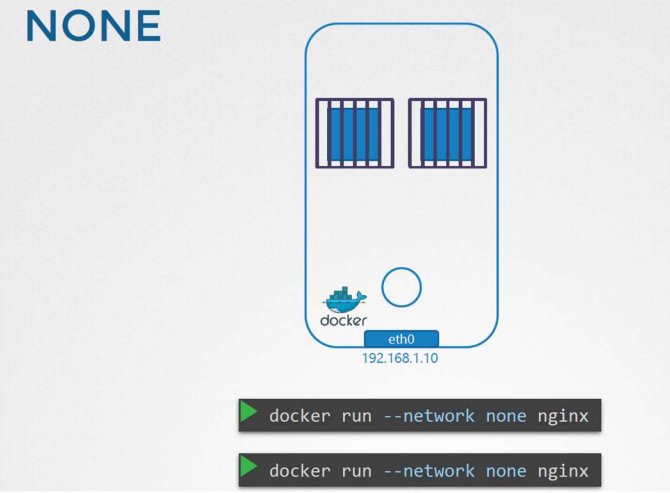
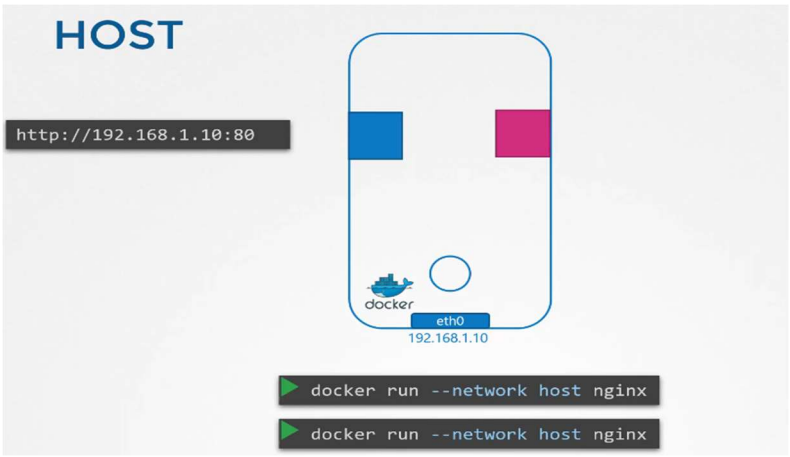
In this lecture, we look at networking in Docker. We will start with the basic networking options in Docker and then try and relate it to the concepts around networking namespaces. Let's start with a single Docker host, a server with Docker installed on it. It has an internet interface at 80 that connects to the local network with the IP address 192.168.1.10. When you run a container, you have different networking options to choose from.

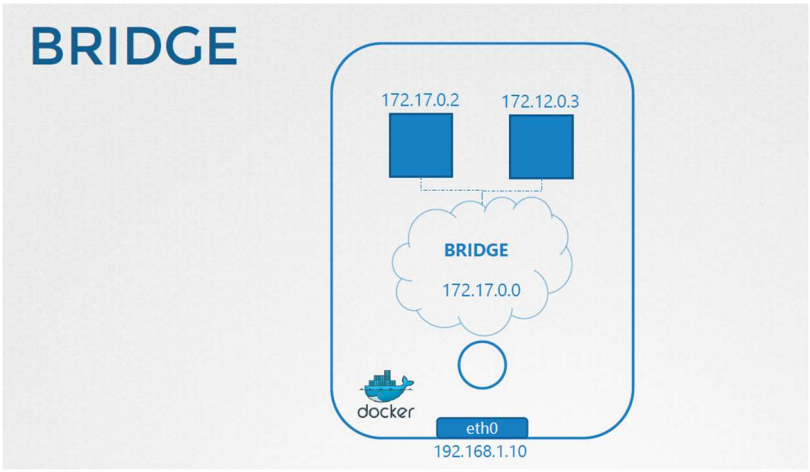
First, let's see the none network. With a none network, the Docker container is not attached to any network. The container cannot reach the outside world, and no one from the outside world can reach the container. If you're on multiple containers, they are all created without being part of any network and cannot talk to each other or to the outside world.



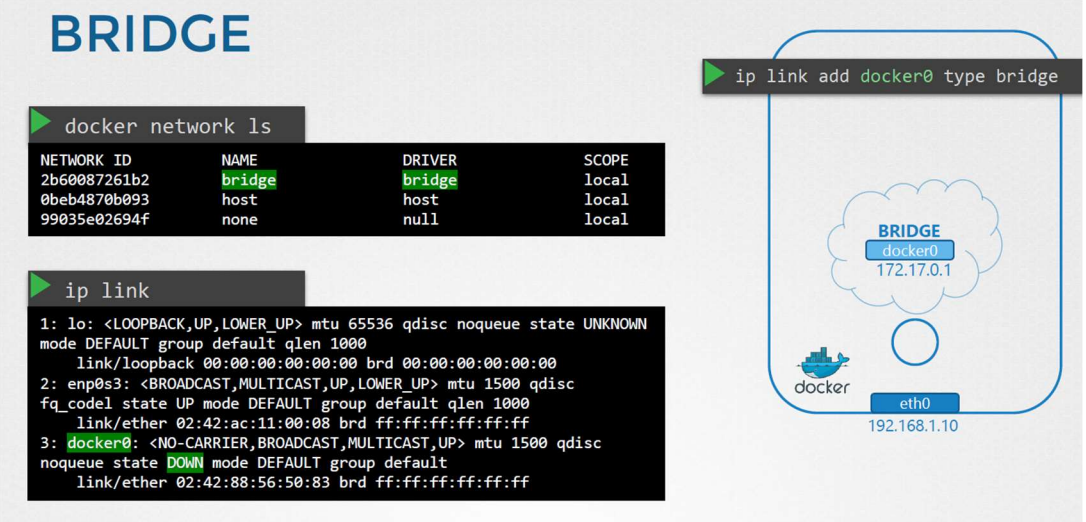
Next is the host network. With the host network, the container is attached to the host network. There is no network isolation between the host and the container. If you deploy a web application listening on port 80 in the container, then the web application is available on port 80 on the host without having to do any additional port mapping. If you try to run another instance of the same container that listens on the same port, it won't work as they share the host networking, and two processes cannot listen on the same port at the same time.



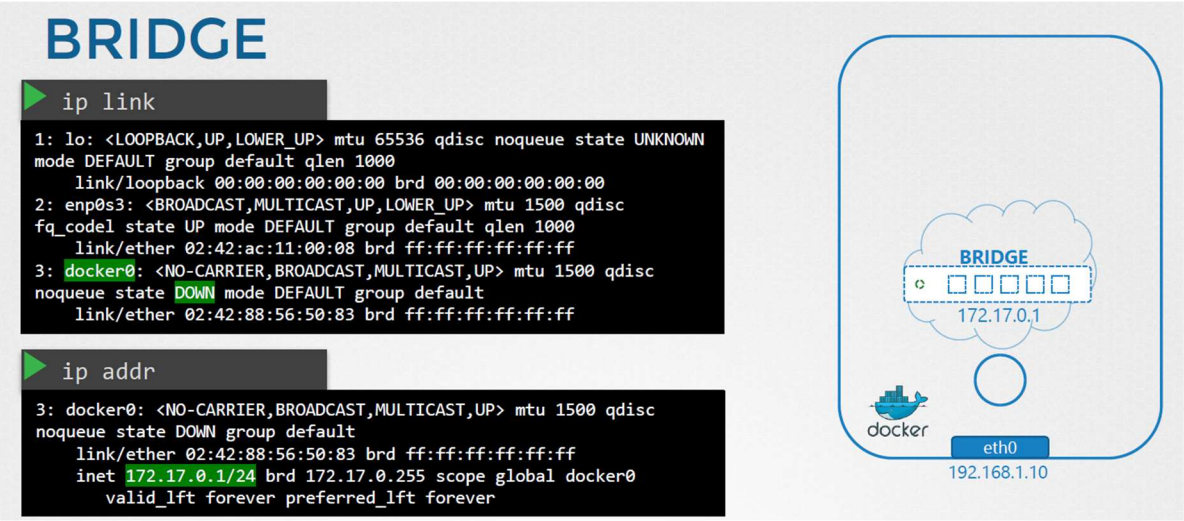
The third networking option is the bridge. In this case, an internal private network is created which the Docker host and containers attach to. The network has an address 172.17.0.0 by default, and each device connecting to this network get their own internal private network address on this network. This is the network that we are most interested in, so we will take a deeper look at how exactly Docker creates and manages this network.



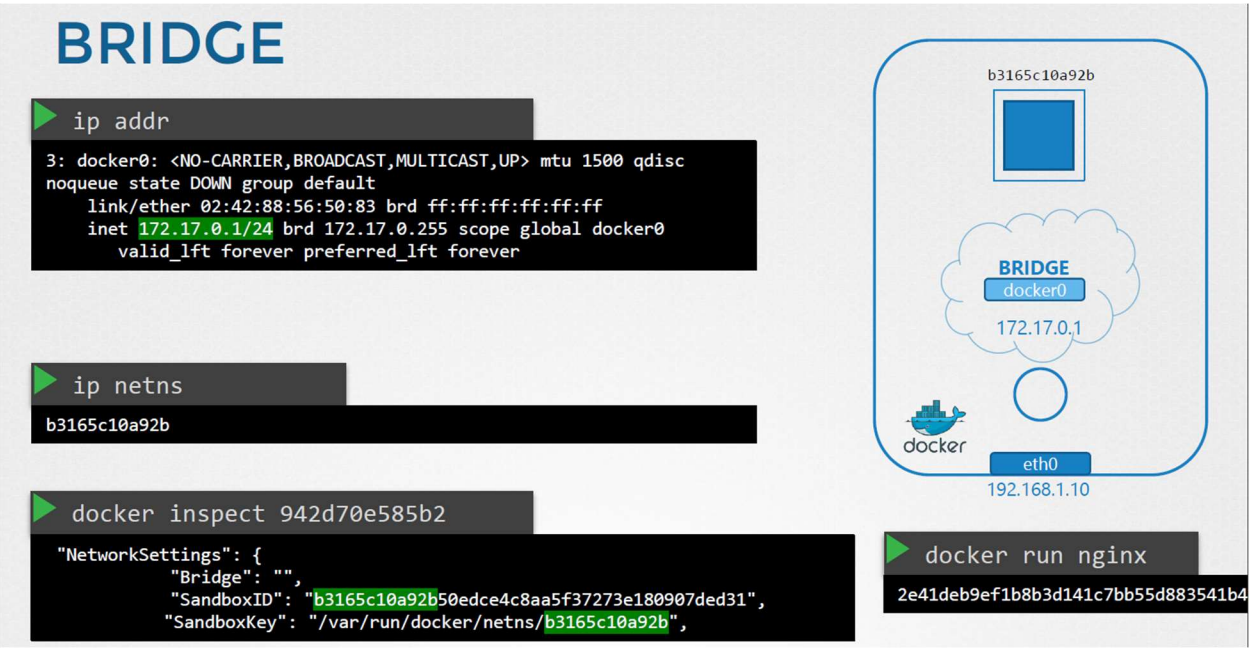
When Docker is installed on the host, it creates an internal private network called Bridge by default. You can see this when you're the Docker network ls command. Now, Docker calls the network by the name Bridge, but on the host, the network is created by the name Docker0. You can see this in the output of the IP link command. Docker internally uses a technique similar to what we saw in the video on namespaces by running the IP link add command with the type set to bridge. So, remember, the name bridge in the Docker network ls output refers to the name Docker0 on the host. They are one and the same thing. Also note that the interface, or network, is currently down.



Now, remember, we said that the bridge network is like an interface to the host but a switch to the namespaces or containers within the host. So the interface Docker0 on the host is assigned an IP 172.17.0.1. You can see this in the output of the IP ADDR command.



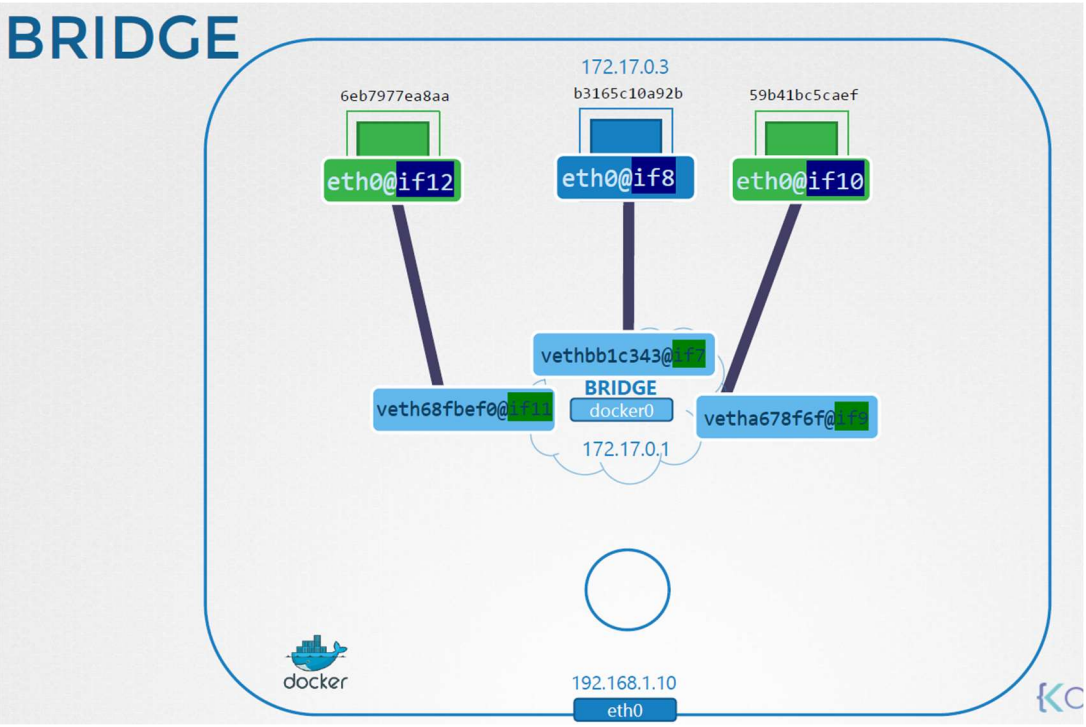
Whenever a container is created, Docker creates a network namespace for it. Just like how we created network namespaces in the previous video. Run the IP netns command to list the namespace. Note that there is a minor hack to be done to get the IP netns command to list the namespaces created by Docker. Check out the resources section of this lecture for information on that. The namespace has the name starting B3165. You can see the namespace associated with each container in the output of the Docker inspect command.



So how does Docker attach the container to the bridge? As we did before, it creates a cable, a virtual cable, with two interfaces on each end. Let's find out what Docker has created here. If you run the IP link command on the Docker host, you see one end of the interface which is attached to the local bridge, Docker0. If you run the same command again, this time with the dash end option with a namespace, then it lists the other end of the interface within the container namespace.

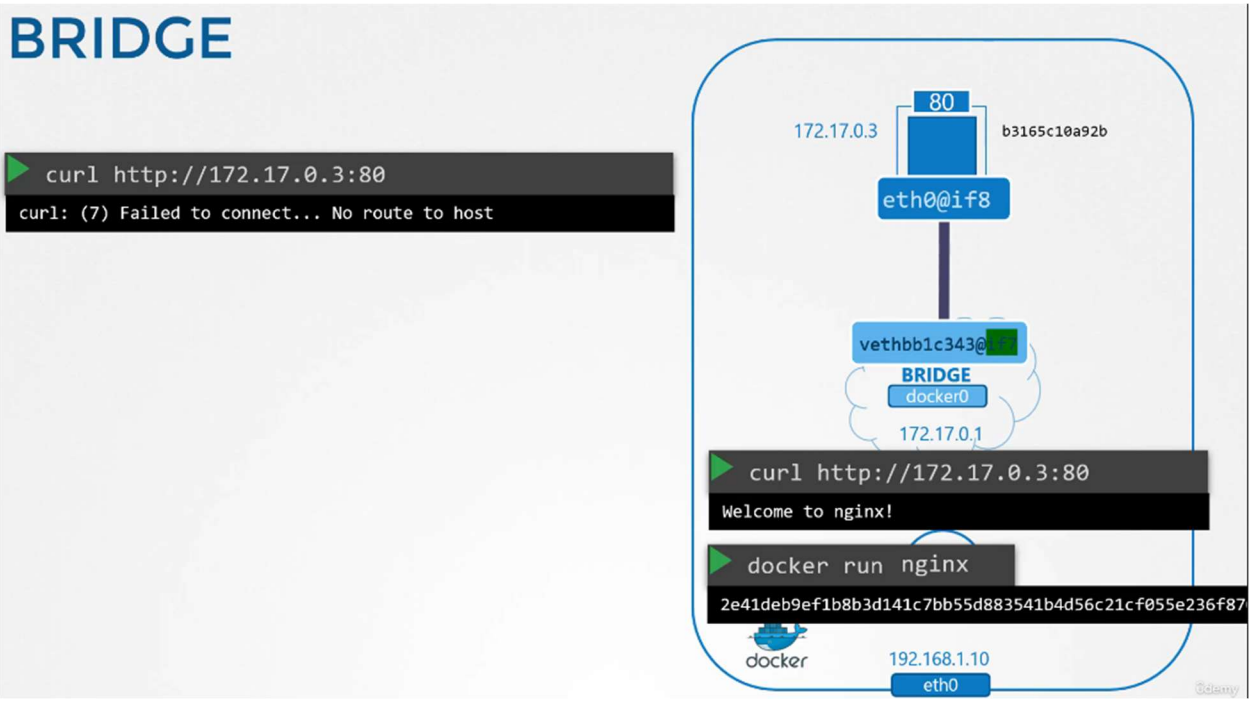
The interface also gets an IP assigned within the network. You can view this by running the IP ADDR command, but within the container's namespace. The container gets assigned 172.17.0.3

The same procedure is followed every time a new container is created. Docker creates a namespace, creates a pair of interfaces, attaches one end to the container and another end to the bridge network. The interface pairs can be identified using their numbers. Odd and even form a pair. Nine and ten are one pair. Seven and eight are another. Eleven and 12 are one pair. The containers are all part of the network now. They can all communicate with each other.

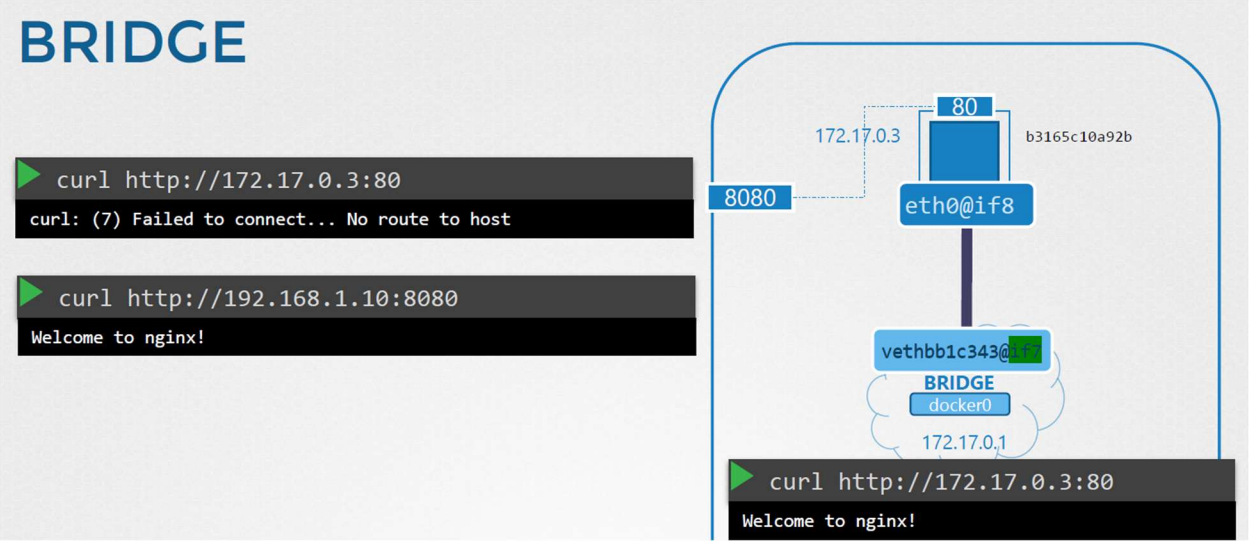


Let us look at port mapping now. The container we created is Nginx, so it's a web application serving webpage on port 80. Since our container is within a private network inside the host, only other containers in the same network, or the host itself, can access this webpage. If you try to access the webpage using

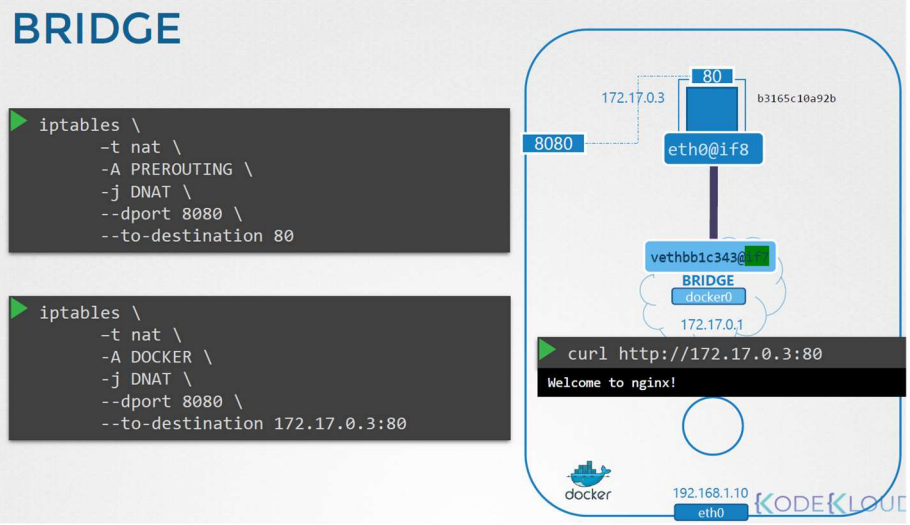
curl with the IP of the container from within Docker host on port 80, you will see the webpage. If you try to do the same thing outside the host, you cannot view the webpage.



To allow external users to access the applications hosted on containers, Docker provides a port publishing, or port mapping, option. When you run containers, tell Docker to map port 8080 on the Docker host to port 80 on the container. With that done, you could access the web application using the IP of the Docker host and port 8080. Any traffic to port 8080 on the Docker host will be forwarded to port 80 on the container. Now all of your external users and other applications or servers can use this URL to access the application deployed on the host.



But how does Docker do that? How does it forward traffic from one port to another? Well, what would you do? Let's forget about Docker and everything else for a second. The requirement is to forward traffic coming in on one port to another port on the server. We talked about it in one of our prerequisite lectures. We create a NAT rule for that. Using IP tables, we create an entry into the NATs table to append the rules to the prerouting chain to change the destination port from 8080 to 80. Docker does it the same way. Docker adds the rule to the Docker chain and sets destination to include the container's IP as well.



You can see the rule Docker creates when you list the rules in IP tables.