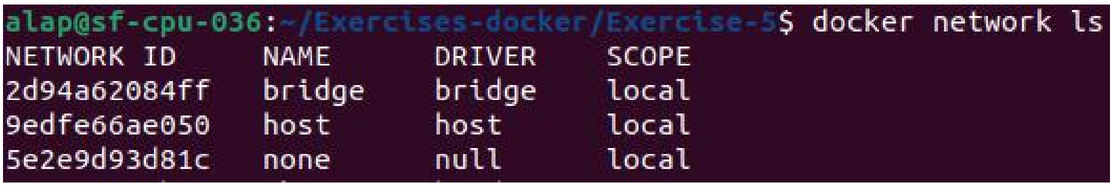
Exercise 6: Networking

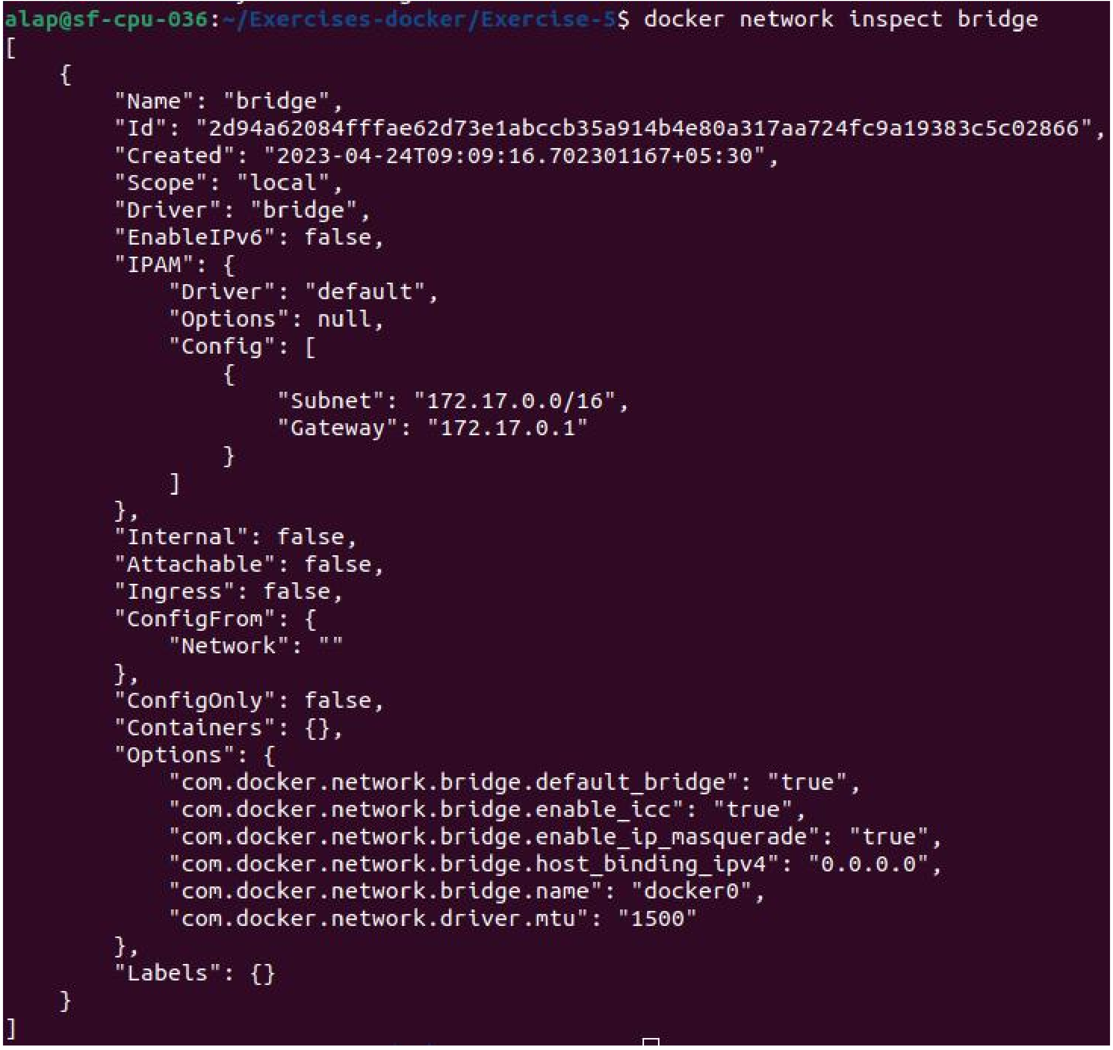
Listing networks

- We can list the networks using docker network ls command.
- There are 3 pre-defined networks inside docker networks.



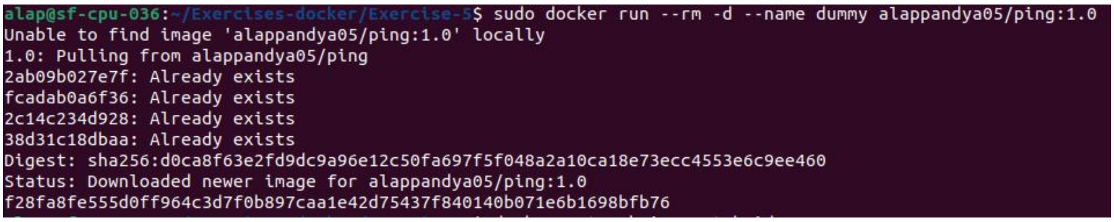
```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
2d94a62084ff   bridge    bridge    local
9edfe66ae050   host      host      local
5e2e9d93d81c   none      null      local
```

- All new containers, if given no other configuration, will be automatically added the bridge network. This network acts as a pass through to your host's ethernet, so your Docker containers can access the internet.
- We can inspect the bridge network by running docker network inspect bridge:

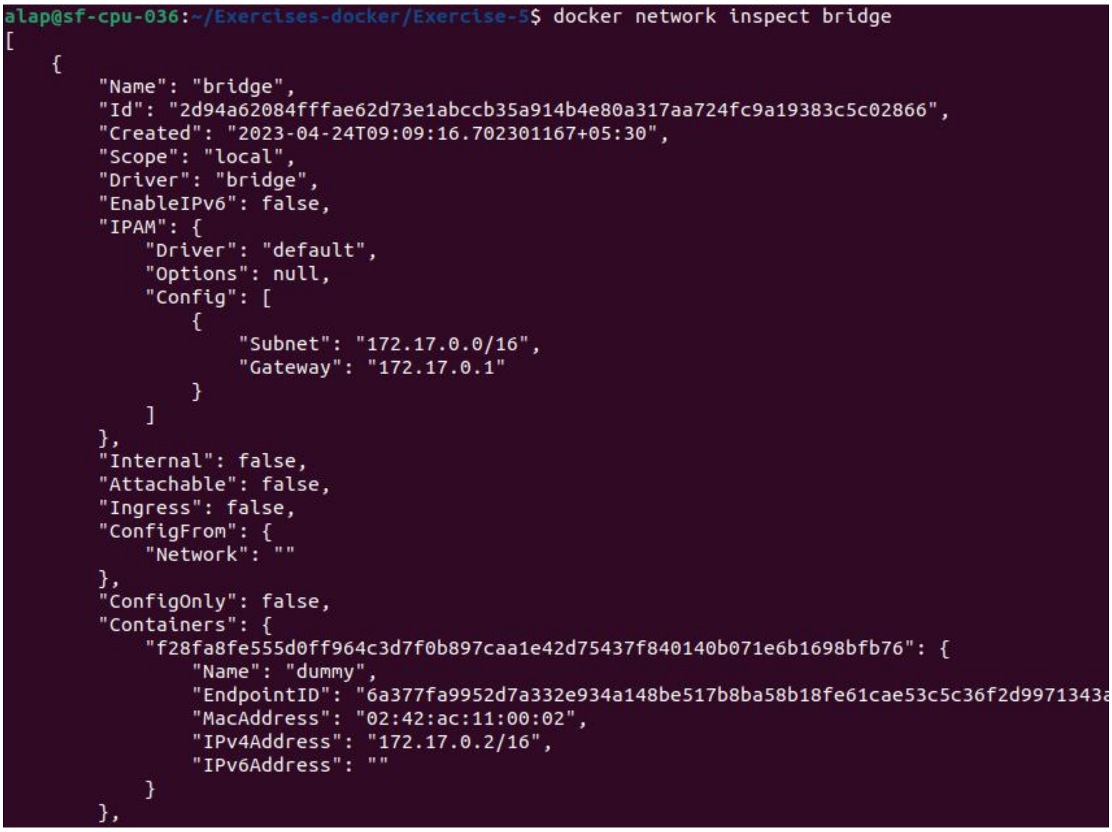


```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "2d94a62084fffae62d73e1abccb35a914b4e80a317aa724fc9a19383c5c02866",
        "Created": "2023-04-24T09:09:16.702301167+05:30",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

- Here, containers section is empty, so we can run a container and lets see the change in this bridge network’s container’s section.

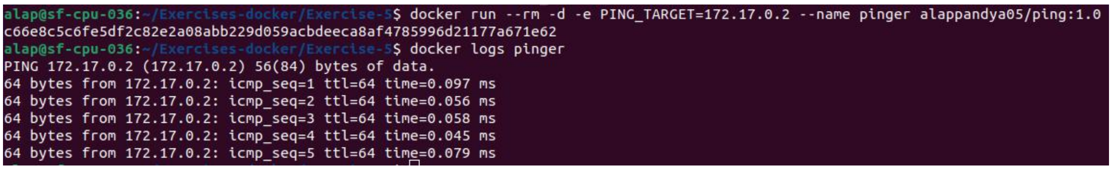


```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ sudo docker run --rm -d --name dummy alappandya05/ping:1.0
Unable to find image 'alappandya05/ping:1.0' locally
1.0: Pulling from alappandya05/ping
2ab09b027e7f: Already exists
fcadab0a6f36: Already exists
2c14c234d928: Already exists
38d31c18dbaa: Already exists
Digest: sha256:d0ca8f63e2fd9dc9a96e12c50fa697f5f048a2a10ca18e73ecc4553e6c9ee460
Status: Downloaded newer image for alappandya05/ping:1.0
f28fa8fe555d0ff964c3d7f0b897caa1e42d75437f840140b071e6b1698bfb76
```



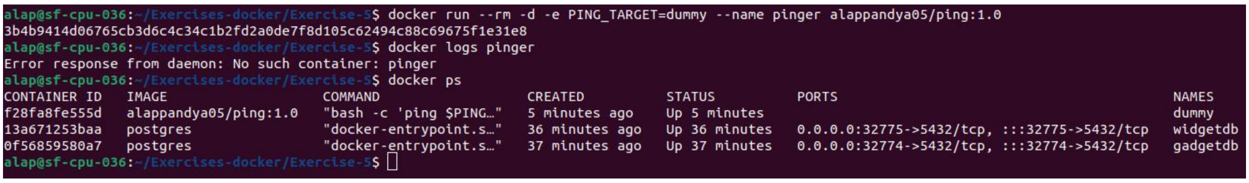
```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "2d94a62084fffae62d73e1abccb35a914b4e80a317aa724fc9a19383c5c02866",
        "Created": "2023-04-24T09:09:16.702301167+05:30",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "f28fa8fe555d0ff964c3d7f0b897caa1e42d75437f840140b071e6b1698bfb76": {
                "Name": "dummy",
                "EndpointID": "6a377fa9952d7a332e934a148be517b8ba58b18fe61cae53c5c36f2d9971343a
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
```

- We can see the container was added to the default network. Now let's add another ping container, and set it to ping our first.



```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker run --rm -d -e PING_TARGET=172.17.0.2 --name pinger alappandya05/ping:1.0
c66e8c5c6fe5df2c82e2a08abb229d059acbdeeca8af4785996d21177a671e62
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker logs pinger
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.097 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.056 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.045 ms
64 bytes from 172.17.0.2: icmp_seq=5 ttl=64 time=0.079 ms
```

- Inspecting the logs for pinger we can see it was able to successfully ping the other container in the network.
- While IP address does work, it's very cumbersome and prone to error if addresses change.
- It would be better to use a hostname, specifically the container name dummy, to always resolve to the correct container.
- Running ping with the dummy as the target:



```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker run --rm -d -e PING_TARGET=dummy --name pinger alappandya05/ping:1.0
3b4b9414d06765cb3d6c4c34c1b2fd2a0de7f8d105c62494c88c69675f1e31e8
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker logs pinger
Error response from daemon: No such container: pinger
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          PORTS                                         NAMES
f28fa8fe555d   alappandya05/ping:1.0  "bash -c 'ping $PING…"   5 minutes ago    Up 5 minutes                                                  dummy
13a671253baa   postgres               "docker-entrypoint.s…"   36 minutes ago   Up 36 minutes   0.0.0.0:32775->5432/tcp, :::32775->5432/tcp   widgetdb
0f56859580a7   postgres               "docker-entrypoint.s…"   37 minutes ago   Up 37 minutes   0.0.0.0:32774->5432/tcp, :::32774->5432/tcp   gadgetdb
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ 
```

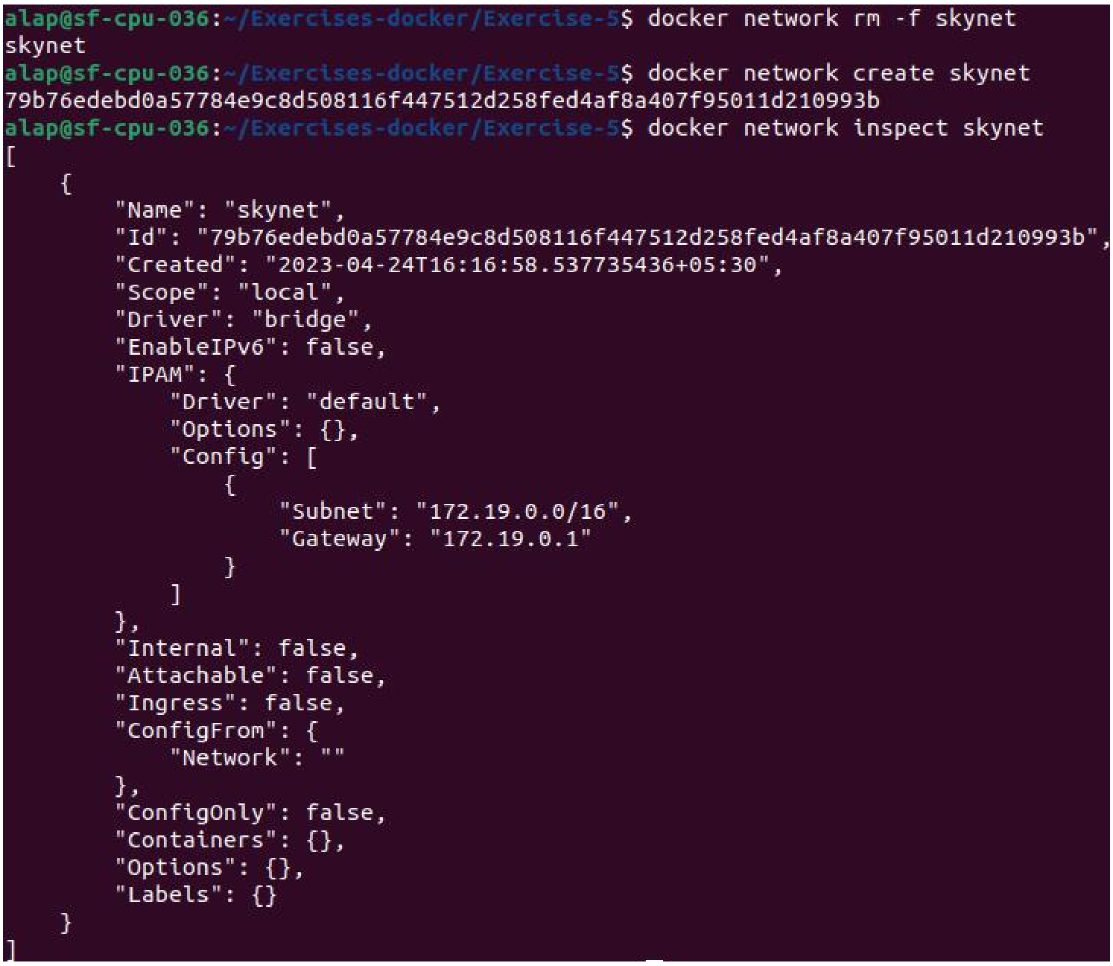
- ...results in failure. The host name couldn't be resolved, thus causing the command to error and the container exit and autoremove.

- The default bridge network will not automatically allow you to network containers by container name. We can, however, easily accomplish host resolution using a custom network.

- Stop and remove the dummy container by running docker stop dummy.

Managing custom networks

- To create a new network, use the docker network create command and provide it a network name.

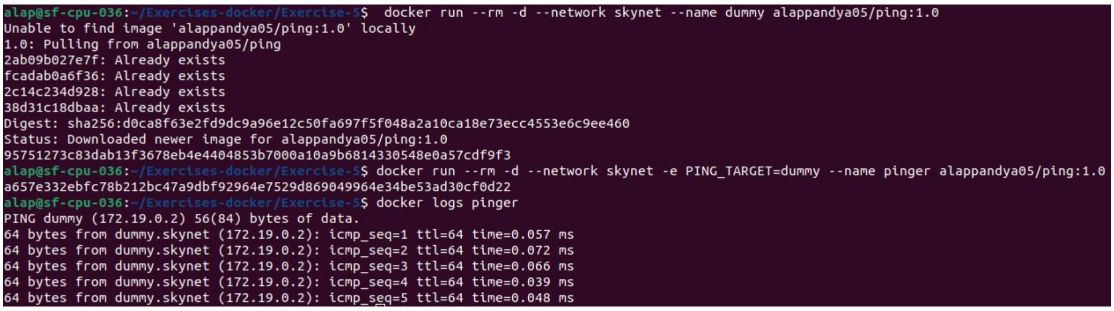


```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker network rm -f skynet
skynet
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker network create skynet
79b76edebd0a57784e9c8d508116f447512d258fed4af8a407f95011d210993b
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker network inspect skynet
[
    {
        "Name": "skynet",
        "Id": "79b76edebd0a57784e9c8d508116f447512d258fed4af8a407f95011d210993b",
        "Created": "2023-04-24T16:16:58.537735436+05:30",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

- To remove networks, run docker network rm and provide it the network name.

Adding containers to a network

- Let's rerun the ping container, this time assigning it a network.
- Then the pinger, targeting the dummy ping container:

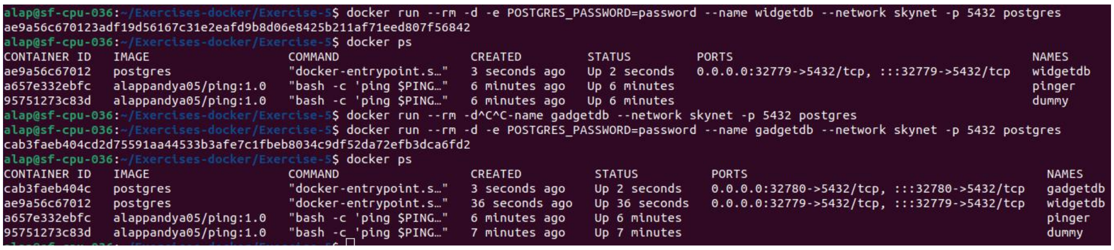


```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$  docker run --rm -d --network skynet --name dummy alappandya05/ping:1.0
Unable to find image 'alappandya05/ping:1.0' locally
1.0: Pulling from alappandya05/ping
2ab09b027e7f: Already exists
fcadab0a6f36: Already exists
2c14c234d928: Already exists
38d31c18dbaa: Already exists
Digest: sha256:d0ca8f63e2fd9dc9a96e12c50fa697f5f048a2a10ca18e73ecc4553e6c9ee460
Status: Downloaded newer image for alappandya05/ping:1.0
95751273c83dab13f3678eb4e4404853b7000a10a9b6814330548e0a57cdf9f3
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker run --rm -d --network skynet -e PING_TARGET=dummy --name pinger alappandya05/ping:1.0
a657e332ebfc78b212bc47a9dbf92964e7529d869049964e34be53ad30cf0d22
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker logs pinger
PING dummy (172.19.0.2) 56(84) bytes of data.
64 bytes from dummy.skynet (172.19.0.2): icmp_seq=1 ttl=64 time=0.057 ms
64 bytes from dummy.skynet (172.19.0.2): icmp_seq=2 ttl=64 time=0.072 ms
64 bytes from dummy.skynet (172.19.0.2): icmp_seq=3 ttl=64 time=0.066 ms
64 bytes from dummy.skynet (172.19.0.2): icmp_seq=4 ttl=64 time=0.039 ms
64 bytes from dummy.skynet (172.19.0.2): icmp_seq=5 ttl=64 time=0.048 ms
```

- Notice this time the host name resolves successfully. This is Docker's Embedded DNS in action.
- It's most useful when orchestrating multiple containers in a single application, such as a web server, database, and cache.
- Instead of using IP addresses, you can define each of the respective connection strings using container names to leverage DNS resolution.
- Stop and remove the containers by running docker stop pinger and docker stop dummy.

Connecting between containers in a network

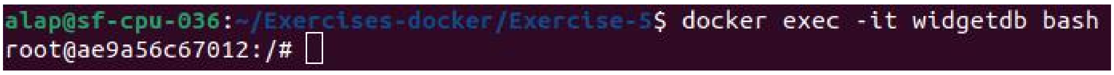
- We can resolve host names and ping, but this isn't the same as connecting with TCP/UDP between containers.
- Let's setup two postgres databases to connect to one another: a widget database, and gadget database.
- Start each of the databases and add them to the network:



```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker run --rm -d -e POSTGRES_PASSWORD=password --name widgetdb --network skynet -p 5432 postgres
ae9a56c670123adf19d56167c31e2eafd9b8d06e8425b211af71eed807f56842
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED         STATUS         PORTS                                         NAMES
ae9a56c67012   postgres                  "docker-entrypoint.s…"   3 seconds ago   Up 2 seconds   0.0.0.0:32779->5432/tcp, :::32779->5432/tcp   widgetdb
a657e332ebfc   alappandya05/ping:1.0     "bash -c 'ping $PING…"   6 minutes ago   Up 6 minutes                                                 pinger
95751273c83d   alappandya05/ping:1.0     "bash -c 'ping $PING…"   6 minutes ago   Up 6 minutes                                                 dummy
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker run --rm -d^C^C-name gadgetdb --network skynet -p 5432 postgres
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker run --rm -d -e POSTGRES_PASSWORD=password --name gadgetdb --network skynet -p 5432 postgres
cab3faeb404cd2d75591aa44533b3afe7c1fbeb8034c9df52da72efb3dca6fd2
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED          STATUS          PORTS                                         NAMES
cab3faeb404c   postgres                  "docker-entrypoint.s…"   3 seconds ago    Up 2 seconds    0.0.0.0:32780->5432/tcp, :::32780->5432/tcp   gadgetdb
ae9a56c67012   postgres                  "docker-entrypoint.s…"   36 seconds ago   Up 36 seconds   0.0.0.0:32779->5432/tcp, :::32779->5432/tcp   widgetdb
a657e332ebfc   alappandya05/ping:1.0     "bash -c 'ping $PING…"   6 minutes ago    Up 6 minutes                                                  pinger
95751273c83d   alappandya05/ping:1.0     "bash -c 'ping $PING…"   7 minutes ago    Up 7 minutes                                                  dummy
```

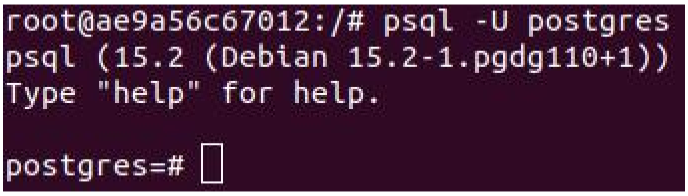
- By default, port 5432 is blocked and inaccessible. However, by adding -p 5432, we are permitting other containers to access it through port 5432, the default Postgres port.

- Now that they're running, start a shell session in the widgetdb using docker exec:



```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker exec -it widgetdb bash
root@ae9a56c67012:/# 
```

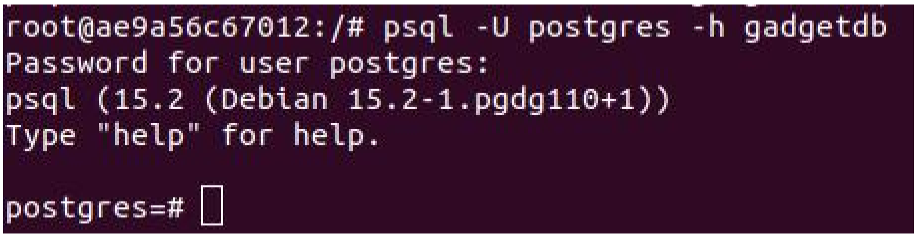
You can then connect to the local database using psql. (End the psql session by entering \q.)



```
root@ae9a56c67012:/# psql -U postgres
psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

postgres=# 
```

Or to the gadget database by referring to it by name:



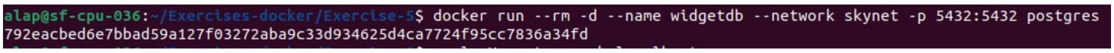
```
root@ae9a56c67012:/# psql -U postgres -h gadgetdb
Password for user postgres:
psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

postgres=# 
```

Type exit to end the session, then docker stop widgetdb gadgetdb to stop and remove the containers.
Binding ports to the host

Sometimes its useful to access an application running in a Docker container directly, as if it were running on your host machine.

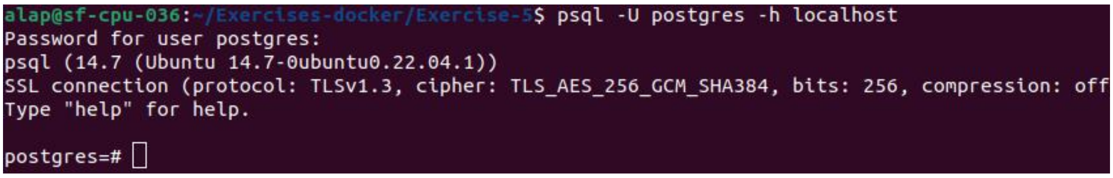
To this end, you can bind ports from a container to a port on your host machine. To do this, the altered command from our previous Postgres example would look like:



```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ docker run --rm -d --name widgetdb --network skynet -p 5432:5432 postgres
792eacbed6e7bbad59a127f03272aba9c33d934625d4ca7724f95cc7836a34fd
```

The -p flag given <host port>:<container port> does this mapping, making the server available as localhost:5432:

You can then run psql (if the utility is installed) on your host machine to access the Postgres database



```
alap@sf-cpu-036:~/Exercises-docker/Exercise-5$ psql -U postgres -h localhost
Password for user postgres:
psql (14.7 (Ubuntu 14.7-0ubuntu0.22.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off
Type "help" for help.

postgres=# 
```

It's important to keep in mind that you can only bind one application to a host port at a time. If you try to start any applications on your host machine, or other Docker containers that want to bind to a port already in use, they will fail to do so.

Type docker stop widgetdb to stop and remove the container.