Docker Ağ yapıları ve kullanımı

# **#1 - Ağ temelleri**

## Adım 1: ‘docker network’ komutu

docker network komutu konteyner ağlarını yönetmek ve ayarlamak için kullanabileceğimiz komuttur. Sağdaki(üsste) ilk ekrandan docker network komutunu çalıştırın.

**docker network**

**Usage: docker network COMMAND**

**Manage networks**

**Options:**

**--help Print usage**

**Commands:**

**connect Connect a container to a network**

**create Create a network**

**disconnect Disconnect a container from a network**

**inspect Display detailed information on one or more networks**

**ls List networks**

**prune Remove all unused networks**

**rm Remove one or more networks**

**Run 'docker network COMMAND --help' for more information on a command.**

Komut çıktısı ile docker network ile kullanabileceğiniz alt komutların listesi gelecektir. Çıktıdan görebileceğiniz gibi docker network komutu sizin ağlar yaratmanızı, ağları listelemenizi, ağları incelemenizi ve ağları silmenizi sağlar. Bu komut ayrıca konteynerlerinizi ağlara bağlamanızı ve ayırmanızı sağlar.

## Adım 2: Ağları listelemek

Bulunduğunuz Docker sunucusu üzerindeki var olan konteyner ağlarını görebilmek için docker network ls komutunu çalıştırmalısınız.

**docker network ls**

Yukarıdaki çıktı, Docker’ın standart kurulumunun parçası olarak oluşturulan konteyner ağlarını gösteriyor.

Oluşturduğunuz yeni ağlar’da docker network ls çıktısı içinde yer alacaktır.

Gördüğünüz gibi her bir network için bir (kimlik) ID ve (isim) NAME vardır. Her network aynı zamanda bir sürücü(driver) ile eşleştirilmiştir. “bridge” (köprü) ve “host” olan ağların NAME(isim) ve DRIVER(sürücü) aynı olduğunu görebilirsiniz.

## Adım 3: Bir ağı incelemek (inspect)

docker network inspect komutunu kullanarak docker ağ yapısını derinlemesine inceleyebilirsiniz. Bu incelemeniz sırasında göreceğiniz detaylar; name, ID, driver, IPAM driver, subnet info, connected containers, ve daha bir çok ayrıntıdır.

docker network inspect <network> komutunu kullanırsanız sadece belirttiğiniz konteyner ağı hakkında bilgi edinebilirsiniz. Docker networklerinin sadece Docker çalıştıran sunucunuz üzerinde olduğunu unutmayın lütfen. Aşağıdaki komut bridge ağı ile ilgili ayrıntıları gösterecektir.

**docker network inspect bridge**

**NOT:** docker network inspect komutunun kullanım şekli docker network inspect <network> şeklinde olmalıdır, burada <network> ağ ismi veya ağ ID olabilir. Yukarıda gördüğünüz örnek içersinde “bridge” isimli ağın ayrıntılarını gösteriyoruz. Buradaki “bridge” kullanımı sürücü(driver) için değildir!

## Adım 4: Ağ sürücü eklentilerinin listesi (List network driver plugins)

docker info komutu, kurulu Docker servisi ile ilgili oldukça fazla ve ayrıntılı bilgi sağlar.

docker info komutunu çalıştıralım ve “network pugins” kısmına bir bakalım.

**docker info**

yukarıdaki çıktı **bridge**, **host**,**macvlan**, **null**, ve **overlay** sürücülerini gösteriyor.

# **#2 - Bridge(köprü) Ağlar**

## Adım 1: Temeller

Her sıfır Docker kurulumu **bridge** isminde bir ağı otomatik olarak kurar. Bunu docker network ls yaparak görebiliriz.

**docker network ls**

Yukarıda görebileceğimiz gibi **bridge** ağı bridge sürücüsü(driver) ile ilişkilendirilmiştir. Ağın ve sürücünün birbiri ile bağlantılı olduğunu bilmemiz çok önemlidir fakat isim ve sürücü aynı şey demek değildir.Burada size göstereceğimiz örnek içersinde ağ ve sürücü aynı isimdedirler fakat lütfen ikisini birbiri ile karıştımayın!

Yukarıda gördüğünüz örnekte **bridge** ağı “SCOPE” olarak yerel(local) tanımına sahiptir. Yani buradaki ağ sadece Docker çalışan sunucu üzerinde varlığını sürdürür, erişim alanı sunucu üzerindeki ağ dır. Bu durum bridge(köprü) sürücüsü kullanan tüm ağlar için geçerlidir - bridge sürücüsü tek-suncu’lu bir ağ tanımı yapar. bridge olan bir sürücü varlığını sadece bulunduğu bilgisayarda gösterebilir.

Bridge üzerinde tanımlayacağınız ip adresleri ve işlemler sadece çalışan bilgisayar için varlığını sürdürecektir. Yerel Ağ(Local network) üzerindeki diğer bilgisayarların bridge üzerindeki ağ ile iletişimi yoktur.

bridge sürücüsü ile yaratılmış tüm ağlar Linux bridge (sanal switch - virtual switch) ile desteklenmektedir. Linux bridge sanallaştırma yöntemlerinde(XEN,KVM) ve LXC,Vagrant gibi konteyner sanallaştırması yöntemlerinde oldukça sık şekilde kullanılmaktadır. Linux bir sunucu üzerinde oluşturacağınız sanallaştırma ortamında mutlaka dışarıdaki ağlar ile olan bağlantıyı linux bridge ile yapmanız gerekir. (OVS)Open Virtual Switch bile kendisini alt katmanda Linux bridge ile eşleştirir.

brctl uygulamasını kurun ve kullanın, bu sayede Docker sunucusu üzerinde çalışan Bridge ağlar ile ilgili bilgi edinebilirsiniz.. bu programı kurmak için çalıştırmanız gereken komut sudo apt-get install bridge-utils.

**apt update**

**apt install -y bridge-utils**

Sonrasında brctl show komutunu çalıştırarak Docker sunucusu üzerinde çalıştan birdge(köprü ağ) ağlarını listeleyebilirsiniz.

**brctl show**

Bridge networkler çok kullanışlıdırlar ve aynı zamanda çok basit yapılardır, network sorunları ile uğraştığım on yıllar içinde network bridge kaynaklı hiç bir problem görmedim.

Yukarıda yazdığınız komutun çıktısı size **docker0** isimli bir bridge verecektir. Bu birdge(köprü ağ) **bridge** ismini taşıyan ağ için docker tarafından otomatik olarak kurulmuştur. Şuanda herhangi bir sanal ağ bağdaştırıcı(interface) bağlı olmadığını görebilirsiniz. en sağdaki “interfaces” kısmı size bağlı bulunan VIF (Virtual Interface) veya sanal ağ bağdaştırıcı bilgisini verir.

Bu duruma şöyle örnek verebiliriz, docker konteynerlerin ağ bağdaştırıcıları ve kabloları olsun, buradaki interfaces yazan kolonda eğer kablonuz bağlı ise, konteynerinizin ağ bilgisini görürsünüz.

ip a komutunu çalıştırarak **docker0** bridge(köprü ağ) ile ayrıntılı bilgi edinebilirsiniz.

**ip a**

## Adım 2: Bir konteyner bağlamak

**bridge** ağı, yeni konteynerler için varsayılan ağ dır. Bunun anlamı, farklı bir şekilde tanım yapmadığınız sürece, tüm konteynerler **bridge** olan ağ üzerine erişim sağlayacaklardır. Yani **bridge** olan ağ için konfigürasyon içersinde herhangi bir tanıma ihtiyaç yoktur.

docker run -dt ubuntu sleep infinity komutunu çalıştıralım ve yeni bir konteyner oluşturalım.

**docker run -dt ubuntu sleep infinity**

Bu komut ubuntu:latest imajını temel alan ve ubuntu dağıtımının son sürümünü içeren bir konteyner çalıştıracaktır ve sleep komutu sayesinde başlatılmış konteyner arka planda çalışmaya devam edecektir. Bunu doğrulamak için docker ps komutu ile konteynerin halen çalıştığını gözlemleyebilirsiniz.

**docker ps**

docker run komutunda herhangi bir ağ tanımı yapılmadığı için konteyner **bridge** ağını herhangi bir konfigürasyona gerek olmadan ekler.

brctl show komutunu şimdi tekrar çalıştıralım.

**brctl show**

**docker0** bridge üzerine bağlanmış bir (ağ bağdaştırıcısı) -interface olduğunu görebilirsiniz. Bu ağ bağdaştırıcısı veya VID doğrudan **docker0** bridge(köprü ağı) üzerine bağlanacaktır.

Şimdi **bridge** ağını tekrar ‘ispect’ ederek inceleyebilirsiniz, bunun içindocker network inspect bridge komutunu kullanmanız gerekir, bu sayede yeni yaratılmış olan konteyner’in **bridge** üzerinde bağlı olduğunu görebilirsiniz.

**docker network inspect bridge**

## Adım 3: Ağ bağlantısını test edelim

önceki docker network inspect komutu çıktısı bize yeni konteynerin IP address bilgisini verir. Önceki örnek için bu adres “172.17.0.2” ‘tir, muhtemelen sizin gördüğünüz adres daha farlıdır.

Docker sunucunuz üzerinden ping -c5 <IPv4 Adresi> komutunu yazarak (<IPv4 adresi> sizin ortamınızdan olmalı)konteyner networkünün erişilebilir olduğunu görebilirsiniz.

**ping -c5 172.17.0.2**

**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=0.055 ms**

**64 bytes from 172.17.0.2: icmp\_seq=2 ttl=64 time=0.031 ms**

**64 bytes from 172.17.0.2: icmp\_seq=3 ttl=64 time=0.034 ms**

**64 bytes from 172.17.0.2: icmp\_seq=4 ttl=64 time=0.041 ms**

**64 bytes from 172.17.0.2: icmp\_seq=5 ttl=64 time=0.047 ms**

**--- 172.17.0.2 ping statistics ---**

**5 packets transmitted, 5 received, 0% packet loss, time 4075ms**

**rtt min/avg/max/mdev = 0.031/0.041/0.055/0.011 ms**

Yukarıda gördüğümüz çıktı bize **bridge** ağı üzerinden konteyner üzerine erişebildiğimizi gösterir. Fakat aynı zamanda konteynerin dış dünya ile bağlantısı olduğu anlamına da gelir. isterseniz konteyner üzerine bağlanalım, sonrasında ping programını kuralım, ve www.github.com pingleyelim.

Öncelikle konteyner ID ye ihtiyacımız olacak, docker ps komutu ile bunu öğrenebiliriz.

**docker ps**

Sonrasında, konteyner içerisine bağlanalım ve bir shell(kabuk) çalıştıralım. bunu yapmak için docker exec -it <CONTAINER ID> /bin/bash komutunu yazmalıyız..

**docker exec -it yourcontainerid /bin/bash**

**root@846af8479944:/#**

Daha sonra, ping programını kurmamız gerekir. Yani, apt-get update && apt-get install -y iputils-ping komutunu çalıştırabiliriz.

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

www.github.com adresini pingleyebilmek için ping -c5 www.github.com çalıştıralım.

**ping -c5** [**www.github.com**](http://www.github.com)

Son olarak bağlantımızı kesip konteyner üzerinden çıkalım. Bunun için exit yazmamız yeterli.

**Exit**

Bu testi yaptığımız konteyneri durdurlmalıyız, bunun için docker stop <CONTAINER ID> yazmamız yeterli.

**docker stop [SizinKonteynerID'niz]**

Burada yaptığımız deneme sonrası gördükki konteyner üzerinden internet erişimimiz var ve dış dünyaya bağlanabiliyoruz.

## Adım 4: Dış bağlantı için NAT yapılandırması

Bu örnekte yeni bir **NGINX** konteyner çalıştıracağız ve Docker sunucusu üzerindeki 8080 portunu konteyner üzerindeki 80 portu ile eşleştireceğiz(map edeceğiz). Bunun anlamı Docker sunucusu üzerindeki 8080 portu üzerine gelen istekler konteyner içersindeki 80 portu üzerine yönlenecekler.

**NOT:** Standart NGINX imajı ile bir konteyner çalıştırırsanız, varsayılan olarak 80 portu üzerine bu eşleştirmeyi yapacaktır.

Standart NGINX imajını temel alan bir konteyner çalıştıralım, ve 8080 ile 80 portu eşleştirmemiz olsun docker run --name web1 -d -p 8080:80 nginx.

**docker run --name web1 -d -p 8080:80 nginx**

docker ps komutunu çalıştırarak çalışan konteynerin hangi portu bağladığını ve çalışıp çalışmadığını kontrol edelim.

**docker ps**

Göreceğiniz gibi en üstte **web1** isminde bir NGINX imajını temel almış konteyner çalıştığını görüyoruz. Ayrıca portların nasıl bağlandığı ile ilgili - 0.0.0.0:8080->80/tcp bilgisini görüyoruz burada söylenmek istenen Docker sunucusu üzerindeki 8080 portunun **web1** konteyneri içindeki 80 portuna bağlı olduğu bilgisidir. Bu port bağlantısı sayesinde çalışmış olan konteynera dışarıdaki ağlar üzerinden bağlantı sağlanabilir. Yani yaptığımız şey basit olarak bir NAT operasyonuydu, bu NAT operasyonunu sunucumuzun üzerindeki bir konteynerin bir portuna verdik.

Şuanda konteynerin çalıştığını ve docker host üzerindeki bir port üzerinden yönlenen bir port ile çalışan bi NGINX sunucusu olduğunu görebilirsiniz.

Sağ tarafta gördüğünüz iki bilgisayar simülasyonu içersinde Üstteki terminal üzerinde çalışan sunucu ve alttaki terminal üzerindeki sunucu farklı bilgisayarları temsil etmektedir. Docker çalışan sunucu üzerinden curl 127.0.0.1:8080 komutu ile erişebileceğiniz çıktıyı “127.0.0.1” ip adresi yerine diğer sunucunun ip adresini yazarak ulaşabilirsiniz.

Örnek: lütfen 172.20.0.9 adresini üstteki ip adresi ile değitirin!

**curl 172.20.0.9:8080**

**<!DOCTYPE html>**

**<html>**

**<Snip>**

**<head>**

**<title>Welcome to nginx!</title>**

**<Snip>**

**<p><em>Thank you for using nginx.</em></p>**

**</body>**

**</html>**

curl komutunu farklı bir ip ve port için kullanırsanız aynı çıktı gelmeyecektir.

**NOT:** Buradaki port yönlendirme işlemi aslında port address translation (PAT) yöntemidir.

# **#3 - Overlay(üst katman) Ağlar**

## Adım 1: Temeller

Bu örnek içinde yeni bir Swarm oluşturacaksınız, bir worker node ekleyecek ve işlemin gerçekleştiğini kontrol edeceksiniz. Docker Swarm, birden fazla docker sunucu çalıştırmak istediğinizde ve onların bir docker sunucu gibi çalışmasını istediğinizde kullandığınız bir çözümdür.

Bu komutu üstteki terminalde çalıştıralım docker swarm init --advertise-addr $(hostname -i).

**docker swarm init --advertise-addr $(hostname -i)**

Birinci terminalden aldığınız çıktıda docker swarm join ... ile başlayan kısmı alıp ikinci terminale yapıştıralım.

Şimdi docker node ls komutunu çalıştıralım ve her iki node çalışıyormu kontrol edelim.

**docker node ls**

ID ve HOSTNAME için verilen değerler sizin için farklı olabilir. Burada önemli olan iki bilgisayarın bir araya gelip bir Docker Swarm üzerinde ready ve active statülerine sahip olmasıdır.

## Adım 2: Bir overlay(üst katman) ağ yaratmak

Bir Swarm başlattığımıza göre, Swarm içerisindeki her bilgisayardan erişilebilen ağlar yaratabiliriz. Bunu yapabilmek için **overlay** (üst katman) ağlar yaratabiliriz.

“overnet” isminde bir ağ yaratalım, bunun için docker network create -d overlay overnet komutunu çalıştırmamız yeterli olacaktır.

**docker network create -d overlay overnet**

docker network ls komutunu kullanarak overlay ağımızın başarılı şekilde yaratıldığını görebiliriz.

**docker network ls**

Yukarıdaki çıktıda en alt satırda “overnet” ismindeki ağı görebilirsiniz. DRIVER olarak **Overlay** sürücüsü ile bağlı olduğuna dikkat edin, ayrıca SCOPE olarak tüm **swarm** üzerinde erişimi olduğunu görebiliriz.

**NOT:** Swarm oluşturulduğunda, iki ayrı ağ daha oluşur (ingress and docker\_gwbridge) ve bunlar varsayılan olarak tüm swarm üzerinde tanımlanırlar.

Şimdi ikinci sunucumuza (sağ alttaki Terminal) bağlanalım ve ağ bilgilerini almak için docker network ls komutunu tekrar çalıştıralım.

**docker network ls**

Gördüğümüz gibi “overnet” ağı ikinci sunucu üzerinde mevcut **değil**. Herhangi bir hata yok, bunun çok basit bir sebebi var. Bunun sebebi overlay ağlar sadece ihtiyaç duyulduğunda diğer sunucuda yaratılırlar. Bu durum genellikle bir servisin her iki sunucu üzerindeki konteynerleri etkildiğinde değişir. Birazdan örnek içinde bu durumu göstereceğiz.

docker network inspect <network> komutunu kullanarak “overnet” ağı ile ilgili daha derinlemesine bilgi alalım. Bu komutu birinci terminalden çalıştırmanız gerekiyor çünkü henüz bu ağ ikinci node üzerinde aktif değil.

**docker network inspect overnet**

## Adım 3: Bir servis yaratın

Bir swarm oluşturduk ve üzerine bir overlay ağ yarattık. Şimdi yapmamız gereken bu ağ üzerinde tanımlı olacak bir servis yaratmak.

Aşağıdaki komutu birinci(üstteki) terminal içersinden çalıştırın. Bu sayede myservice isminde bir servisi overnet ağı üzerinde iki görev/replika şeklinde tanımlayabiliriz.

**docker service create --name myservice \**

**--network overnet \**

**--replicas 2 \**

**ubuntu sleep infinity**

Servisin başladığından ve her iki replikanın aktif olduğundan emin olalım, bunun için docker service ls komutunu kullanabiliriz.

**docker service ls**

REPLICAS sütunu içindeki 2/2 yazısı bize her iki görevin başladığını ve çalıştığını gösterecektir.

Replikaların her birinin Swarm üzerindeki iki aynı node üzerinde çalıştığından emin olalım, bunun için docker service ps myservice komutunu kullanabiliriz.

**docker service ps myservice**

ID ve NODE değerleri sizin için farklı olabilir. Önemli olan her bir görev/replika(task/replica) farklı bir node üzerinde çalışıyor.

Şimdi “overnet” ağı üzerindeki çalıştırdığımız replikasyon görevi olduğu için ikinci sunucudan (alltaki terminal) docker network ls komutu bize “overnet” ağını gösterecektir.

**docker network ls**

Aynı şekilde ikinci terminalden(alltaki) docker network inspect overnet komutunu çalıştırdığımızda bize ayrıntılı bir şekilde “overnet” ağı ile ilgili ayrıntıları listeler.

**docker network inspect overnet**

Docker 1.12’den itibaren, docker network inspect komutunun sadece yerelde çalışan konteynerleri/görevleri göstermektedir. Bunun anlamı, yukarıdaki örnek için 10.0.0.3 ip’si ikinci node üzerinde çalışan ip bilgisidir. Buradaki bu ip bilgisini kendi ortamınız için not edin lütfen. Bir sonraki test içersinde onu kullanacağız.

## Adım 4: Ağı test edelim

Bu adımı tamamlamak için yukarıda belirttiğimiz gibi önceki adımda öğrendiğimiz **node2** için tanımlı olan servis görevini çalıştıran IP adresine ihtiyacımız olacak (10.0.0.3).

Birinci terminal ekranından(üstteki) burada yazılı olan komutu çalıştırın.

**docker network inspect overnet**

Servis görevini(service task) çalıştıran konteyner için listelenen IP adresinin, ikinci node üzerinde çalışan hizmet görevi(servis task) konteyner’i IP adresinden farklı olduğuna dikkat edin. Yani aynı hizmet görevi için çalışan her bir replika konteyner’in farklı bir ip adresi var, ayrıca bu iki service tastk konteynerinin aynı ağ üzerinde olduklarını unutmayın.

Servis görevi(service task) ID bilgisini alabilmek için tekrar docker ps komutunu çalıştıralım.

**docker ps**

Servis görevi(service task) konteyner’ine bağlanmak için ID bilgisini aldık, şimdi docker exec -it <CONTAINER ID> /bin/bash komutunu kullanalım ve konteyner içine bağlanalım.

**docker exec -it yourcontainerid /bin/bash**

**root@d676496d18f7:/#**

Ping komutunu kurun ve diğer servis görevi(service task) konteyneri üzerinde çalışan 10.0.0.3 ip adresini (sizin için farklı olacaktır) pingleyin.

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

şimdi 10.0.0.3 adresini pingleyelim.

**root@d676496d18f7:/# ping -c5 10.0.0.3**

**....**

Yukarıdaki çıktı bize **myservice** isimli servisin her iki node üzerine dağılmış bir şekilde aynı ağ üzerinde olduklarını gösteriyor ve ayrıca bu tanımlanmış overlay ağını iletişim için kullanabildiklerini gösteriyor.

## Adım 5: Servis keşfinin testi (service discovery)

Artık overlay ağ kullanan bir servisimiz olduğuna göre, servislerin keşfini test edebiliriz.

Şuan konteynerin içinde değilseniz, konteynırda tekrar bir oturum açın. Bunu yapmak için docker exec -it <CONTAINER ID> /bin/bash komutunu kullanabilirsiniz.

Konteyner içinden cat /etc/resolv.conf çalıştırın.

**docker exec -it yourcontainerid /bin/bash**

**cat /etc/resolv.conf**

**search ivaf2i2atqouppoxund0tvddsa.jx.internal.cloudapp.net**

**nameserver 127.0.0.11**

**options ndots:0**

Bizim ilgilendiğimiz değer buradaki nameserver 127.0.0.11 bilgisi. Buradaki değer sayesinde, konteyner içersindeki tüm DNS sorguları, gömülü olarak çalışan bir DNS çözümleyicisine gider (127.0.0.11:53 üzerinde çalışan). Tüm Docker konteynerleri buradaki adres üzerinde bir DNS sunucusu çalıştırırlar.

**NOT:** Dosya içersindeki (resolv.conf) diğer değerler çok daha farklı olabilir, ama biz temelde DNS çözümlemesi ile ilgileniyoruz.

“myservice” ismini konteyner içersinden pinglemeye çalışalım, bakalım bize nasıl bir sonuç dönecek ping -c5 myservice.

**root@d676496d18f7:/# ping -c5 myservice**

**PING myservice (10.0.0.2) 56(84) bytes of data.**

**64 bytes from 10.0.0.2: icmp\_seq=1 ttl=64 time=0.020 ms**

**64 bytes from 10.0.0.2: icmp\_seq=2 ttl=64 time=0.052 ms**

**64 bytes from 10.0.0.2: icmp\_seq=3 ttl=64 time=0.044 ms**

**64 bytes from 10.0.0.2: icmp\_seq=4 ttl=64 time=0.042 ms**

**64 bytes from 10.0.0.2: icmp\_seq=5 ttl=64 time=0.056 ms**

**--- myservice ping statistics ---**

**5 packets transmitted, 5 received, 0% packet loss, time 4001ms**

**rtt min/avg/max/mdev = 0.020/0.042/0.056/0.015 ms**

Gördüğümüz gibi myservice konteyner içersinden tanımlı bir dns adresidir myservice servisinin adını kullanarak pingleyebiliyoruz. Bir diğer önemli nokta geri dönen değerin 10.0.0.2 ip’si olmasıdır. Bir sonraki birkaç adımda bu ip adresinin myservice için tanımlanmış olan bir (VIP) Virtual IP (sanal ip) olduğunu göreceğiz.

exit yazarak konteyner içersinden çıkalım ve docker sunucumuz üzerine bağlanalım.

**root@d676496d18f7:/# exit**

“myservice” servisimizi inceleyelim, bunun için docker service inspect myservice komutunu kullanabiliriz. VIP değerinin ping -c5 myservice ile geri dönen değer olduğunu buradaki incelememiz sırasında görebiliriz.

**docker service inspect myservice**

Yukarıdaki tanımlar içersinde “VirtualIPs” içersinde 10.0.0.2 ip adresini görebilirsiniz. Buradaki VIP tanımı 10.0.0.2 ip adresidir fakat buradaki değer kurulumlara göre değişiklik gösterebilir(sizin ekranınıza farklı bir çıktı gelecektir). Buradaki en önemli nokta, tanımlı olan değer ile bizim konteyner içersinden ping -c5 myservice elde ettiğimiz değerin aynı olmasıdır.

**node2** (alttaki terminal) üzerinden konteyner içersinde yeni bir docker exec oturumu açarak ping -c5 service yapmayı deneyebilirsiniz. Sonucun yine tanımlı olan VIP olduğunu görebilirsiniz.

# **Şimdide yaptıklarımızı temizleyelim**

Umarım bu laboratuvarda Docker ağlarının nasıl çalıştığı hakkında biraz bilgi edinebildiniz. Beraber oluşturduğumuz servisi, başlattığımız konteynerleri temizleyelim ve sonunda Swarm modunu devre dışı bırakalım.

docker service rm myservice komutunu çalıştıralım ve myservice servisini silelim.

**docker service rm myservice**

docker ps komutunu çalıştıralım ve çalışan konteynerleri görelim.

**docker ps**

docker kill <KONTEYNER ID ...> kullanarak çalışan ubuntu ve nginx konteynerini durdurabilirsiniz.

**docker kill yourcontainerid1 yourcontainerid2**

Son olarak, Swarm içinden node1 ve node2 sunucularını çıkaralım. Bunun için docker swarm leave --force komutunu kullanabiliriz.

node1 üzerinde (üstte)docker swarm leave --force komutunu çalıştıralım.

**docker swarm leave –force**

mode2 üzerinde (altta)docker swarm leave --force komutunu çalıştıralım.

**docker swarm leave --force**