**Docker Networks**

**===============**

--> Docker networks are used to establish communication between containers.

--> If both the containers are same network we can establish the connection.

--> If you are not mention any network, while creating a container, It will be created in "default bridge network".

--> By default docker have three networks

1. bridge

2. host

3. none

**1. Bridge Network**

**Overview:** The bridge network is the default network driver used when you don't specify a network driver. It is suitable for creating isolated networks on a single Docker host.

**How It Works:**

* **Single Host:** Containers connected to the same bridge network can communicate with each other, but containers on different bridge networks or different hosts cannot.
* **NAT:** Uses Network Address Translation (NAT) to provide connectivity to external networks.

**Advantages:**

* **Isolation: Containers on different bridge networks are isolated from each other.**
* **Simplicity: Easy to use and provides basic networking functionality.**

**Disadvantages:**

* **Single Host Limitation: Cannot be used to connect containers across different Docker hosts.**

**2. Host Network**

**Overview:** The host network driver removes network isolation between the container and the Docker host. Containers use the host's network stack directly.

**How It Works:**

* **Direct Access:** Containers share the host's network interfaces and IP address.
* **No NAT:** Containers can use the host's network interfaces and ports directly without NAT.

**Advantages:**

* **Performance:** Provides better performance for network-intensive applications by avoiding the overhead of NAT.
* **Access:** Containers can access the host's network interfaces and services directly.

**Disadvantages:**

* **Security:** Less isolation between the container and the host, which may pose security risks.
* **Port Conflicts:** Containers and the host might compete for the same network ports.

**3. None Networks**

**Overview:** The none network driver creates a container that has no network connectivity. It is used for cases where a container should not connect to any network.

**How None Networks Work:**

* **Isolated Container:** Containers connected to the none network have no network interfaces, meaning they cannot send or receive network traffic.
* **Use Case:** This network type is useful for containers that don't need network access or for specific testing scenarios.

**Advantages:**

* **Security:** Ensures that a container cannot communicate over the network, which may be required for specific security or isolation reasons.
* **Simplicity:** Provides a straightforward way to disable network functionality.

**Disadvantages:**

* **No Network Communication:** The container cannot interact with other containers or external services, which limits its functionality.

**4. Overlay Networks**

**Overview:** Overlay networks are designed to enable communication between Docker containers across different Docker hosts. They are useful in multi-host Docker setups, such as Docker Swarm or Kubernetes clusters.

**How Overlay Networks Work:**

* **Virtual Networks:** Overlay networks create a virtual network that spans multiple Docker hosts. Containers on different hosts can communicate as if they were on the same local network.
* **Encapsulation:** They use encapsulation technologies, such as VXLAN (Virtual Extensible LAN), to tunnel network traffic between hosts.
* **Network Swarm Services:** Overlay networks are typically used with Docker Swarm services, allowing containers in a Swarm cluster to communicate with each other across different nodes.

**Advantages:**

* **Cross-Host Communication:** Allows containers on different Docker hosts to communicate securely and efficiently.
* **Service Discovery:** Provides service discovery and load balancing when used with Docker Swarm or Kubernetes.

**Disadvantages:**

* **Performance Overhead:** The encapsulation and tunneling process may introduce additional overhead compared to local networks.
* **Complexity:** Requires Docker Swarm or other orchestration tools for management and operation.

**5. Macvlan Network**

**Overview:** The macvlan network driver allows you to assign a unique MAC address to each container, making it appear as a physical network interface on the network.

**How It Works:**

* **Physical Network:** Containers get their own IP addresses on the network and can communicate directly with other devices on the network.
* **Isolation:** Containers appear as separate devices on the physical network.

**Advantages:**

* **Network Visibility:** Containers appear as separate devices on the physical network.
* **Direct Communication:** Allows containers to communicate directly with other devices on the network.

**Disadvantages:**

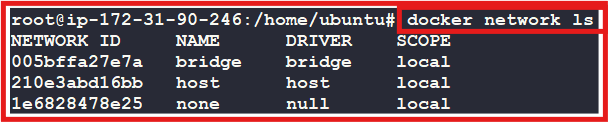
* **Complexity:** Requires additional configuration and might not be suitable for all environments.
* **Limited Support:** Not supported by all Docker setups or network interfaces.

1. **Basic Docker Network Commands**

1.1. Listing Docker Networks:

* **Command:** **docker network ls**
* **Example:** **docker network ls**

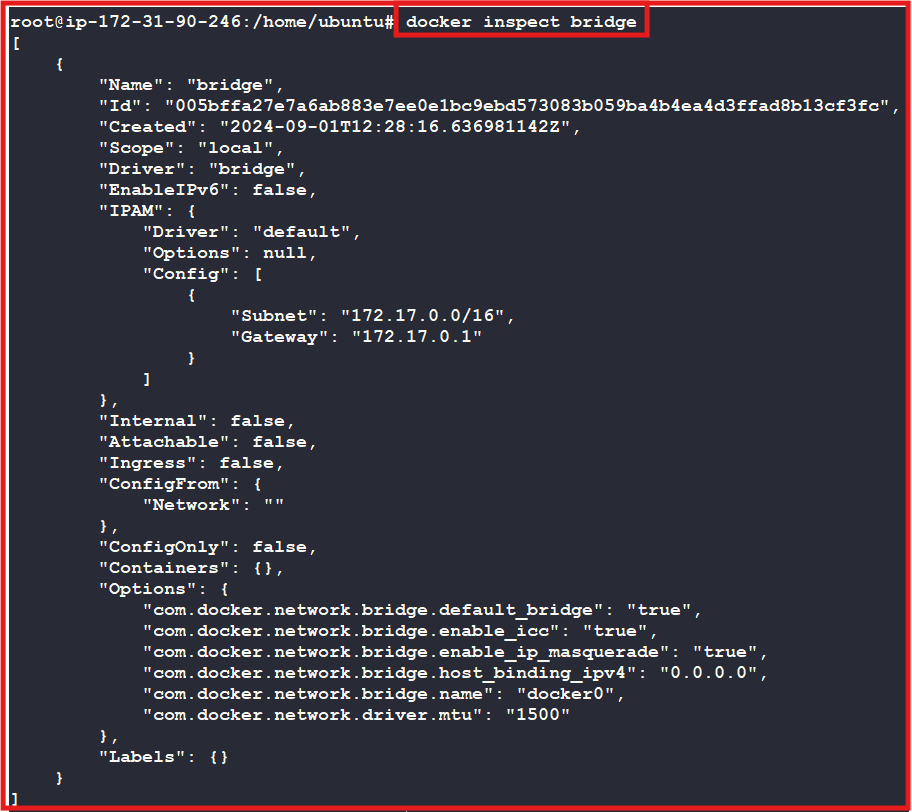
This lists all existing Docker networks, including default networks (bridge, host, none).



**1.2. Inspecting a Docker Network**

* **Command:** **docker network inspect [network\_name]**
* **Example:** **docker network inspect bridge**

This provides detailed information about the bridge network, including connected containers, IP ranges, and settings.

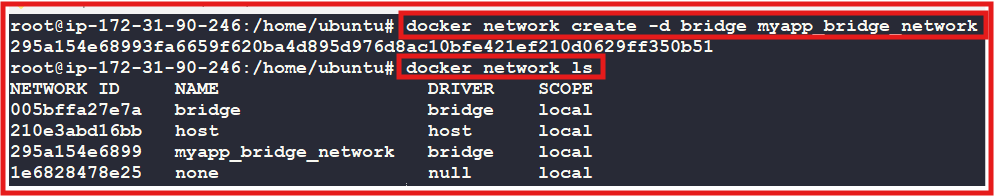


**1.3. Creating a Docker Network**

* **Syntax: docker network create -d <driver> <networkname>**
* **Example: docker network create -d bridge myapp\_bridge\_network**

This creates a custom bridge network called my\_bridge\_network.

Note: If we do not mention -d option default driver will be ‘bridge’.



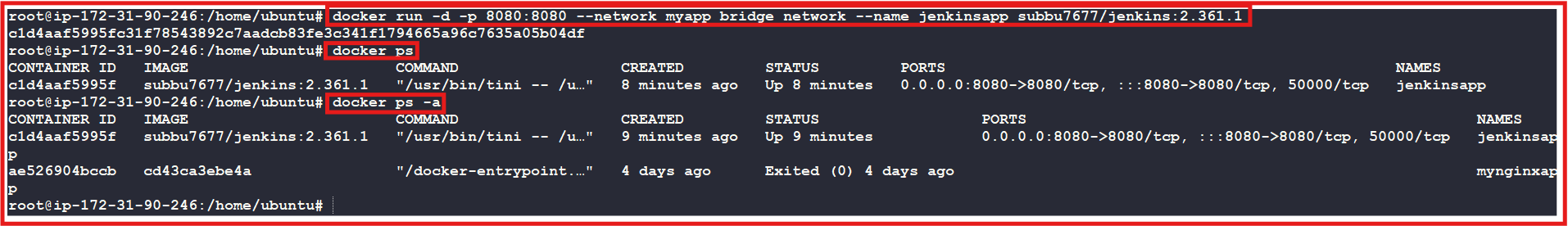
**1.4. Creating a containers in a network**

* **Syntax:**

**docker run -d -p <** **host port>:<** **container port> --network <network-name> --name <container-name> <Repositoryname/imagename:tag>**

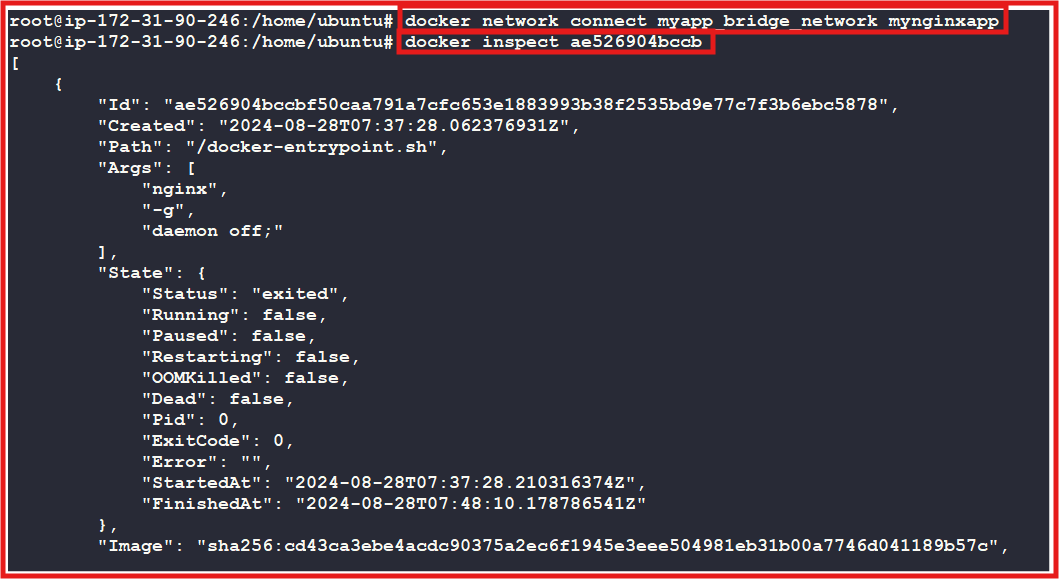
* **Example-1:**

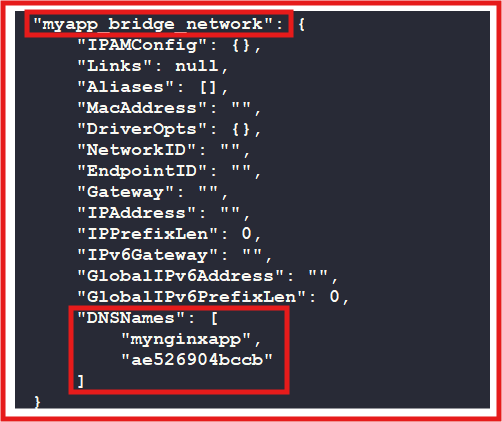
**docker run -d -p 8080:8080 --network myapp\_bridge\_network --name jenkinsapp subbu7677/jenkins:2.361.1**

****

**1.5 Connecting a network to an existing container:**

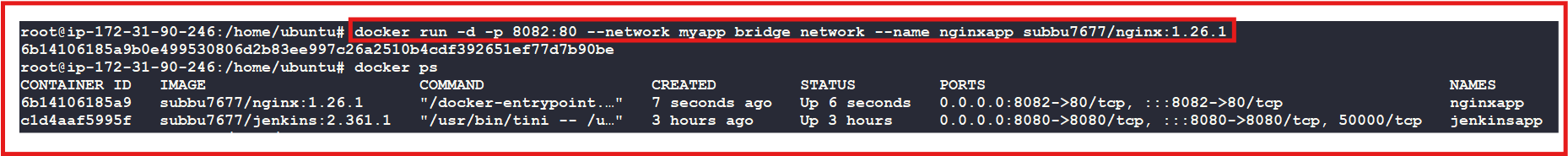
**Syntax: docker network connect [network\_name] [container\_name]**

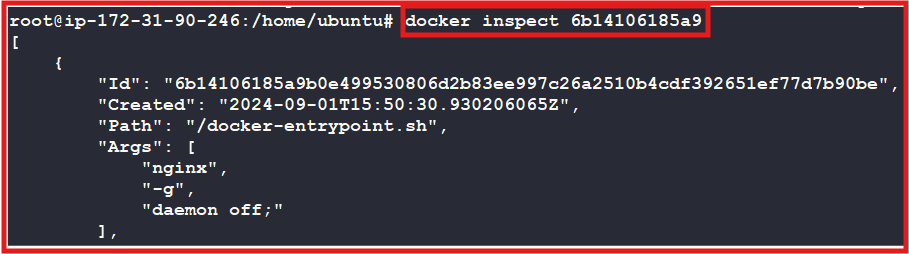
****

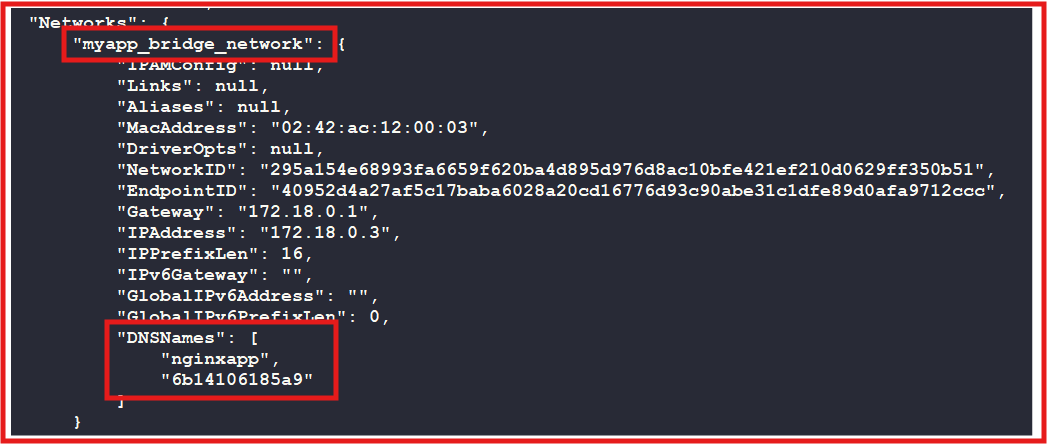
****

* **Example-2:**

**docker run -d -p 8082:80 --network myapp\_bridge\_network --name nginxapp subbu7677/nginx:1.26.1**

****

****

****

**Now jenkinsapp and centosapp are in the same network. We will check**

1. **Jenkinsapp container: 172.18.0.2**
2. **nginxapp container: 172.18.0.3**

**How to check both the containers are connected or not?**

**-------------------------------------------------------**

**step 1: create two containers.**

**step 2: go inside the container one**

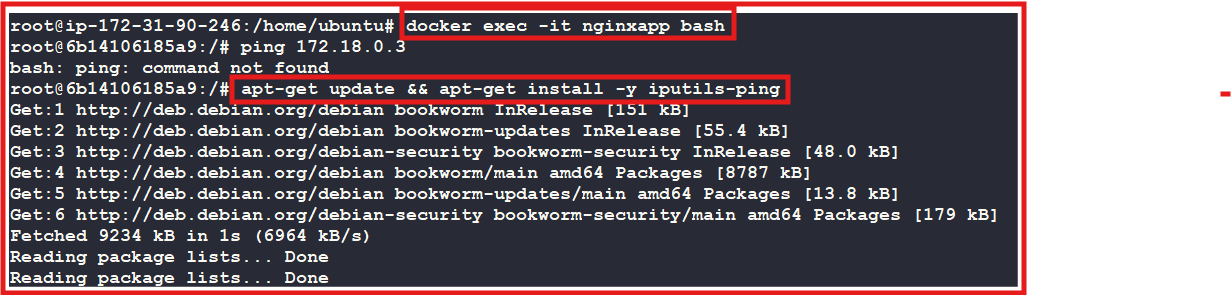
**docker exec -it nginxapp bash**

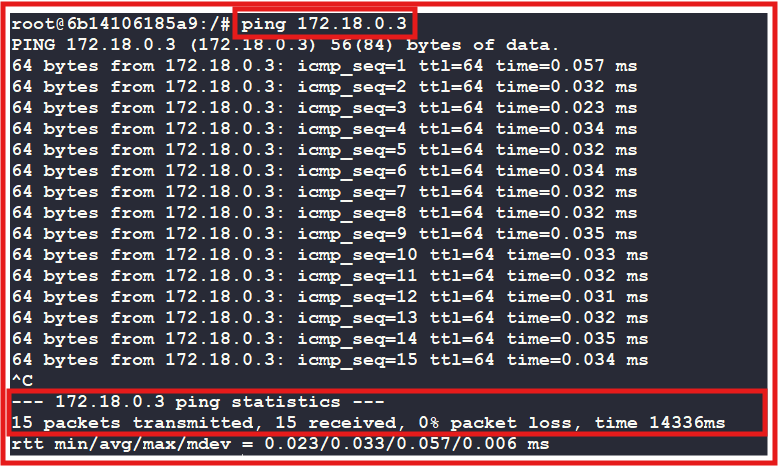
**apt-get update && apt-get install -y iputils-ping**

**step 3: ping with another container ip**

**ping 72.18.0.3**

**curl -v telnet:// 72.18.0.3:8080**

****

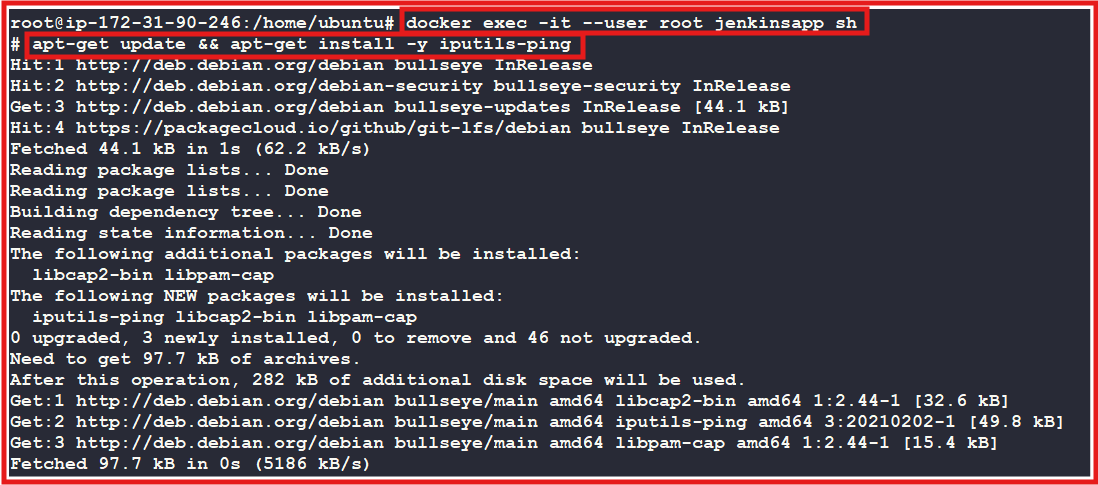
****

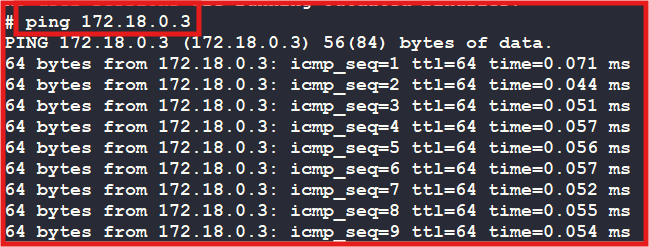
Now go into Jenkins container and check whether connection with nginx container is established or not.

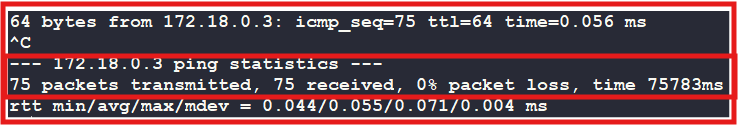
**docker exec -it --user root jenkinsapp sh**

**apt-get update && apt-get install -y iputils-ping**

**ping 172.18.0.3**







**NOTE: In default bridge network only, IP’s will work to connect another container not a container names.**

**Dis advantage of bridge n/w: If container restarted, we are not guarantee to get same IP**

**IQ] what is the difference between default bridge network and custom bridge network?**

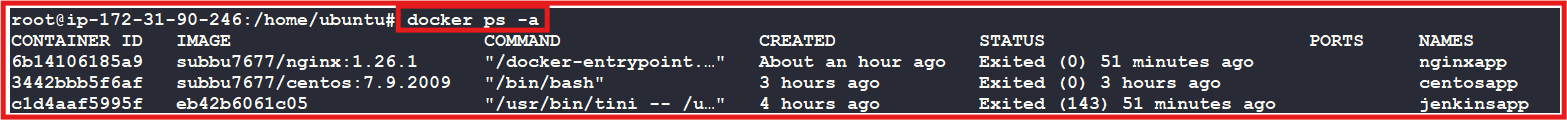
**Ans: In case of default bridge network the communication can happen through only IP address. Where as in custom bridge network the communication can happen through the IP address/ container name.**

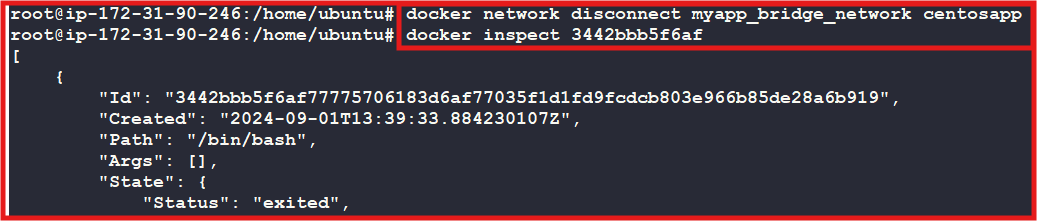
How to disconnect the container from network?

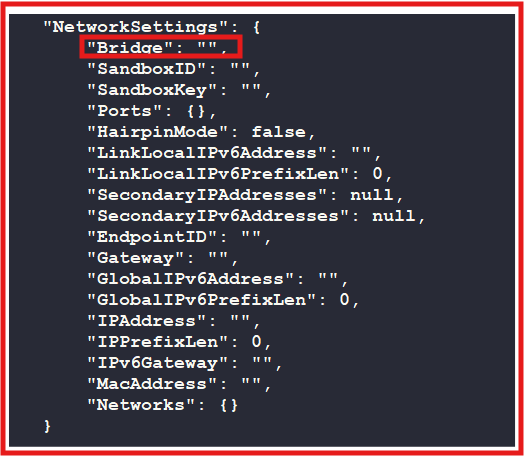
docker network disconnect <network-name> <container-name>

docker network disconnect myapp\_bridge\_network centosapp

docker inspect centosapp or docker inspect 3442bbb5f6af



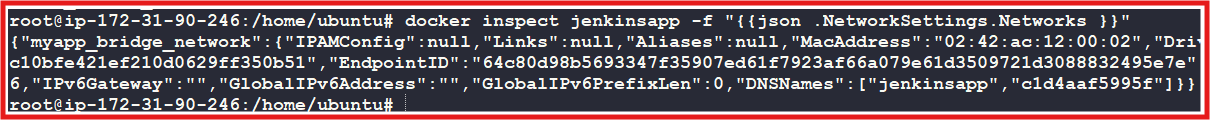




**To see what network(s) your container is on, assuming your container is called c1:**

**docker inspect c1 -f "{{json .NetworkSettings.Networks }}"**

**Ex: docker inspect jenkinsapp -f "{{json .NetworkSettings.Networks }}"**

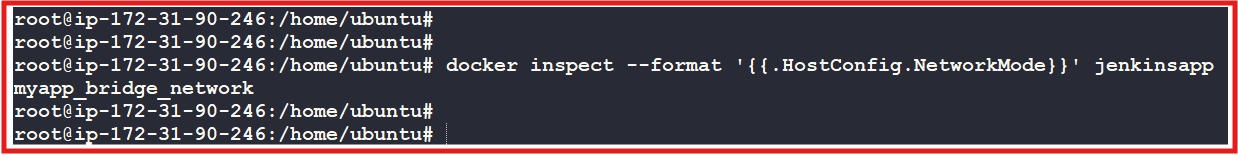
****

**To get the list of networks that a container is connected to:**

**List Networks for a Specific Container:**

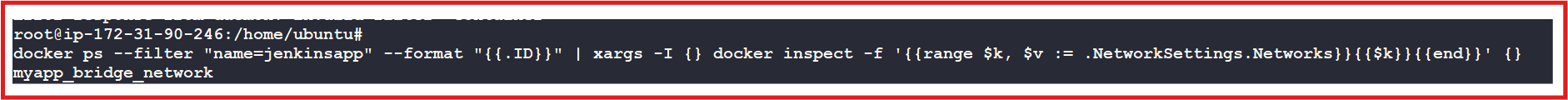
**Ex-1:**

**docker inspect --format '{{.HostConfig.NetworkMode}}' jenkinsapp**

****

**Ex-2:**

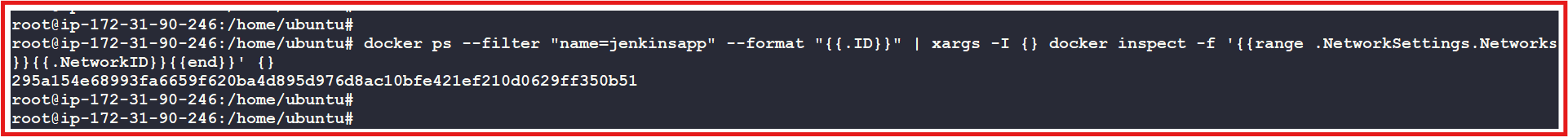
**docker ps --filter "name=jenkinsapp" --format "{{.ID}}" | xargs -I {} docker inspect -f '{{range $k, $v := .NetworkSettings.Networks}}{{$k}}{{end}}' {}**

****

**Convert Network IDs to Network Names:**

**Ex:**

**docker ps --filter "name=jenkinsapp" --format "{{.ID}}" | xargs -I {} docker inspect -f '{{range .NetworkSettings.Networks}}{{.NetworkID}}{{end}}' {}**

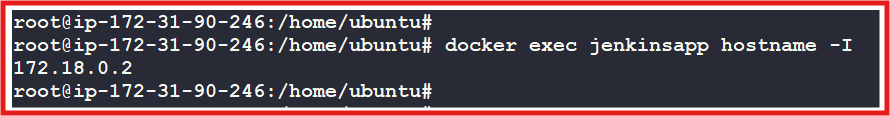
****

**Get IP Address of container network:**

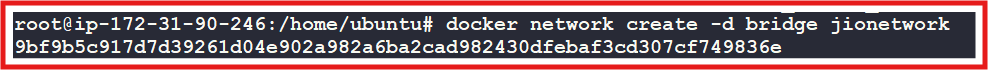
**You can use the docker exec command to run commands inside the container and retrieve network information such as the IP address.**

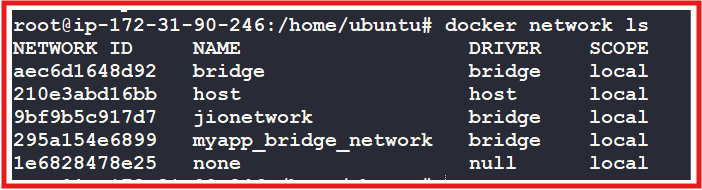
**Syntax: docker exec <container\_name\_or\_id> hostname -I**

**Ex: docker exec jenkinsapp hostname -I**

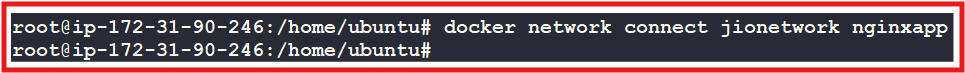
****

**Create another new custom bridge network**

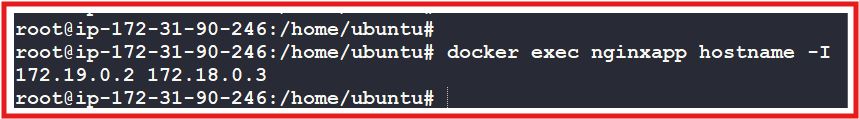
****

****

**Now I have added jionetwork to my nginxapp container which is already hold another network too.**

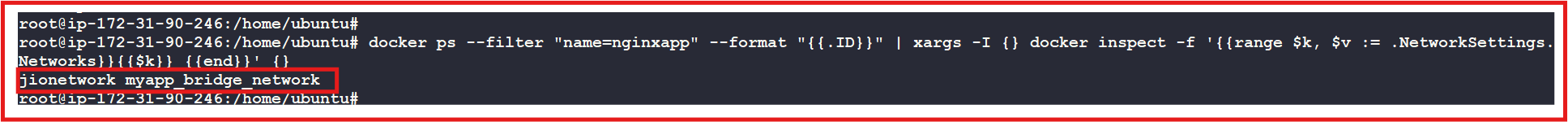
****

**Now we can observe that the two Ip addresses of nginxapp container:**

****

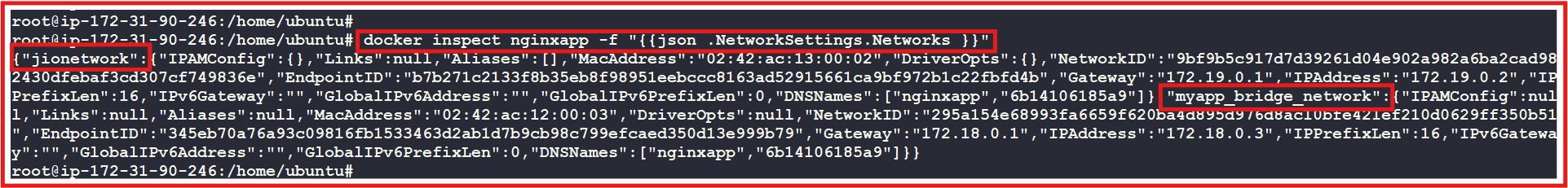
**Now we can observe that the multiple network names of nginxapp container:**

**docker ps --filter "name=nginxapp" --format "{{.ID}}" | xargs -I {} docker inspect -f '{{range $k, $v := .NetworkSettings.Networks}}{{$k}} {{end}}' {}**

****

**To get the complete network details of a container**

**docker inspect nginxapp -f "{{json .NetworkSettings.Networks }}"**

****

**But here there is no clear output formatting , so to get the clear output formatting regarding json information we can use jq tool.**

**jq is a powerful command-line tool designed for processing and manipulating JSON data. It's particularly useful when working with JSON outputs from APIs, configuration files, or any data source that produces JSON.**

**Key Features of jq:**

* **Parsing JSON: jq can parse JSON data and output it in a readable format.**
* **Filtering: It allows you to filter JSON data to extract only the parts you're interested in.**
* **Transforming: You can use jq to transform JSON data, such as converting it into different formats (e.g., CSV, TSV).**
* **Scripting: jq supports complex operations and functions, allowing you to perform calculations, map, reduce, and more.**

**To display the network information of a Docker container in a clear, table-aligned format, you can extract and format specific fields from the JSON output.**

1. **Install jq (If not already installed)**

**sudo apt-get update**

**sudo apt-get install jq**

1. **Use jq to Format the Output:**

**Syntax:**

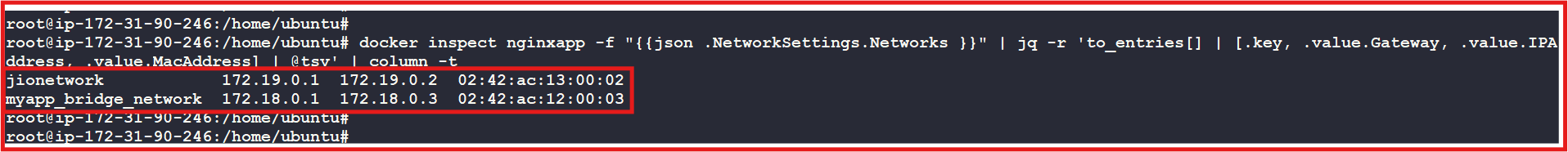
**docker inspect <container-name> -f "{{json .NetworkSettings.Networks }}" | jq -r 'to\_entries[] | [.key, .value.Gateway, .value.IPAddress, .value.MacAddress] | @tsv' | column -t**

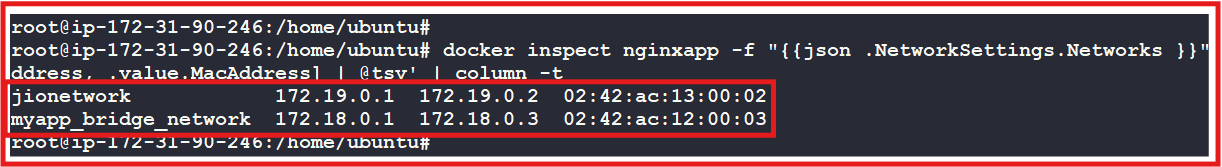
**Explanation:**

* **to\_entries[]: Converts the JSON object into an array of key-value pairs.**
* **[.key, .value.Gateway, .value.IPAddress, .value.MacAddress]: Selects the network name, gateway, IP address, and MAC address.**
* **@tsv: Formats the selected fields as tab-separated values.**
* **column -t: Aligns the output in a table format.**

**Ex:**

**docker inspect nginxapp -f "{{json .NetworkSettings.Networks }}" | jq -r 'to\_entries[] | [.key, .value.Gateway, .value.IPAddress, .value.MacAddress] | @tsv' | column -t**

****

****

**Host Network:**

**What is docker host network?**

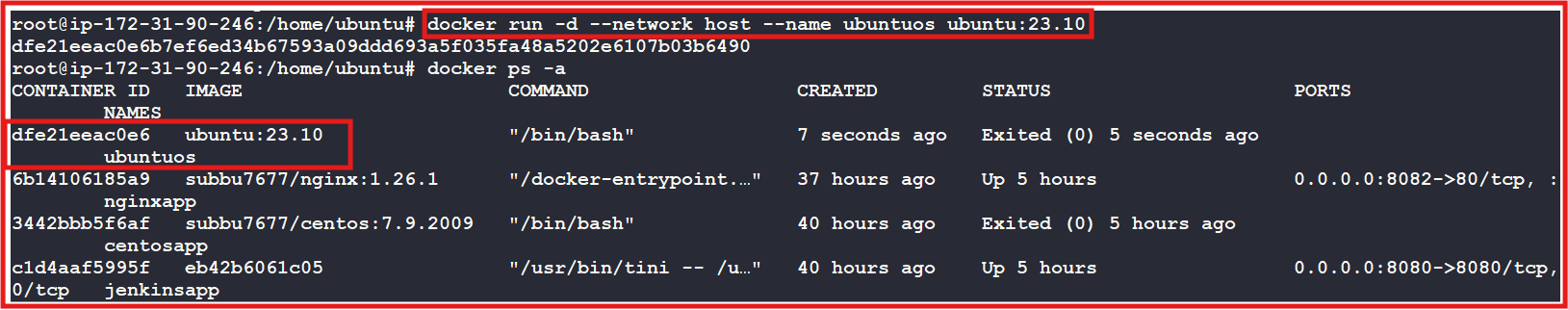
**----------------------------------------**

**If we create container in host network, Container will not have an IP address.**

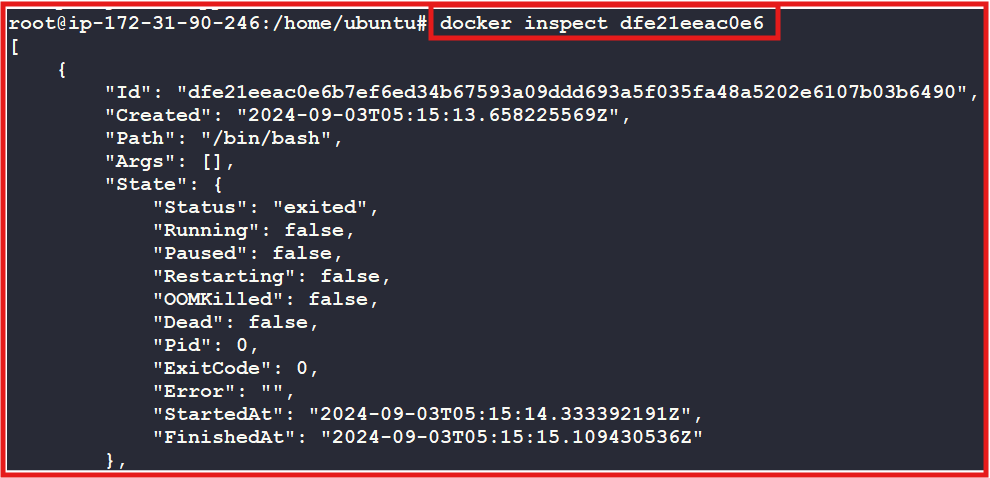
**NOTE: We cannot create more than one container with same container port in the host network.**

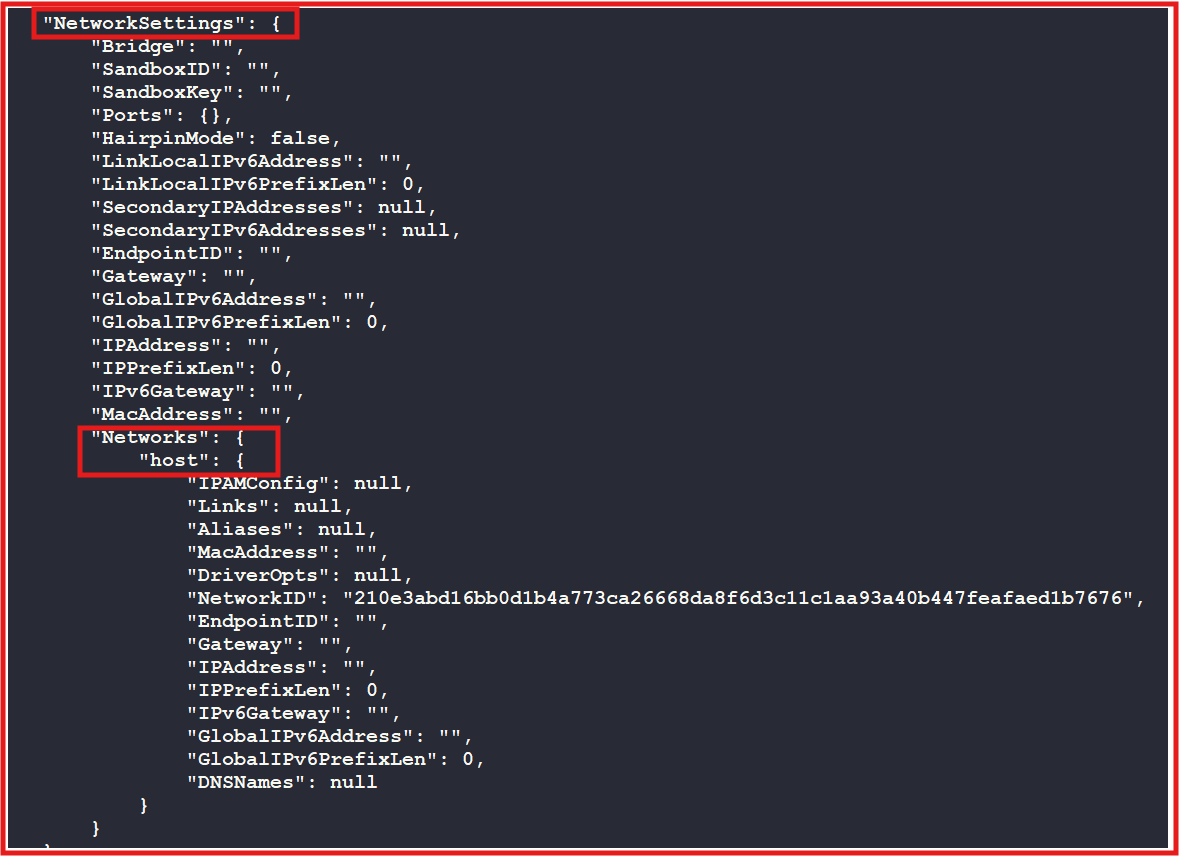
**NOTE: you cannot create custom host network like bridge**

**EX: docker run -d --network host --name ubuntuos ubuntu:23.10**

****

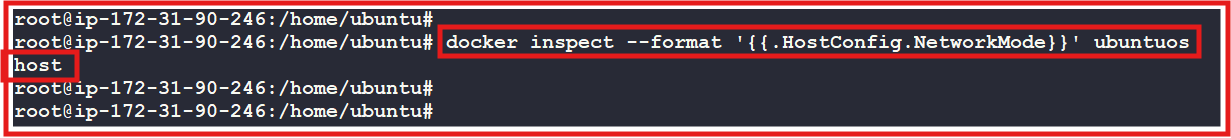
**docker inspect dfe21eeac0e6**

****

****

**Checking if a Container is Using the Host Network:**

**docker inspect --format '{{.HostConfig.NetworkMode}}' ubuntuos**

****

**What is none network?**

**---------------------------------------------------------------------------------------------------------**

**--> If you create containers in none network, Container will not have an IP address.**

**--> we can not access these containers from outside or from any containers or internally.**

**================================================================**

**NOTE: The scope of the bridge/host/none with the docker server only, To overcome this problem we are using "overlay" network [multi host networks]**

**IQ] what is overlay network?**

**Establishing connection between containers in different docker servers.**