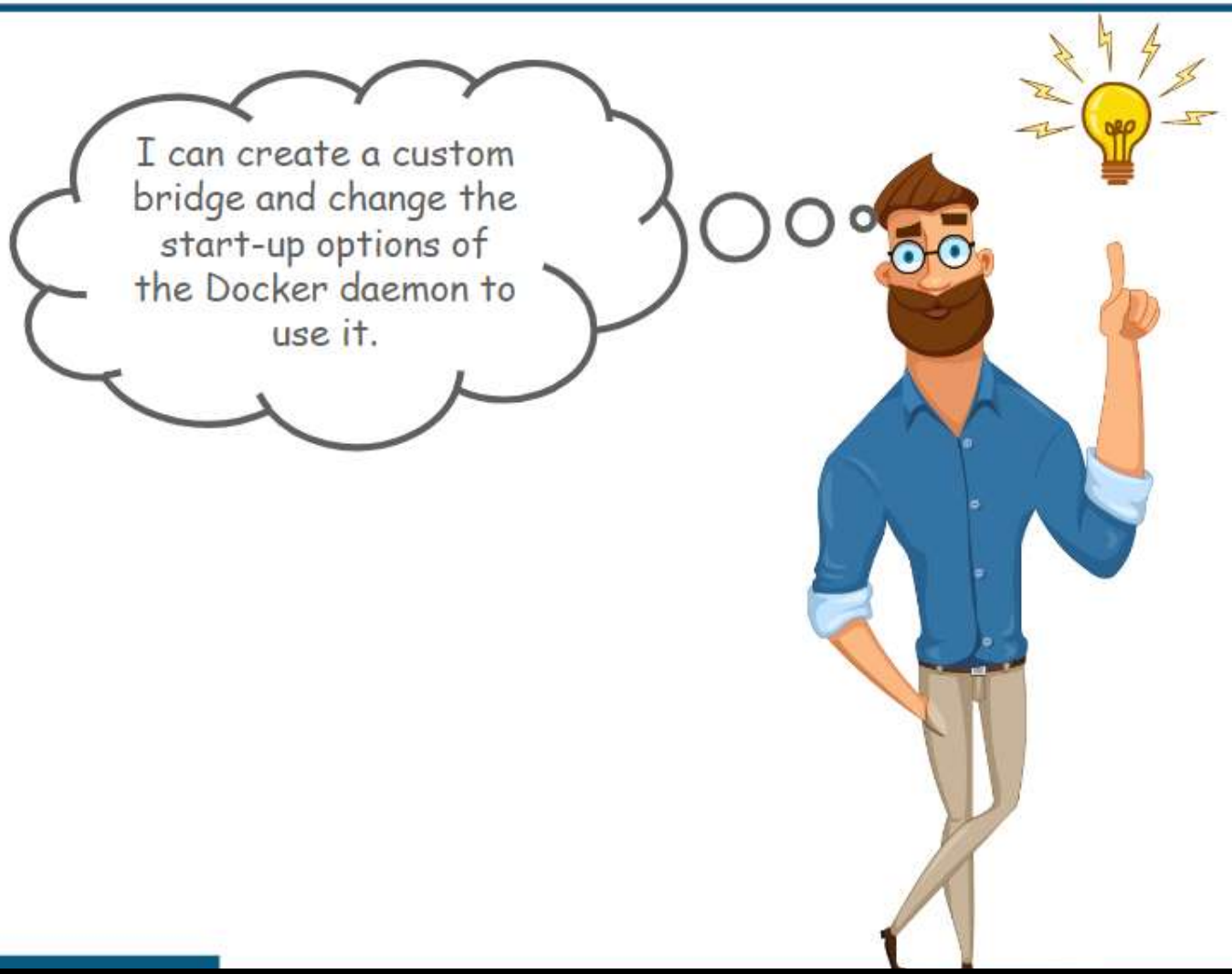
- By default the Docker daemon is allowed to manage the IP table rules on the Docker host. This means that it can add rules that restrict traffic between containers and provide network isolation between them.
- If you disallow Docker to manipulate the IP table rules, it will not be able to add rules that restrict traffic between containers.

CUSTOM BRIDGE

Query


I want to set up my own bridge for Docker to use, instead of using the default.

Solution – Setting Up a Custom Bridge



Setting Up a Custom Bridge – Steps

- Turn off the Docker daemon, delete the docker0 bridge created by default, and create a new bridge called *edureka*.

```
root@test01:~# service docker stop
docker stop/waiting
root@test01:~# ip link set docker0 down
root@test01:~# brctl delbr docker0
root@test01:~# brctl addbr edureka
root@test01:~# ip link set edureka up
root@test01:~# ip addr add 10.0.0.1/24 dev edureka
root@test01:~#
```

- The *edureka* bridge is up, you can now edit the Docker daemon configuration file and restart the daemon.

```
root@test01:~# echo 'DOCKER_OPTS="-b=edureka"' >> /etc/default/docker
root@test01:~# service docker restart
```

Setting Up a Custom Bridge – NAT Rule

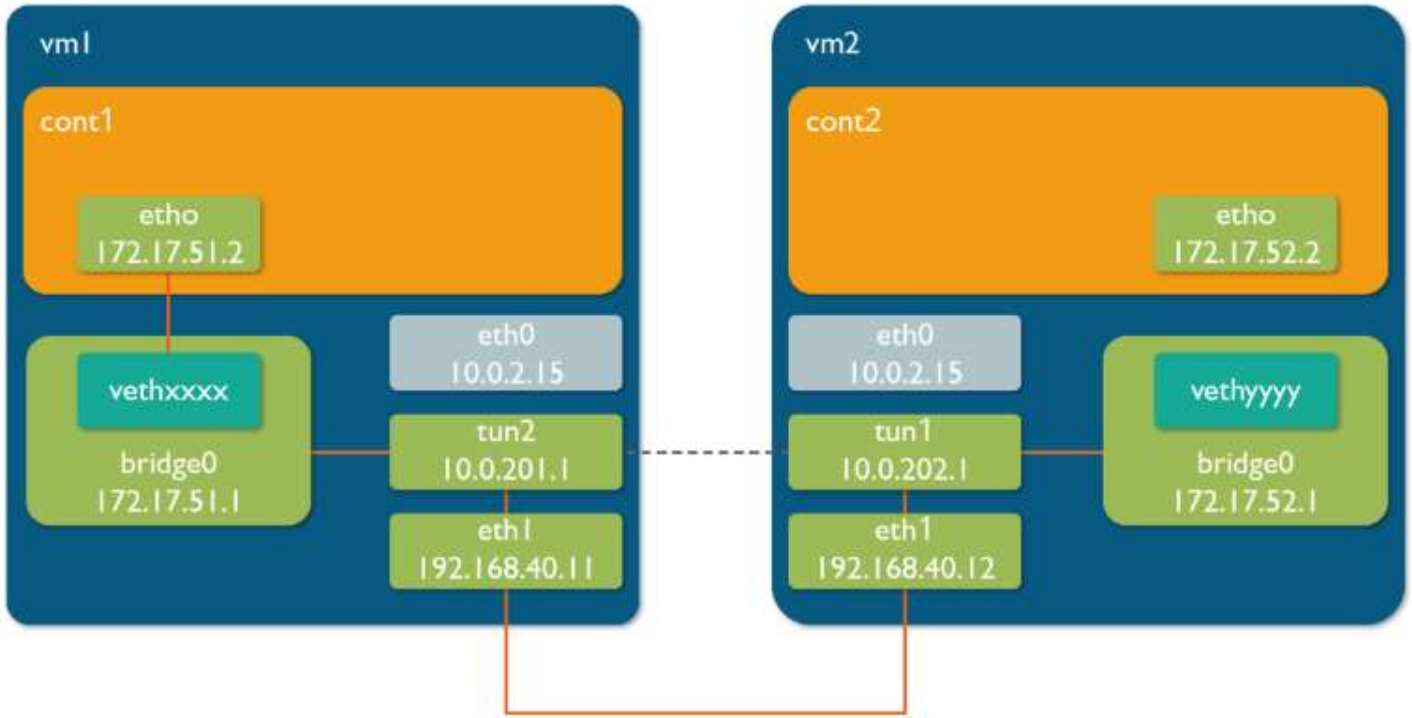
- When the custom bridge is created, Docker also creates the NAT rule for this bridge
- NAT rules allow the rewriting of the source address of traffic

```
root@test01:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  anywhere             anywhere             ADDRTYPE match dst-t
ype LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  anywhere            !127.0.0.0/8          ADDRTYPE match dst-t
ype LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  10.0.0.0/24          anywhere
MASQUERADE  all  --  172.17.0.0/16        anywhere

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
```

Generic Routing Encapsulation (GRE) TUNNEL

Generic Routing Encapsulation (GRE) Tunnel

- GRE is a tunnelling protocol that can encapsulate a wide variety of network layer protocols inside virtual point-to-point links.
- The main idea is to create a GRE tunnel between the VMs and send all traffic through it:



Creating a GRE Tunnel

- In order to create a tunnel you need to specify the name, the type (which is gre in our case) and the IP address of local and the remote end.
- So as per the below screen shot, tun2 is the name used for the tunnel on vm1 as it acts as the tunnel endpoint, which leads to vm2 and every packet sent to tun2 will eventually come out on vm2 end.

```
#GRE tunnel config execute on vm1
sudo iptunnel add tun2 mode gre local 192.168.40.11 remote 192.168.40.12
sudo ifconfig tun2 10.0.201.1
sudo ifconfig tun2 up
sudo route add -net 172.17.52.0 netmask 255.255.255.0 dev tun2
sudo iptables -t nat -F POSTROUTING
sudo iptables -t nat -A POSTROUTING -s 172.17.51.0/24 ! -d 172.17.0.0/16 -j MASQUERADE
sudo docker run -it --rm --name cont1 ubuntu /bin/bash
#Inside the container (cont1)
nc -l 3333
```

GRE Tunnel Configured on vm2

- The below commands execute the GRE tunnel config on a vm2

```
#GRE tunnel config execute on vm2
sudo iptunnel add tun1 mode gre 192.168.40.12 remote 192.168.40.11
sudo ifconfig tun1 10.0.202.1
sudo ifconfig tun1 up
sudo route add -net 172.17.51.0 netmask 255.255.255.0 dev tun1
sudo iptables -t nat -F POSTROUTING
sudo iptables -t nat -A POSTROUTING -s 172.17.52.0/24 ! -d 172.17.0.0/16 -j MASQUERADE
sudo docker run -it --rm --name cont2 ubuntu /bin/bash
#Inside the container (cont2)
nc -w 1 -v 172.17.51.2 3333
#Result: Connection to 172.17.51.2 3333 port [tcp/*] succeeded!
```

- Connection is established successfully

Viewing the logs

- Sudo docker run -itd alpine ping google.com
- Sudo docker ps
- Sudo docker logs <container-id/container-name> would show the logs
- Sudo docker logs -f <container-id/container-name> would show live logs
- Sudo docker logs –timestamps <container-id/container-name> would show logs along with timestamp
- Sudo docker logs –tail 3 <container-id/container-name> would tail the logs and show only last 3 lines
- To check the log file use sudo docker inspect <container-id/container-name>
- To redirect container logs to other files use
  - Sudo docker logs -f container-id/container-name > output.log 2> error.log
  - Where output.log would contain the stdout and error.log would contain the stderr

Looking at Processes

- Sudo docker top <container-id/container-name> ps: this would show all processes running in that container
- Sudo docker top <container-id/container-name> -a: this would show all processes stopped in that container