

Networking

Prerequisites - Switching routing

Aby połączyć ze sobą sieciowo dwa komputery podłączamy je do switcha i on tworzy sieć. Aby je podłączyć potrzebujemy network interface (może on być fizyczny lub virtualny) na każdym z nich (na każdym hoście).

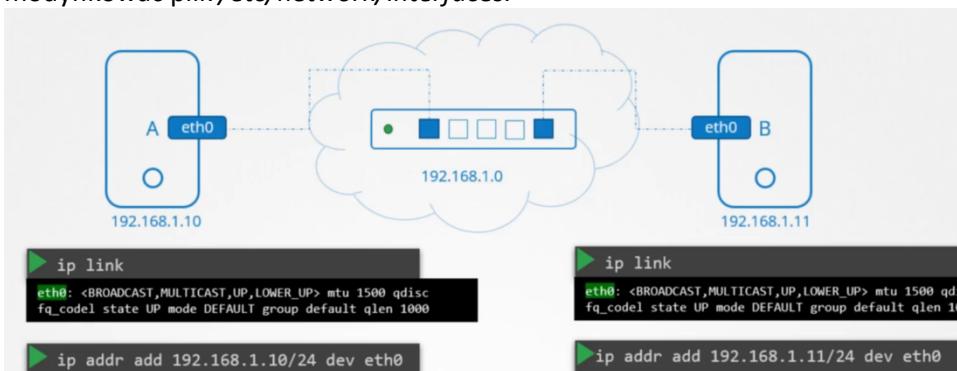
`ip link` - ta komenda pokaże i pozwoli modyfikować network interface na hoście

```
▶ ip link
eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
```

`Ip addr` - wyświetlenie adresów IP przypisanych do network interfaçów

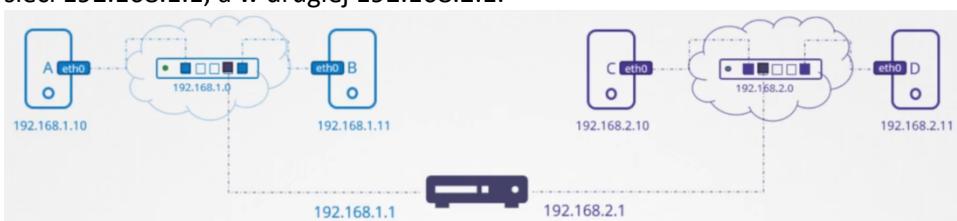
Zakładamy, że ta sieć to 192.168.1.0. Aby umożliwić połączenie nadajemy hostom adres IP zawierający się w przedziale sieci. Używamy do tego komendy: `ip addr add 192.168.1.10/24 dev eth0`

Należy pamiętać, że modyfikacja tych zmian będzie istnieć do rebootu. Aby ustawić to na stałe, należy modyfikować plik `/etc/network/interfaces`.



Teraz komputery mogą się ze sobą komunikować przez switch np. pingując adres IP drugiego komputera. Switch może odbierać dane i przesyłać do innych hostów, tylko wewnątrz danej sieci (tej, w której się znajduje).

Jeśli chcemy połączyć dwa komputery, które są w różnych sieciach, potrzebujemy Routera. Jest to inteligentne urządzenie i można o nim myśleć, jak o serwerze z wieloma portami sieciowymi. Jako, że łączy się z dwoma sieciami, będzie miał przypisane dwa adresy IP. Każdy w danej sieci. Np. w pierwszej sieci 192.168.1.1, a w drugiej 192.168.2.1.



Jeśli teraz np. system B chce wysłać paczkę do systemu C musi wiedzieć, gdzie jest ten Router (lub gateway). Jest on urządzeniem wewnątrz danej sieci, ale musi też być skonfigurowany bezpośrednio w systemie. Gateway to urządzenie, które otwiera ruch z danej sieci na zewnątrz do Internetu lub innych sieci. Żeby zobaczyć istniejącą konfigurację routingu na systemie używamy komendy `route`.

```
route
Kernel IP routing table
Destination     Gateway         Genmask        Flags Metric Ref    Use Iface
```

Wyświetli to routing table. W tym przykładzie na razie nie ma nic, więc system B nie połączy się z systemem C. Aby dodać wpis do routing table używamy komendy `ip route add` podajemy w niej zakres IP dodawanej sieci oraz IP adres Routera, przez który będzie szedł ruch:

```
ip route add 192.168.2.0/24 via 192.168.1.1
```

Ta konfiguracja musi być przeprowadzona na wszystkich systemach.

Jeśli te systemy chcieliby się łączyć z internetem np. chcieliby dostęp do Google przez 172.217.194.0 to trzeba byłoby dodać analogiczne ustawienie pozwalające na to żeby Router łączył się z tym adresem:

```
ip route add 172.217.194.0/24 via 192.168.2.1
```

Wiemy, że jest wiele stron internetowych na różnych serwerach, więc zamiast dodawać je pojedynczo do routing table, można powiedzieć, że dla każdej sieci, do której nie znamy route, należy użyć naszego Routera jako Default Gateway.

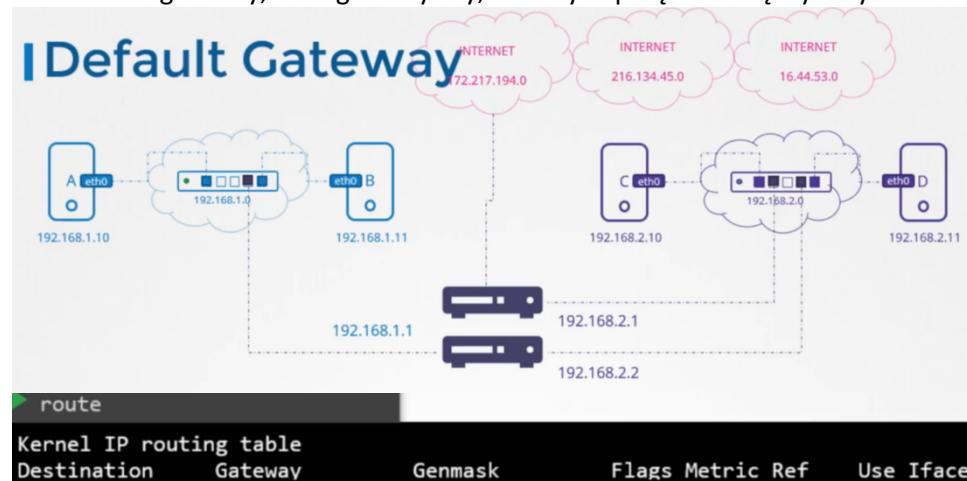
```
ip route add default via 192.168.2.1
```

Zamiast słowa default można też użyć 0.0.0.0. Jeśli dodamy wpis z Gatewayem jako 0.0.0.0 znaczy to, że nie potrzebujemy żadnego gatewaya żeby połączyć się z danym adresem IP. Taka sytuacja występuje wewnątrz danej sieci, wtedy nie potrzebujemy Gatewaya żeby łączyć się z innym systemem wewnątrz danej sieci:

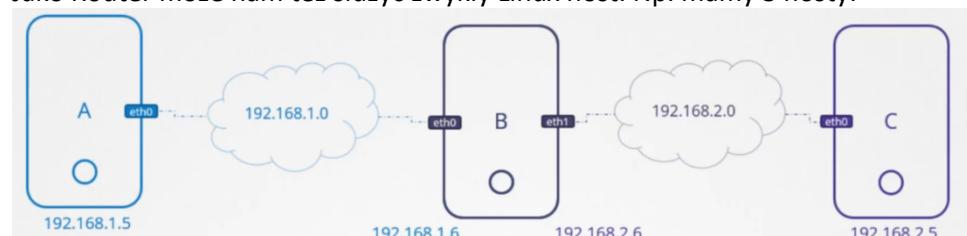
```
route
```

Kernel IP routing table						
Destination	Gateway	Genmask	Flags	Metric	Ref	Use Iface
default	192.168.2.1	255.255.255.0	UG	0	0	0 eth0
0.0.0.0	192.168.2.1	255.255.255.0	UG	0	0	0 eth0
192.168.2.0	0.0.0.0	255.255.255.0	UG	0	0	0 eth0

Jeśli mamy jednak 2 Gateways, jeden do internetu, a drugi do wewnętrznych sieci, to będziemy potrzebować dwóch wpisów do routing table. Jeden będzie do połączenia z wewnętrzną siecią przez 192.168.2.2 gateway, a drugi domyślny, do innych połączeń między innymi do internetu:



Jako Router może nam też służyć zwykły Linux host. Np. mamy 3 hosty:



Host B łączy się z sieciami A i C przez różne sieci. Chcemy aby sieć A łączyła się z C, ale teraz nie ma pojęcia jak dostać się do jego IP. Użyjemy więc hosta B jako Gatewaya. Dodajemy więc następujący wpis w routing table w hoście A:

```
ip route add 192.168.2.0/24 via 192.168.1.6
```

Analogiczny wpis dodajemy na hoście C żeby się łączyć z A.

Jeśli teraz zrobilibyśmy ping, to nie wyskoczy błąd Network is unreachable, ale nie będziemy też widzieć wysłanych paczek. Dzieje się tak, ponieważ domyślnie w Linuxie paczki nie są przesyłane z jednego interfejsu na inne. Np. paczki otrzymane z A w interfejsie eth0 w B nie będą przesłane do eth1 w B. Jest to z powodów bezpieczeństwa. Te ustawienia są w pliku `/proc/sys/net/ipv4/ip` Domyślnie jest tam wartość 0 - brak przesyłania. Jeśli zmienimy to na 1, wtedy komunikacja nastąpi. Należy pamiętać, że aby ta zmiana została zapisana po rebootie, trzeba to też skonfigurować w `/etc/sysctl.conf` Tam ustawiamy `net.ipv4.ip_forward=1`

Prerequisite - DNS

Mamy dwa komputery wewnątrz tej samej sieci, które mogą się ze sobą komunikować przy pomocy adresów IP. Wiemy, że na komputerze B mamy bazę danych, więc nadajemy mu nazwę db. Jeśli teraz chcielibyśmy zrobić ping z komputera A, to nie zadziała, musimy mu powiedzieć, że `db` to host pod danym adresem IP. Aby to zrobić, dodajemy wpis w pliku `/etc/hosts` podając IP i hostname:

```
> cat >> /etc/hosts  
192.168.1.11 db
```

Teraz ping już zadziała. Plik hosts to główne źródło informacji dla systemu, możemy tam nadać dowolne nazwy dla różnych adresów IP (nawet jeśli prawdziwie są one inne, np. gdy uruchomimy komendę `hostname` na docelowym systemie i będzie to inna wartość niż w pliku hosts na naszym komputerze) i będą one zawsze respektowane. Możemy też nadać wiele nazw dla tego samego adresu IP. Proces otrzymywania adresu IP poprzez hostname jest znany jako Name Resolution.

Co jednak jeśli mamy na serwerze wiele hostów i ich adresy IP się zmieniają? Ciężko byłoby to zarządzać manualnie, dlatego tutaj zajmuje się tym automatycznie dedykowany serwer. Jest to serwer DNS. Na serwerze DNS jest plik hosts, który posiada wszystkie rozwiązania adresów IP, a każdy komputer skonfigurowany tak, aby kierować zapytania o nieznane hosty właśnie do serwera DNS. W naszym przykładzie serwer DNS ma IP 192.168.1.100:



Aby skonfigurować serwer DNS na hoście musimy dodać to IP w pliku `/etc/resolv.conf`:

```
> cat /etc/resolv.conf  
nameserver 192.168.1.100
```

Jeśli adres IP dowolnego hosta by się zmienił, wystarczy zaktualizować server DNS i każdy host będzie w stanie to rozwiązać. Oprócz serwera DNS system będzie jednak ciągle korzystał z pliku hosts, więc jak mamy jakiś testowy endpoint, do którego chcemy się łączyć tylko z danego komputera, to możemy go umieścić w hostach.

Jeśli jednak ten sam wpis będzie dodany w hosts i w serwerze DNS, ale z różnymi adresami IP, to w pierwszej kolejności domyślnie system sprawdzi plik hosts i zostanie użyty jego adres IP. Możemy to zachowanie zmienić tak, by najpierw był sprawdzany DNS. Wtedy w pliku `/etc/nsswitch.conf` trzeba zmienić linijkę hosts: na wartość `dns files`.

Jeśli chcemy móc rozwiązywać strony internetowe to trzeba do ustawień w resolv.conf dodać dodatkowy serwer DNS, który zna adresy stron internetowych. Znany serwer DNS hostowany przez Google to 8.8.8.8.

Innym sposobem jest ustawienie naszego wewnętrznego serwera DNS żeby wysyłał wszystkie zapytania o nieznane adresy IP do innego serwera DNS i tu ustawić ten od Google.

Jeśli mamy adres strony internetowej zaczynający się od www i kończący się np. na com, to jest to Domain Name. Wszystkie domeny są pogrupowane na podstawie końcówki .com, .net itp. Wszystkie te końcówki to Top Level domains, które reprezentują intencję strony internetowej np. com oznacza reklamę lub ogólne użytkowanie, net to sieciowe rzeczy, edu edukacyjne, org to organizacje non profit. Składniki Domain Name:

. Jest to root, tu się wszystko zaczyna,

.com to top level domain,

Google to domain name,

Www to subdomain (pomaga ona grupować rzeczy wewnętrz domeny google). Mamy np. maps.google, drive.google itp. Możemy mieć wiele subdomain.

Jeśli chcemy się połączyć z apps.google.com z naszej organizacji to najpierw to pytanie idzie do naszego serwera DNS, potem on przesyła to do internetu i tam trafia ono na Root DNS server. On następnie przesyła je na .com DNS server, który ostatecznie przesyła to do Google DNS servera i on znajduje adres IP. Aby przyspieszyć ten proces w przyszłości nasz server DNS może zrobić cache tego połączenia.

Jeśli mamy wewnętrzną sieć w organizacji. Mamy domenę mycompany.com, subdomain mail, drive www itp. Oraz nasz server DNS. Serwer ten umożliwi rozwiązywanie wszystkich adresów IP poprzez ich domain name, ale co jeśli chcemy sobie dodać alias żeby używać np. web zamiast web.mycompany.com. Wtedy trzeba dodać kolejny wpis w /etc/resolv.conf. Dodajemy wpis search i tam podajemy domenę, którą będziemy chcieć zawsze dodawać do adresu, który szukamy. W tym przypadku będzie to mycompany.com i będzie to zawsze dodane do każdego zapytania:

```
▶ cat >> /etc/resolv.conf
nameserver      192.168.1.100
search          mycompany.com
```

Można w searchu dodać wiele domen do użycia i host będzie je wszystkie sprawdzał przy połączeniu. Jest on też inteligentny na tyle, by nie dodawać dwa razy tej samej domeny np. jak robimy ping web.mycompany.com to już nam drugi raz tej domeny nie doda.

Przechowywanie rekordów w serwerze DNS:

A record - jest to mapowanie IP do hostname,

AAAA record - mapowanie IPv6 do hostname,

CNAME - mapowanie jednego hostname do drugiego. Używane, gdy chcemy dodać alias.

Aby sprawdzić, czy dany hostname jest rozwiązywany poprawnie dobrze jest użyć komendy nslookup.

```
▶ nslookup www.google.com
Server:      8.8.8.8
Address:     8.8.8.8#53

Non-authoritative answer:
Name:   www.google.com
Address: 172.217.0.132
```

Trzeba pamiętać, że ta komenda nie bierze jednak pod uwagę wpisów z etc hosts file, więc tamtejsze wpisy nie będą rozwiązyane.

Kolejnym przydatnym tooliem jest *dig*. Podaje on więcej szczegółów o połączeniu:

```
▶ dig www.google.com

; <>> DiG 9.10.3-P4-Ubuntu <>> www.google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 28065
;; flags: qr rd ra; QUERY: 1, ANSWER: 6, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.google.com.           IN      A

;; ANSWER SECTION:
www.google.com.        245      IN      A      64.233.177.103
www.google.com.        245      IN      A      64.233.177.105
www.google.com.        245      IN      A      64.233.177.147
www.google.com.        245      IN      A      64.233.177.106
www.google.com.        245      IN      A      64.233.177.104
www.google.com.        245      IN      A      64.233.177.99

;; Query time: 5 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Sun Mar 24 04:34:33 UTC 2019
;; MSG SIZE  rcvd: 139
```

Prerequisite - CoreDNS

CoreDNS służy do konfiguracji hosta jako DNS server. Można go pobrać jako binarkę z GitHuba, lub jako Docker image.

```
▶ wget https://github.com/coredns/coredns/releases/download/v1.4.0/coredns_1.4.0_linux_amd64.tgz
coredns_1.4.0_linux_amd64.tgz

▶ tar -xvf coredns_1.4.0_linux_amd64.tgz
coredns

▶ ./coredns
.:53
2019-03-04T10:46:13.756Z [INFO] CoreDNS-1.4.0
2019-03-04T10:46:13.756Z [INFO] linux/amd64, go1.12,
8dccfc
CoreDNS-1.4.0
linux/amd64, go1.12, 8dccfc
```

192.168.1.10	web
192.168.1.11	db
192.168.1.20	web
192.168.1.21	db-1
192.168.1.22	nfs-1
192.168.1.30	web-1
192.168.1.31	db-2
192.168.1.32	nfs-2
192.168.1.40	web-2
192.168.1.41	sql
192.168.1.42	web-5
192.168.1.50	web-test
192.168.1.61	db-prod
192.168.1.52	nfs-4
192.168.1.60	web-3
192.168.1.61	db-test
192.168.1.62	nfs-prod



Uruchamiamy go przez *./coredns*, domyślnie nasłuchiwa on na porcie 53, który jest domyślnym dla serwera DNS.

Aby dodać wpisy hostname to IP do serwera DNS najpierw zaciągamy je wszystkie do pliku */etc/hosts* na tym serwerze (hoście). Następnie trzeba skonfigurować CoreDNS, aby czytał ten plik. CoreDNS ładuje wszystkie ustawienia z pliku Corefile, więc musimy to zrobić tam:

```
▶ cat > Corefile
· {
    hosts /etc/hosts
}
```

Wpisy można też dodawać w inny sposób np. poprzez plugin.

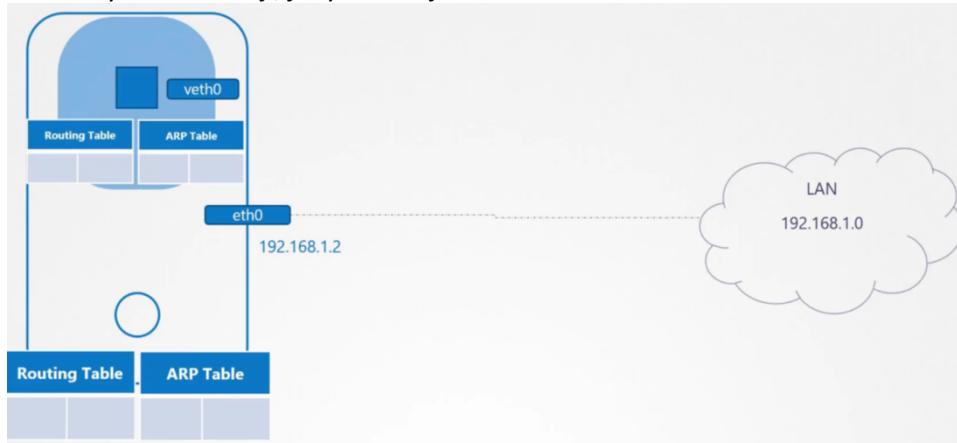
Prerequisite - Network Namespaces

Network namespaces są używane przez kontenery np. w Dockerze aby izolować sieci. Jeśli tworzymy kontenery na hoście, to chcemy żeby były one od siebie odizolowane i widziały tylko swoje procesy. Tą izolację realizują namespaces. Jeśli wylistujemy procesy w kontenerze, to będą widoczne tylko te, które sam stworzył. Jeśli jednak wylistujemy procesy na hoście to zobaczymy wszystkie procesy hosta i procesy wszystkich kontenerów:

```
ps aux (On the container)
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.0 4528 828 ? Ss 03:06 0:00 nginx

ps aux (On the host)
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
project 3720 0.1 0.1 95500 4916 ? R 06:06 0:00 sshd: project@pts/0
project 3725 0.0 0.1 95196 4132 ? S 06:06 0:00 sshd: project@notty
project 3727 0.2 0.1 21352 5340 pts/0 Ss 06:06 0:00 -bash
root 3802 0.0 0.0 8924 3616 ? Sl 06:06 0:00 docker-containerd-shim -namespace m
root 3816 1.0 0.0 4528 828 ? Ss 06:06 0:00 nginx
```

W przypadku sieci, nasz host ma network interface, poprzez który łączy się z siecią. Ma też swoje routing i ARP table. Gdy stworzymy kontener w network namespace na hoście to kontener nie będzie widział tych informacji, jedynie swoje sieciowe ustawienia.



Aby stworzyć nowy Network Namespace używamy komendy:

`ip netns add nazwa`

`ip netns` - wyświetli wszystkie istniejące network namespaces.

Jeśli chcemy wyświetlić network interface to używamy komendy: `ip link`. Jeśli jednak chcemy wyświetlić interface tylko wewnątrz danego namespace to musimy dodać odnośnik, że chcemy tą komendę uruchomić wewnątrz danego namespace:

`ip netns exec nazwa_namespace ip link`

Można też użyć: `ip -n nazwa_namespace link`

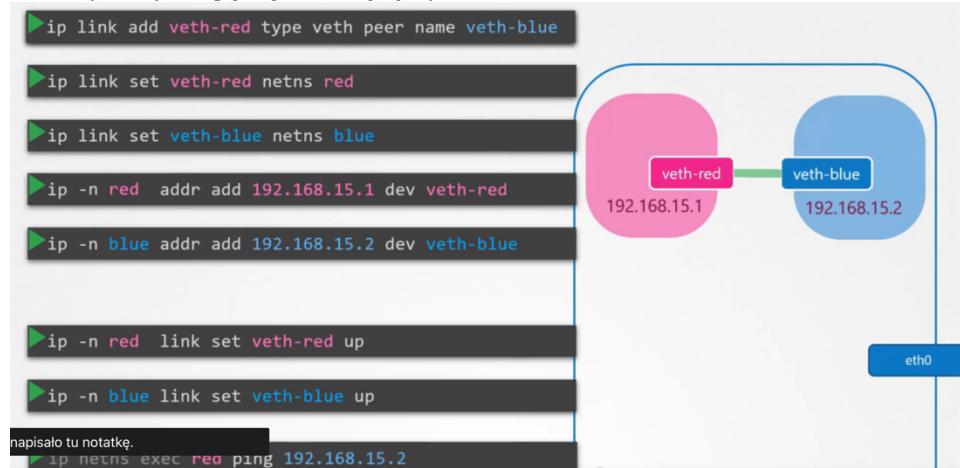
```
ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc state UP mode DEFAULT group 1000
    link/ether 02:42:ac:11:00:08 brd ff:ff:ff:ff:ff:ff

ip netns exec red ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

ip -n red link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

W ten sam sposób możemy sprawdzać ARP i routing Table. Komendą `arp` lub `route` wyświetlimy odpowiednią tabelę dla hosta, a komendą `ip netns exec nazwa arp (route)` - wyświetlimy tabelę dla danego namespace.

Jeśli chcemy umożliwić komunikację pomiędzy dwoma namespace, to możemy to zrobić za pomocą Virtual Ethernet Pair lub Virtual Cable. Aby stworzyć Virtual Cable używamy komendy: `ip link add veth-red type veth peer name veth-blue`. Precyzujemy tutaj nazwy dwóch network interface dla każdego namespace oraz rodzaj połączenia na veth. Następnym krokiem jest połączenie network interfaców z namespace'ami. Robimy to komendą: `ip link set veth-red netns red`. Następnie należy dodać adresy IP do obu network interfaców komendą: `ip -n red addr add 192.168.15.1/24 dev veth-red` (należy tutaj pamiętać, że trzeba dodać maskę sieci, czyli /24, jak tego nie zrobimy to może to powodować błędy połączenia). Potem trzeba uruchomić te interface'y: `ip -n red link set veth-red up`. Dopiero teraz namespace'y mogą się ze sobą łączyć.



Jeśli chcielibyśmy usunąć takie połączenie, to wystarczy to zrobić w jednym namespace i automatycznie zostanie też usunięte w drugim. Teraz, gdy wyświetlimy ARP Table na obu interfacach, to zobaczymy, że każdy ma już dodanego hosta do połączeń:

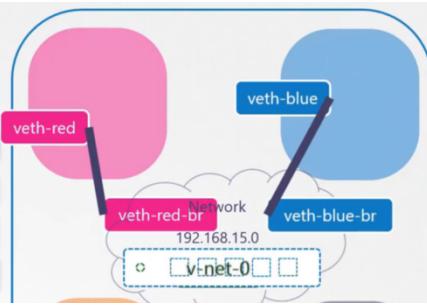
ARP Table					ARP Table				
Address	Hwtype	Hwaddress	Flags	Mask	Iface	Address	Hwtype	Hwaddress	Flags
192.168.15.2	ether	ba:b0:6d:68:09:e9	C		veth-red	192.168.15.2	ether	ba:b0:6d:68:09:e9	C
192.168.15.1	ether	7a:9d:9b:c8:3b:7f	C		veth-blue	192.168.15.1	ether	7a:9d:9b:c8:3b:7f	C

Jeśli wyświetlimy teraz ARP Table hosta, nie byłoby tam widać tych wpisów.

Co jeśli jednak mamy wiele namespace i chcemy uruchomić szybko komunikację pomiędzy nimi? Najlepiej stworzyć na hoście Virtual Network, ale żeby taka sieć powstała, to potrzebujemy też Virtual Switcha. Następnie łączymy do niego wszystkie namespace'y i tak umożliwimy komunikację. Jest wiele możliwości na tworzenie Virtual Switcha np. Linux Bridge, czy Open vSwitch. Tu użyjemy Linux Bridge. Aby stworzyć Virtual Bridge network dodajemy nowy network interface z typem bridge do hosta komendą: `ip link add v-net-0 type bridge`. Następnie trzeba uruchomić ten interface: `ip link set dev v-net-0 up`. Teraz musimy połączyć namespace'y z tym Virtual Switchem. Użyjemy do tego Virtual Cable. Tak jak poprzednio tworzymy najpierw Cable: `ip link add veth-red type veth peer name veth-red-br` (analogicznie dla innych namespace) i następnie łączymy jeden koniec Cable z namespace: `ip link set veth-red netns red`, a drugi ze switchem w stworzonym Virtual Network: `ip link set veth-red-br master v-net-0`.

LINUX BRIDGE

```
▶ ip link set veth-red netns red  
▶ ip link set veth-red-br master v-net-0  
▶ ip link set veth-blue netns blue  
▶ ip link set veth-blue-br master v-net-0
```



Następnie nadajemy adresy IP interfacom i je włączamy:

```
▶ ip -n red addr add 192.168.15.1 dev veth-red
```

```
▶ ip -n blue addr add 192.168.15.2 dev veth-blue
```

```
▶ ip -n red link set veth-red up
```

Teraz kontenery będą się mogły ze sobą komunikować przez sieć.

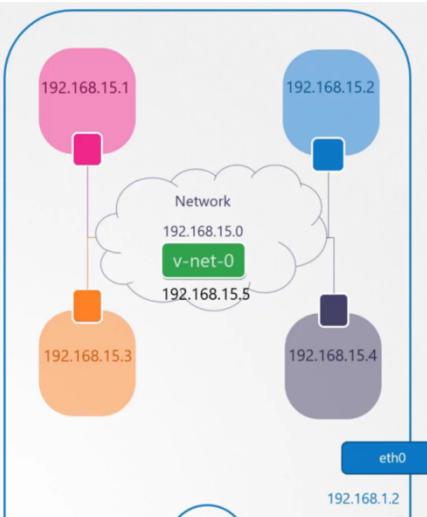
W tej konfiguracji, jeśli chcielibyśmy się połączyć z hostem na którym mamy kontenery bezpośrednio do kontenera, który jest w namespace, to nie uda się to, bo nie mamy jeszcze połączenia pomiędzy hostem (który jest w jednej sieci), a namespacesami (które są w Virtual Network). Jako że istniejący Switch jest tak naprawdę network interfelem dla hosta, jedyne co musimy zrobić to nadać mu adres IP i będziemy się już łączyć z hosta do kontenerów:

LINUX BRIDGE

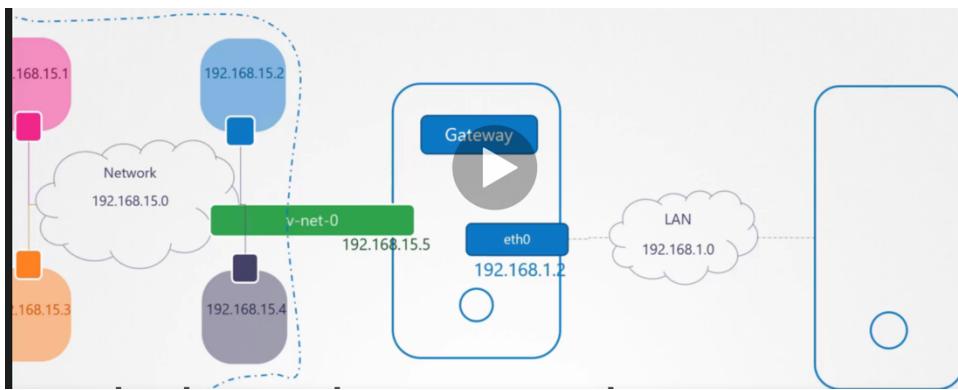
```
▶ ping 192.168.15.1  
Not Reachable!
```

```
▶ ip addr add 192.168.15.5/24 dev v-net-0
```

```
▶ ping 192.168.15.1  
PING 192.168.15.1 (192.168.15.1) 56(84) bytes of data.  
64 bytes from 192.168.15.1: icmp_seq=1 ttl=64 time=0.026 ms
```



Należy pamiętać, że cały czas Virtual Network jest wewnętrzną siecią, która nie ma dostępu do zewnętrznego świata. Jeśli chcielibyśmy się do niego łączyć, to musimy połączyć ją z istniejącym network interfelem eth0, który jest połączony do sieci LAN. Założmy, że chcemy się połączyć z innym hostem w sieci LAN o adresie 192.168.1.3 z niebieskiego kontenera. Kontener widzi, że przy pingowaniu tego IP jest to inna sieć (kontener ma 192.168.15 a tu jest 192.168.1), sprawdza więc w routing table, ale tam nie ma żadnej informacji na ten temat, więc połączenie zakończy się błędem. Trzeba więc dodać wpis w routing table, aby kontenery wiedziały, jakiego gatewaya muszą użyć, aby dostać się do internetu. W tym przypadku, wiemy, że to nasz host, na którym są kontenery będzie tym gatewayem, bo ma on 2 network interfeace - jeden łączy się z namespacesami, a drugi z siecią LAN, w której jest drugi host, do którego chcemy się połączyć:



Możemy teraz dodać wpis w routing table w niebieskim namespace mówiący, że jeśli chcemy się połączyć z **192.168.1.0/24** to mamy to robić przez **192.168.15.5**, czyli Virtual Switch łączący hosta z namespacesami:

```
ip netns exec blue ip route add 192.168.1.0/24 via 192.168.15.5
```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
192.168.15.0	0.0.0.0	255.255.255.0	U	0	0	0	veth-blue
192.168.1.0	192.168.15.5	255.255.255.0	UG	0	0	0	veth-blue

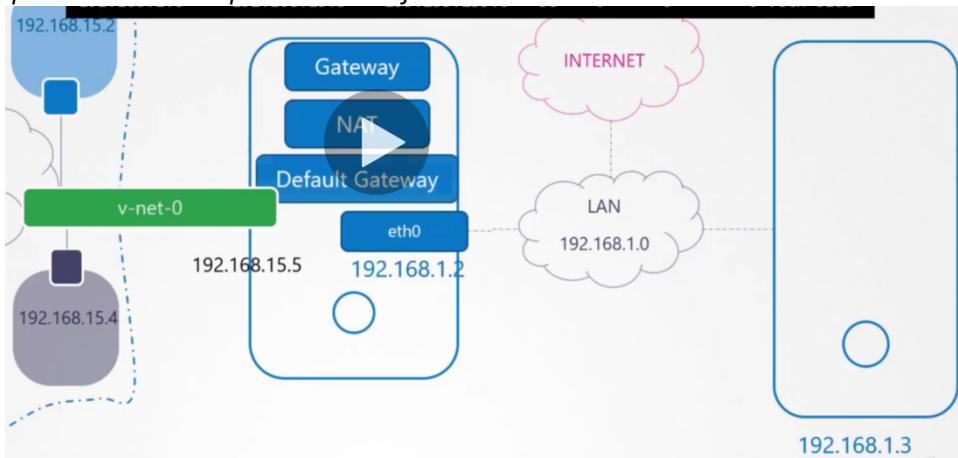
Teraz jeśli wyślemy ping do drugiego hosta z sieci LAN, to dane zostaną wysłane, ale nie dostaniemy żadnej odpowiedzi, bo sieć LAN będzie miała wewnętrzne IP niebieskiego kontenera, ale nie będzie wiedziała, jak do niego się dostać. W tym przypadku, przyda nam się tutaj NAT. Musimy włączyć go w głównym hoście, aby zachowywał się jako gateway i mógł wysyłać paczki do sieci LAN wraz ze swoją nazwą i swoim adresem IP. Aby to włączyć używamy komendy:

```
iptables -t nat -A POSTROUTING -s 192.168.15.0/24 -j MASQUERADE
```

Dodajemy w ten sposób NAT rulę z POSTROUTING chain aby móc "zamaskować" paczki z źródłowej sieci (niebieski kontener) adresem IP NAT gatewaya. W ten sposób każdy, kto dostanie paczkę będzie myślał, że przyszła ona bezpośrednio od głównego hosta i będzie mógł odesłać dane od siebie, bo do niego będzie znał drogę. Następnie wewnętrz hosta, paczka ta zostanie wysłana do faktycznego źródła, czyli niebieskiego kontenera.

Jeśli chcielibyśmy się połączyć z kontenera do Internetu np. do **8.8.8.8** (jeśli sieć LAN miałyby do niego dostęp), to znowu się nie uda, bo nie mamy wpisu w routing table w kontenerze. Możemy teraz ustawić nasz główny host jako Default Gateway dla kontenerów, tak aby każde połączenie z nieznanym dla kontenera adresem IP szło przez hosta:

```
ip netns exec blue ip route add default via 192.168.15.5:
```



Teraz połączenie do internetu zadziała.

Jeśli chcielibyśmy, aby zewnętrzny świat (inne hosty z LANa lub internet) łączył się bezpośrednio z kontenerem (np. ma on jakąś aplikację Webową), to mamy dwie opcje. Pierwsza to give away

tożsamości prywatnej sieci niebieskiego kontenera do zewnętrznego świata. Polega to na dodaniu IP route entry np. do drugiego hosta w LANie mówiąc mu, że wewnętrzna sieć 192.168.15 może być uzyskana przez głównego hosta 192.168.1.2. Druga opcja to dodanie PORTFORWARDING rule używając iptables mówiącej, że każdy ruch po porcie 80 przychodzący do głównego hosta należy przekazać na port 80 do IP przypisanego do niebieskiego kontenera:

```
Iptables -t nat -A PREROUTING --dport 80 --to-destination 192.168.15.2:80 -j DNAT
```

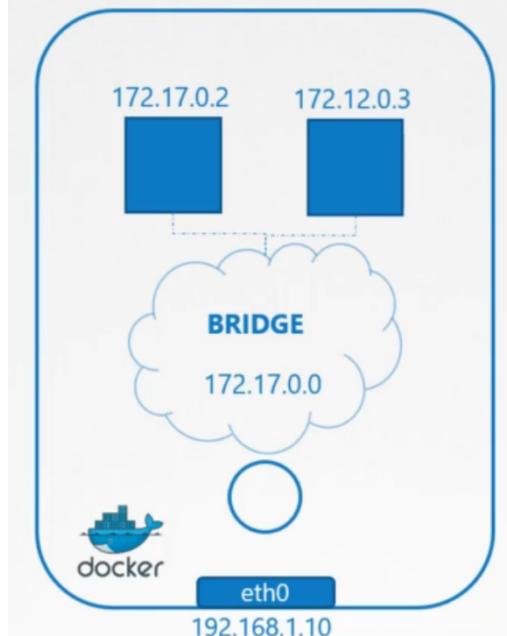
Prerequisite - Docker networking

Mamy serwer z zainstalowanym na nim Dockerem. Ma on network interface eth0 z adresem IP 192.168.1.10 poprzez który łączy się do lokalnej sieci. Jeśli teraz stworzymy kontener, to mamy wiele możliwości sieciowych:

`docker run --network none nginx` - z opcją none kontener nie jest podłączony do żadnej sieci i zewnętrzny świat nie ma do niego dostępu. Jak stworzymy kilka takich kontenerów, to nie będą one mogły ze sobą rozmawiać.

`docker run --network host nginx` - druga opcja to host network. W tym przypadku kontener jest podłączony do tej samej sieci co host i nie ma między nimi żadnej izolacji. Jeśli stworzymy np. aplikację w kontenerze na porcie 80, to będzie ona od razu dostępna dla wszystkich w sieci pod adresem IP hosta, bez żadnych dodatkowych mappingów. Jeśli stworzylibyśmy teraz drugi taki sam kontener to będzie błąd, bo dwa różne kontenery w sieci nie mogą jednocześnie nasłuchiwać na tym samym porcie.

`docker run --network bridge nginx` - w tym przypadku tworzona jest wewnętrzna prywatna sieć, do której będzie podłączony Docker host i w której będą kontenery. Założmy, że ta sieć ma adres IP 172.17.0.0. Domyślnie każde urządzenie w tej sieci (kontenery) dostanie swój prywatny adres IP. Tak samo każde urządzenie łączące się do tej sieci otrzyma swój prywatny adres sieciowy.



Trzeci rodzaj sieci jest najbardziej pożądany.

W momencie instalacji Dockera na hoście, tworzona jest wewnętrzna prywatna sieć nazwana domyślnie bridge (można to sprawdzić `docker network ls`), ale na hoście będzie ona nazwana docker 0 (można to zobaczyć przez `ip link`). Docker tworząc tą sieć używa komendy: `ip link add docker0 type bridge`.

Należy pamiętać, że sieć typu bridge jest jak interface dla hosta, ale jako switch dla namespace i kontenerów wewnętrznie. Dlatego też network interface `docker0` ma przypisany adres IP, tu `172.17.0.1` (można to zobaczyć komendą `ip addr`). Za każdym razem, gdy tworzony jest kontener,

Docker tworzy dla niego network namespace (można je wyświetlić poprzez `ip netns`). Można też zobaczyć namespace przypisane do kontenera używając komendy `docker inspect id_kontenera`.

BRIDGE

```
▶ ip addr
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    state DOWN group default
    link/ether 02:42:88:56:50:83 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/24 brd 172.17.0.255 scope global docker0
        valid_lft forever preferred_lft forever

▶ ip netns
b3165c10a92b

▶ docker inspect 942d70e585b2
"NetworkSettings": {
    "Bridge": "",
    "SandboxID": "b3165c10a92b50edce4c8aa5f37273e180907ded31",
    "SandboxKey": "/var/run/docker/netns/b3165c10a92b"

▶ docker run nginx
2e41deb9ef1b8b3d141c7bb55d883541b4
```

Następnie Docker musi połączyć sieć bridge z network namespace, w którym jest kontener. Tworzy więc Virtual Cable z dwoma network interfacami w każdym z końców. Jeśli uruchomimy `ip link` na Docker hoście, to zobaczymy jeden z końców połączony z docker0:

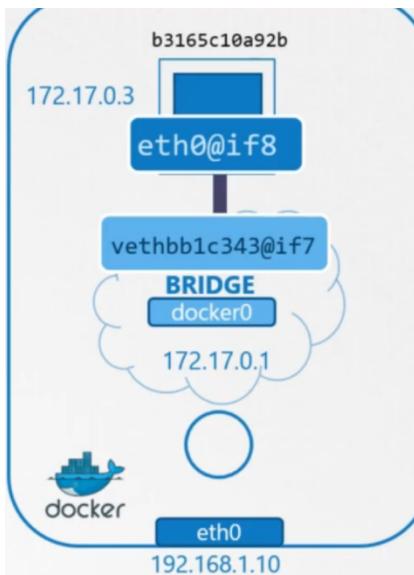
```
▶ ip link
...
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT
group default
    link/ether 02:42:9b:5f:d6:21 brd ff:ff:ff:ff:ff:ff
8: vethbb1c343@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
state UP mode DEFAULT group default
    link/ether 9e:71:37:83:9f:50 brd ff:ff:ff:ff:ff link-netnsid 1
```

Jeśli uruchomimy tą komendę wewnętrz network namespace to zobaczymy drugi koniec:

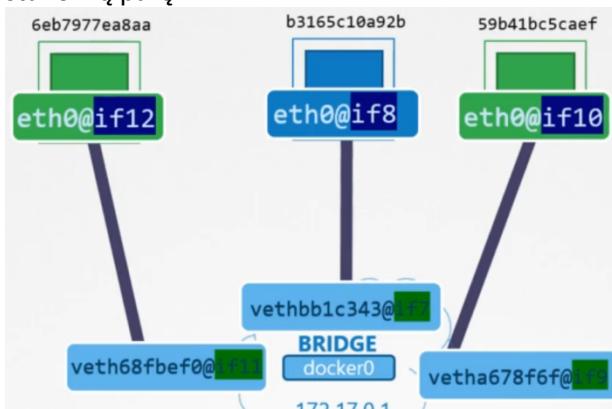
```
▶ ip -n b3165c10a92b link
...
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT
group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff link-netnsid 0
```

Tak jak w innych lekcjach interfacy dostają także adresy IP:

```
▶ ip -n b3165c10a92b addr
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
```



Ta sama procedura powtarza się zawsze, gdy tworzony jest nowy kontener. Pary network interfaców połączonych ze sobą można zidentyfikować po ich nazwach, na końcu są liczby i dwie obok siebie stanowią parę:



Co w przypadku portów? Założmy, że w naszym kontenerze jest web apka wystawiona na porcie 80. Teraz może się z nią tylko komunikować inny kontener oraz Docker host, bo są razem w sieci. Aby połączenie zadziałało z zewnątrz Docker umożliwia użycie port publishing lub port mapping. Należy powiedzieć Dockerowi w trakcie tworzenia kontenera, aby zmapował np. port 8080 na Docker hoście z portem 80 w kontenerze.

```
▶ docker run -p 8080:80 nginx
2e41deb9ef1b8b3d141c7bb55d883541b4d56c21cf055e236f870bd0f274e52b
```

Teraz aplikacja będzie dostępna pod 192.168.1.10:8080 - jest to IP network interface `eth0`. Używając tej komendy tworzymy NAT rulę w iptables: `iptables -t nat -A DOCKER -j DNAT --dport 8080 --to-destination 172.17.0.3:80` - Docker jako cel podaje od razu adres IP kontenera z portem. Można wyświetlić te rule na Docker hoście poprzez wyświetlenie iptables:

```
▶ iptables -nvL -t nat
Chain DOCKER (2 references)
target    prot opt source          destination
RETURN   all  --  anywhere        anywhere
DNAT     tcp  --  anywhere        anywhere          tcp dpt:8080 to:172.17.0.2:80
```

Prerequisite - CNI

Do tej pory wiemy jak stworzyć Network Namespace, Bridge Network, Virtual Cable, jak je ze sobą łączyć, jak dodać IP i je włączyć i skonfigurować NAT. W ten sam sposób takie połączenie sieciowe jest

realizowane przez wszystkie proramy do konteneryzacji (Docker, rkt, Mesos, Kubernetes itp.). Aby ułatwić to dla różnych providerów stworzono jeden program, który robi te same kroki sieciowe aby połączyć kontener z bridge network i nazwano go Bridge. Wystarczy użyć komendy: *bridge add container_ID namespace_location*:

```
▶ bridge add 2e34dcf34 /var/run/netns/2e34dcf34
```

Jeśli chcielibyśmy stworzyć taki program samemu, to musimy podązać za stworzonym już standardem, tak aby nasz program używał odpowiednich funkcji i miał odpowiednie argumenty. Dzięki temu wszystkie container solutions będą mogły z niego korzystać.

Tutaj właśnie CNI (Container Network Interface) przychodzi z pomocą. Jest to zbiór standardów definiujących, jak programy powinny być stworzone, by rozwiązywać zadania sieciowe w środowisku skonteneryzowanym. Programy te są określane, jako pluginy. W przypadku opisanym wyżej, mamy bridge plugin. CNI opisuje, jak w przypadku tworzenia kontenera ma się zachować plugin, a jak skonteneryzowane środowisko. Tutaj:

- środowisko musi stworzyć network namespace, zidentyfikować sieć, do której będzie podpięty kontener, uruchomić bridge plugin gdy kontener jest dodawany (ADD) lub usuwany (DEL), podać konfiguracje sieciową jako JSON file.

- plugin musi wspierać komendy ADD, DEL, CHECK, parametry container id, network ns itp., zarządzać przypisaniem adresu IP do poda, wyświetlić rezultaty w odpowiednim formacie.

Inne od razu zdefiniowane pluginy to VLAN, IPVLAN, MACVLAN, WINDOWS lub IPAM pluginy takie jak DHCP host-local, są też inne gotowe pluginy od 3rd party np. weave, flannel, calico itp. CNI dba o to aby każdy kontener mógł korzystać z każdego pluginu. Jedynym, który nie wspiera CNI jest Docker, ponieważ ma on swój własny standard CNM (Containter Networking Model). Przez to pluginy CNI nie działają w Dockerze.

```
Xdocker run --network=cni-bridge nginx
```

Nie znaczy to jednak, że nie można w ogóle korzystać z CNI w Dockerze. Można najpierw stworzyć kontener bez ustawień sieciowych, a potem samemu uruchomić CNI plugin.

```
▶ docker run --network=none nginx
```

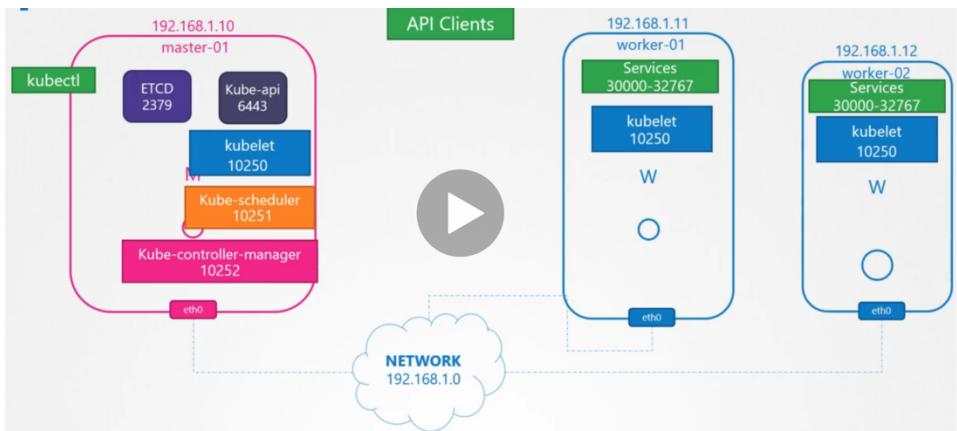
```
▶ bridge add 2e34dcf34 /var/run/netns/2e34dcf34
```

W ten sam sposób robi to Kubernetes, najpierw tworzy kontenery z *none* network, a następnie uruchamia skonfigurowany CNI plugin, aby dokończyć konfigurację.

Cluster Networking

Każdy node w Kuberntesie (master i worker) musi mieć przynajmniej jeden network interface, który go podepnie do sieci. Każdy interface musi mieć też adres IP, a każdy node hostname i MAC address. Musimy też otworzyć odpowiednie porty, z których korzystają komponenty Kuberntesa:

- master musi akceptować połączenia na 6443 do Kube-API
- kubelety na masterze i worker nodach nasłuchują na 10250
- kube-scheduler wymaga portu 10251
- kube-controller-manager wymaga portu 10252
- worker nody wystawiają serwisy dla zewnętrznego dostępu na portach 30000-32767
- etcd serwer nasłuchuje na porcie 2379



Jeśli mamy kilka masterów, to te same porty muszą być otwarte na wszystkich z nich. Dodatkowo trzeba otworzyć port 2380, aby ETCD clienty mogły się ze sobą komunikować pomiędzy masterami.

Przydatne komendy:

```

▶ ip link
▶ ip addr
▶ ip addr add 192.168.1.10/24 dev eth0
▶ ip route
▶ ip route add 192.168.1.0/24 via 192.168.2.1
▶ cat /proc/sys/net/ipv4/ip_forward
1
▶ arp
▶ netstat -plnt

```

▶ route

Pod tym adresem możemy znaleźć komendy przydatne do instalowania pluginów na clustrze:
[Creating Highly Available clusters with kubeadm | Kubernetes](#)

Ćwiczenia

Aby sprawdzić jaki network interface jest podpięty do noda, trzeba najpierw znaleźć IP noda (`kubectl describe node` i tam będzie Internal IP), a potem wylistować szczegółowo network interfacy (`ip a`) i zobaczyć, gdzie jest podpięte to IP.

Dwa nody mogą mieć ten sam MAC address.

`ifconfig -a` - pokazuje wszystkie network interfacy na danym node, inna opcja to otworzyć plik `/etc/network/interfaces` i tam będą opisane.

Komenda `netstat -nplt` - pokazuje programy oraz porty, adresy IP i ich PIDy. Przykładowy output:

`root@controlplane:~# netstat -nplt`

Active Internet connections (only servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN	761/sshd
tcp	0	0	127.0.0.11:33303	0.0.0.0:*	LISTEN	-
tcp	0	0	127.0.0.1:10248	0.0.0.0:*	LISTEN	4806/kubelet

tcp	0	0	127.0.0.1:10249	0.0.0.0:*	LISTEN	5873/kube-proxy
tcp	0	0	127.0.0.1:2379	0.0.0.0:*	LISTEN	3854/etcd
tcp	0	0	10.31.124.3:2379	0.0.0.0:*	LISTEN	3854/etcd
tcp	0	0	10.31.124.3:2380	0.0.0.0:*	LISTEN	3854/etcd
tcp	0	0	127.0.0.1:2381	0.0.0.0:*	LISTEN	3854/etcd
tcp	0	0	127.0.0.1:34031	0.0.0.0:*	LISTEN	4806/kubelet
tcp	0	0	0.0.0.0:8080	0.0.0.0:*	LISTEN	757/ttyd
tcp	0	0	127.0.0.1:10257	0.0.0.0:*	LISTEN	3837/kube-controlle
tcp	0	0	127.0.0.1:10259	0.0.0.0:*	LISTEN	4108/kube-scheduler
tcp	0	0	127.0.0.53:53	0.0.0.0:*	LISTEN	480/systemd-resolve
tcp6	0	0	::22	::*	LISTEN	761/sshd
tcp6	0	0	::8888	::*	LISTEN	5233/kubectl
tcp6	0	0	::10250	::*	LISTEN	4806/kubelet
tcp6	0	0	::6443	::*	LISTEN	3941/kube-apiserver
tcp6	0	0	::10256	::*	LISTEN	5873/kube-proxy

Pod networking

Oprócz ustawienia głównej sieci, tak aby wszystkie nody w clustrze mogły się ze sobą komunikować, bardzo ważna jest też warstwa sieciowa dla podów. Pozwoli to na komunikację pomiędzy podami z różnych nodów, dostęp do serwisów itp. Kubernetes nie ma automatycznego toola, który to rozwiąże, więc musimy zrobić to sami. Wymagania to: każdy pod musi mieć swój adres IP, każdy pod powienien być w stanie komunikować się z innym podem na tym samym node, każdy pod powinien być w stanie komunikować się z podami na innych nodach bez NATu.

Jest wiele networking solutions, które to umożliwiają (np. flannel, vmware nsx, cilium itp.), ale jeśli chcemy zrobić to sami to należy wykonać poniższe kroki.

Mamy 3 nody w clustrze, są one częścią zewnętrznzej sieci i mają adresy IP w przedziale 192.168.1.0. Następnie w nodach mamy stworzone kontenery i Kubernetes tworzy dla nich namespaces. Aby umożliwić komunikację pomiędzy namespacesami trzeba je połączyć do sieci, w tym przypadku najlepiej to zrobić z bridge network, więc tworzymy ten rodzaj sieci w każdym nodzie:

ip link add v-net-0 type bridge

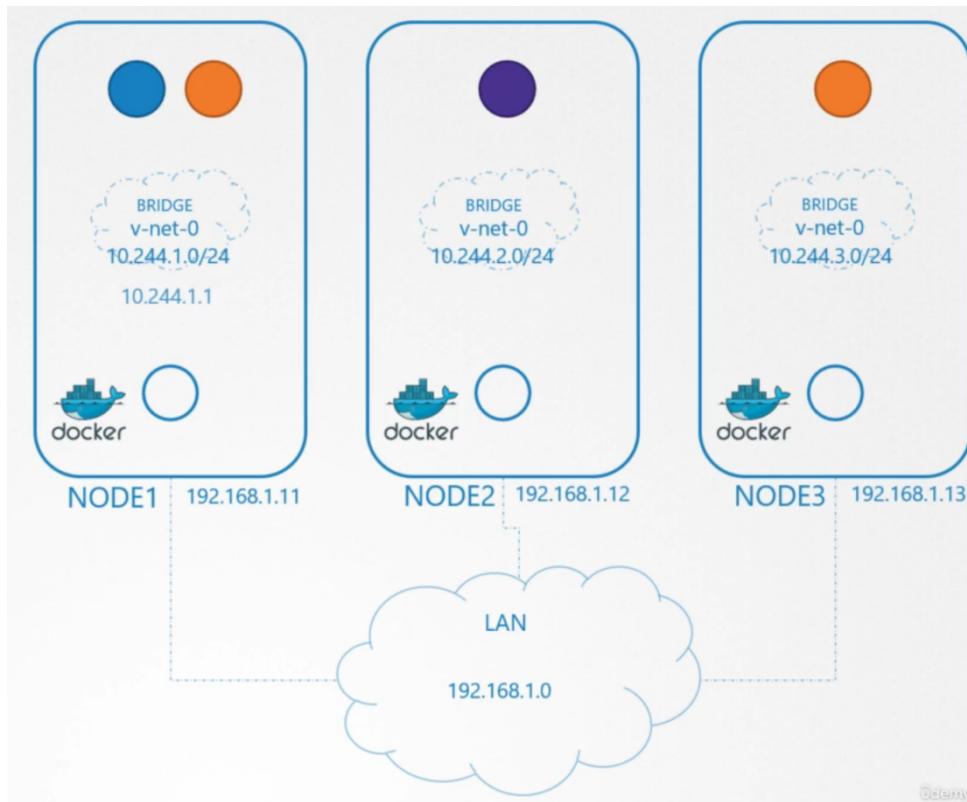
Następnie je uruchamiamy:

ip link set dev v-net-0 ip

Następnie trzeba przypisać adresy IP do bridge network. Należy pamiętać, żeby każda bridge network miała swoją własną podsieć, to znaczy, żeby adresy IP w różnych nodach się nie pokrywały. Tutaj nadajemy 10.244.1.0/24, 10.244.2.0/24, 10.244.3.0/24. Teraz przypisujemy adresy IP do network interfaców w bridge network:

Ip addr add 10.244.1.1/24 dev v-net-0

Podstawa jest zrobiona:



Teraz będzie wiele kroków do zrobienia dla kontenera, oraz w momencie, gdy kontener będzie stworzony, więc najlepiej napisać skrypt. Potrzebujemy komend:

ip link add ... - żeby stworzyć virtual cable

ip link set ... - aby połączyć jeden koniec kabla z kontenerem, a drugi z bridge network.

ip -n <namespace> addr add ... - przypisanie adresu IP do kontenera

ip -n <namespace> route add ... - dodanie route dla default gatewaya

ip -n <namespace> link set ... - uruchomienie network interface w namespace kontenera

net-script.sh

```
# Create veth pair
ip link add .....

# Attach veth pair
ip link set .....
ip link set .....

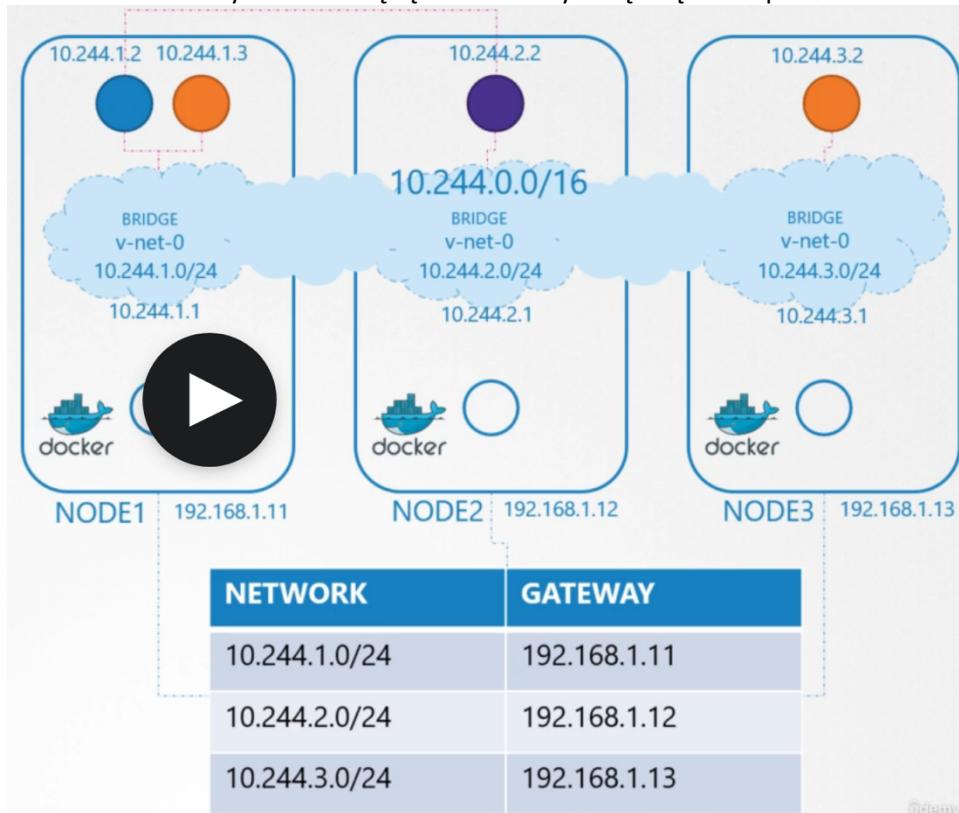
# Assign IP Address
ip -n <namespace> addr add .....
ip -n <namespace> route add .....

# Bring Up Interface
ip -n <namespace> link set .....
```

Ten skrypt trzeba uruchomić dla wszystkich podów na wszystkich nodach i to sprawi, że pody wewnętrz nodą będą mogły się ze sobą komunikować. Teraz, aby mogły się dodatkowo komunikować pomiędzy nodami musimy dodać wpisy do routing table.

ip route add 10.244.2.2 via 192.168.1.12 - dodanie np. w podzie 1 wpisu do łączenia się z podem w drugim node. Takie same wpisy trzeba dodać na wszystkich podach, by mogły się łączyć pomiędzy clustrami. To zadziała, jak mamy mało podów, ale przy większej ilości lepiej jest konfigurować routing

table bezpośrednio na routerze, jeśli jest w sieci. Następnie trzeba ustawić wszystkie kontenery, aby używały go, jako default gateway. W ten sposób łatwo zarządzać wszystkimi routami. Lokalne sieci stworzone na każdym nodzie będą teraz tworzyć większą sieć o przedziale 10.244.0.0/16:



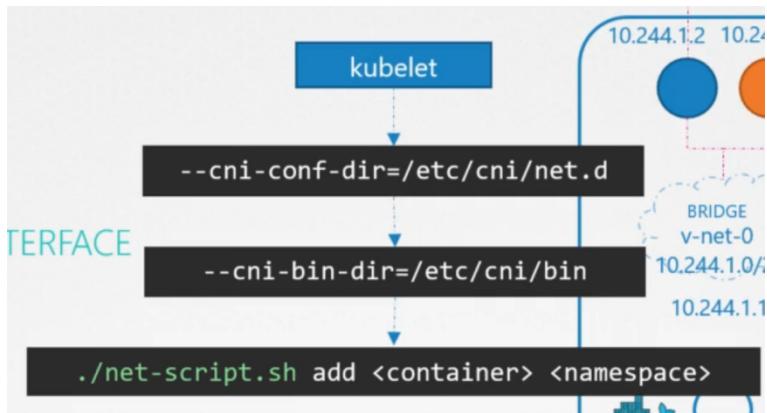
Jeśli chcielibyśmy zautomatyzować ten proces, to możemy użyć CNI. Mówiąc o Kubernetesowi, w jaki sposób powinien uruchamiać skrypt w momencie tworzenia kontenera. Nam CNI mówi, jak powinien wyglądać skrypt, aby spełniać standardy. Musimy nieco zmodyfikować nasz skrypt dodając sekcję ADD i DELETE opisujące, jakie rzeczy możemy dodawać i jakie usuwać:

```
net-script.sh
ADD)
# Create veth pair
# Attach veth pair
# Assign IP Address
# Bring Up Interface
ip -n <namespace> link set .....

DEL)

# Delete veth pair
ip link del .....
```

Na każdym nodzie kubelet jest odpowiedzialny za tworzenie kontenerów, więc trzeba mu w opcjach dodać argument `--cni-conf-dir=/etc/cni/net.d` aby mógł znaleźć nazwę skryptu, następnie szuka go w `-cni-bin-dir=/etc/cni/bin` i go uruchamia z komendą add: `./net-script.sh add nazwa_kontenera namespace.`



Teraz skrypt zajmie się resztą.

CNI in Kubernetes

Zobaczmy jak Kubernetes jest skonfigurowany, aby używać network plugins. Jak wiemy, CNI określa wymagania dla Kuberntesa (jako tworzyciela kontenerów). Kubernetes musi tworzyć network namespace, zidentyfikować sieć, do której podejmie kontener, musi uruchomić Network Bridge plugin, gdy kontener jest dodawany i usuwany, oraz wszystkie konfiguracje sieciowe muszą mieć format .json.

Jako że kubelet jest używany do tworzenia kontenerów, to właśnie w nim trzeba ustawić argumenty związane z CNI:

```
--network-plugin=cni
--cni-bin-dir=/opt/cni/bin (domyślna lokalizacja)
--cni-conf-dir=/etc/cni/net.d
```

```

[kubelet.service]
ExecStart=/usr/local/bin/kubelet \\
  --config=/var/lib/kubelet/kubelet-config.yaml \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --image-pull-progress-deadline=2m \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --cni-bin-dir=/opt/cni/bin \\
  --cni-conf-dir=/etc/cni/net.d \\
  --register-node=true \\
  --v=2
  
```

Te same informacje znajdziemy, gdy wyświetlimy proces kubeleta:

```

▶ ps -aux | grep kubelet
root      2095  1.8  2.4 960676 98788 ?        Ssl  02:32   0:36 /usr/bin/kubelet --bootstrap-
kubernetes=kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --cgroup-driver=cgroups --cni-bin-dir=/opt/cni/bin --cni-conf-dir=/etc/cni/net.d --network-plugin=cni
  
```

W CNI bin directory mamy wszystkie binarki pluginów cni:

```

▶ ls /opt/cni/bin
bridge  dhcp  flannel  host-local  ipvlan  loopback  macvlan  portmap  ptp  sample  tuning
vlan   weave-ipam  weave-net  weave-plugin-2.2.1
  
```

W CNI conf directory jest konfiguracja i właśnie tam kubelet patrzy, jakiego pluginu powinien użyć.

Tak wygląda plik konfiguracyjny, określony przez standardy CNI:

```
▶ cat /etc/cni/net.d/10-bridge.conf
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

Są tu wszystkie ustawienia - typ pluginu, bridging, routing, ipMasq (czy ma być NAT), isGateway (czy bridge network ma mieć adres IP żeby być gatewayem), ipam (tu ustawiamy, jakie zakresy IP mają być dodawane do podów wraz z potrzebnymi routami).

CNI weave

Mamy już swój skrypt, który stworzyliśmy i zintegrowaliśmy z kubeletem przez CNI. Teraz możemy skorzystać z gotowego pluginu, aby nie łączyć sieciowo wszystkich podów manualnie. Rozwiążanie z routing table też nie było idealne, bo jeśli mielibyśmy kilkanaście nodów z setkami podów, to mógłby być problem. Dlatego najlepiej użyć tutaj pluginu. Przykładowo Weave Works.

Instaluje on swojego agenta na każdym z nodów, agenci łączą się ze sobą w celu wymiany informacji o resourcach na każdym z nodów. Każdy agent przechowuje strukturę całej architektury sieci i podów, tak aby wszyscy znali pody ich adresy IP. Następnie Weave tworzy swoją własną sieć bridge i nazywa ją weave, przypisuje adresy IP do każdej z sieci. Należy pamiętać, że każdy pod może być przypisany do kilku sieci na raz, np. weave i docker bridge jednocześnie. Weave dba o to aby pody miały ustawione odpowiednie routy, aby dostać się do agenta.

Jeśli wysyłamy paczkę do poda na innym node, to przechwytuje ją agent i widzi, że cel jest w innej sieci. W takiej sytuacji następuje enkapsulacja paczki i zostaje stworzone nowy adresat i nadawca paczki. Paczka ta zostaje wysyłana w sieci agentów, tak aby każdy mógł ją sprawdzić. Agent na innym node dekapsuluje ją i otrzymuje oryginalnego adresata i nadawcę, więc wysyła ją do odpowiedniego poda.

Najłatwiej zdeployować weave w Kubernetesie jako poda. Jak już mamy nody i połączenie sieciowe między nimi, oraz wszystkie podstawowe komponenty controlplane są ustawione, możemy zdeployować weave tą komendą apply:

```
▶ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.extensions/weave-net created
```

Stworzy ona wszystkie potrzebne komponenty, a weave będzie w postaci DeamonSetu, tak aby zawsze na każdym nodzie był stworzony pod weave.

Jeśli stworzyliśmy cluster przy pomocy kubeadm tool i weave plugin, to w namespace kube-system będą już stworzone odpowiednie pody. Jeśli chcemy sprawdzić jakiś troubleshooting, to możemy zobaczyć ich logi:

```
▶ kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS   AGE     IP           NODE
NOMINATED NODE
coredns-78fcdf6894-99khw          1/1    Running   0          19m    10.44.0.2   master   <none>
coredns-78fcdf6894-p7dpj          1/1    Running   0          19m    10.44.0.1   master   <none>
etcd-master                         1/1    Running   0          18m    172.17.0.11  master   <none>
kube-apiserver-master             1/1    Running   0          18m    172.17.0.11  master   <none>
kube-scheduler-master              1/1    Running   0          17m    172.17.0.11  master   <none>
weave-net-5cmb                     2/2    Running   0          19m    172.17.0.30  node02  <none>
weave-net-fr9n9                    2/2    Running   0          19m    172.17.0.11  master   <none>
weave-net-mc6s2                    2/2    Running   0          19m    172.17.0.23  node01  <none>
weave-net-tbzvz                   2/2    Running   0          19m    172.17.0.52  node03  <none>

▶ kubectl logs weave-net-5cmb weave -n kube-system
```

Instalacja Weave plugin: [Integrating Kubernetes via the Addon \(weave.works\)](https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')&env.IPALLOC_RANGE=10.50.0.0/16)

Konfiguracja Weave: [Integrating Kubernetes via the Addon \(weave.works\)](https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')&env.IPALLOC_RANGE=10.50.0.0/16)

Ćwiczenie:

Najpierw należy sprawdzić, czy domyślny adres IP, który jest przypisywany do Weave, nie pokryje się z przedziałem IP przypisanym do hosta Kurnetesa. Zakres IP hosta - `ip a / grep eth0`. Tu tak jest, bo domyślny dla Weave to 10.32.0.0/12. Znajdziemy na to potwierdzenie w logach:

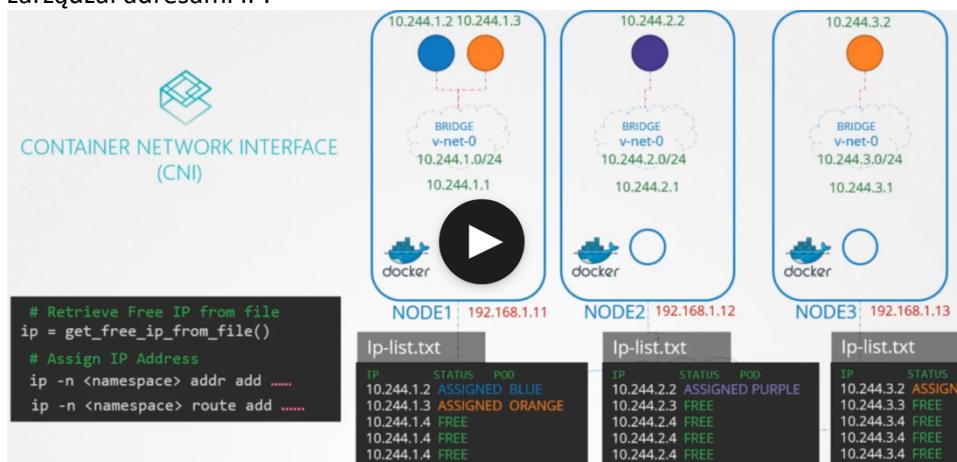
`root@controlplane:~# kubectl logs -n kube-system weave-net-6mckb -c weave Network 10.32.0.0/12 overlaps with existing route 10.40.56.0/24 on host`

Jeśli chcemy zmienić domyślny zakres IP dla Weave, to trzeba go podać w zmiennej przy pobieraniu plików:

`kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')&env.IPALLOC_RANGE=10.50.0.0/16"`

IP Address Management (IAM) - Weave

Nie będziemy tu omawiać przypisywania adresów IP do nodów. Omówimy, jak virtual bridge network w nodzie otrzymuje swój subnet i jak potem pody dostają adresy IP. Pamiętamy, że za te kroki według CNI jest odpowiedzialny plugin i ważne jest żeby żaden adres IP się nie powtórzył zwłaszcza, gdy mamy dużo podów. Najłatwiej więc jest przechowywać tę listę w pliku i napisać kod, który będzie zarządzać listą, tak aby nie było żadnych duplikatów. Taki plik będzie umieszczony na każdym nodzie i będzie zarządzał adresami IP:



CNI ma dwa pluginy, które mogą wykonać to zadanie za nas automatycznie. Są to DHCP i host_local, jednak musimy pamiętać, że to my musimy uruchomić plugin naszym skryptem (można ustawić też

skrypt tak, aby wybierał różne pluginy). W pliku konfiguracyjnym CNI jest sekcja ipam->type, w której ustawiamy ten plugin oraz subnet i routy:

```
▶ cat /etc/cni/net.d/net-script.conf
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "net-script",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.244.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
used
```

Jeśli chodzi o Weave, on domyślnie rozdziela adresy IP z przedziału: 10.32.0.0/12 (od 10.32.0.1 do 10.47.255.254) dla całej sieci. Z tego przedziału Weave agenci rozdzielają te adresy IP po równo dla wszystkich nodów, więc znajdujące się na nich pody będą stąd brać swoje adresy IP. Oczywiście wartości te można zmienić w trakcie deployowania Weave pluginu na clustrze.

Ćwiczenie:

Aby wyświetlić zakres IP połączony z danym network interface: *ip addr show nazwa_interface*

Sprawdzamy default Gateway na nodzie 01, więc trzeba tam stworzyć poda. Najpierw robimy plik yaml:

```
kubectl run test_pod --image=busybox --command sleep 1000 --dry-run=client -o yaml > pod.yaml
```

Dodajemy nodeName w sekcji spec:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: test_pod
  name: test_pod
spec:
  nodeName: node01
  containers:
  - command:
    - sleep
    - "1000"
    image: busybox
    name: test_pod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  status: {}
```

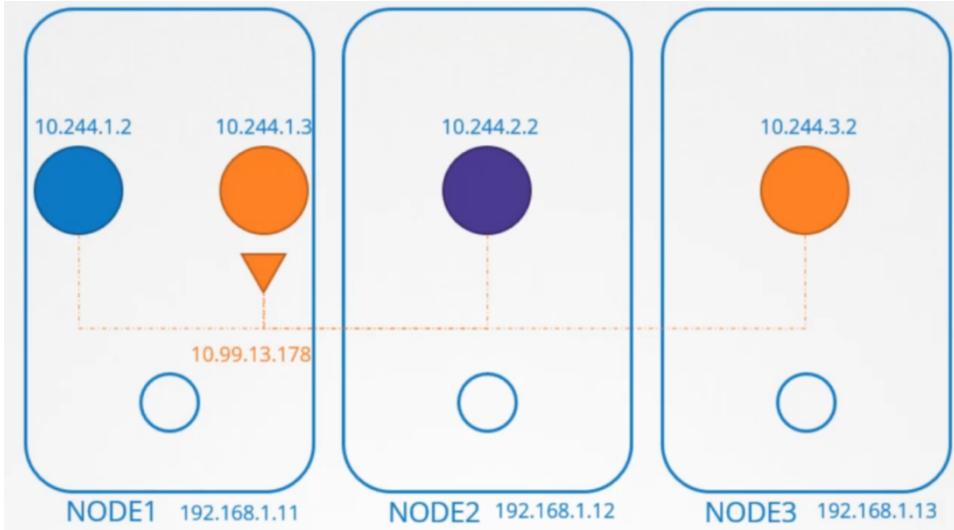
Teraz tworzymy poda i potem uruchamiamy w nim komendę *route*.

Service Networking

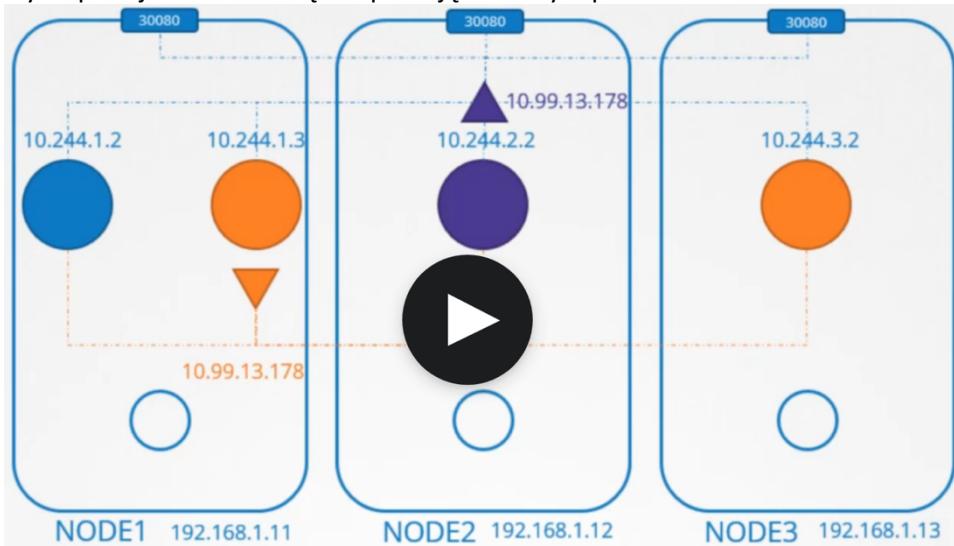
W poprzednich wykładach łączyliśmy sieciowo pody, jednak rzadko zdarza się, że łączymy się bezpośrednio z podem, bo jeśli chcemy się dostać do jego serwisu, to będziemy się łączyć z Service.

Rodzaje Service:

-Np. tworząc pomarańczowy Service umożliwiamy dostęp dla niebieskiego poda do kontaktu z pomarańczowym podem na nodzie 1. Gdy stworzymy service jest on też dostępny dla wszystkich podów w clustrze, bez względu na to, na jakich nodach są. Trzeba pamiętać, że service nie jest przypisany do żadnego noda, dodatkowo nie jest też dostępny z zewnątrz. Taki rodzaj to ClusterIP.



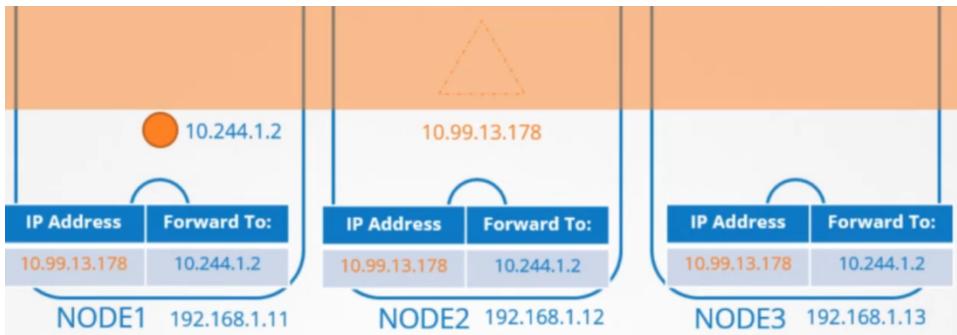
-Załóżmy, że na fioletowym podzie mamy web apkę, która udostępnia ten pod na zewnątrz. Potrzebujemy do tego kolejnego serwisu o typie NodePort. Dostanie on także swój adres IP. Działa on podobnie jak Cluster IP, więc wszystkie pody będą mogły się do niego łączyć, ale dodatkowo wyeksponuje on na zewnątrz aplikację na danym porcie na nodach w clustrze:



Aby service był automatycznie dostępny dla wszystkich podów, aby udostępniał aplikację na zewnątrz na danym porcie trzeba wykonać odpowiednie ustawienia sieciowe. Pamiętamy, że na każdym node mamy Kubelet, który tworzy pody na nodach i potem uruchamia CNI plugin żeby stworzyć ustawienia sieciowe. Analogicznie zachowuje się inny komponent - kube-proxy, za każdym razem, gdy nowy service jest stworzony, komponent ten wykonuje akcje. Najpierw service dostaje adres IP, który jest dobrany z domyślnego zakresu. Można go oczywiście zmienić funkcją (przy zmianie zakres ten nie może pokrywać się z zakresem sieci podów):

```
kube-api-server --service-cluster-ip-range ipNet (Default: 10.0.0.0/24)
```

Następnie kube-proxy bierze to IP i tworzy sieciowe rule mówiące, aby każde połączenie do IP serwisu, było przekierowane do danego poda (połączenie to określa też konkretny port):



Za każdym razem, jak service jest tworzony lub usuwany, kube-proxy tworzy lub usuwa te rule.

Tworzenie ruli. Są trzy metody: userspace - kube-proxy nasłuchuje na porcie dla każdego service i przekazuje połączenia do podów, tworzenie ipvs rule lub iptables. Sposób tworzenia można ustawić przez:

`kube-proxy --proxy-mode [userspace / iptables / ipvs]` - domyślny to iptables.

Przykładowo mamy pod db i chcemy do niego stworzyć service. Po przypisaniu do niego adresu IP, możemy sprawdzić stworzone przez kube-proxy rule komendą:

```
▶ iptables -L -t nat | grep db-service
KUBE-SVC-XA50GUC7YRH053PU  tcp  --  anywhere  10.103.132.104  /* default/db-service: cluster IP */ tcp dpt:3306
DNAT
KUBE-SEP-JBWCWHQM57V2WN7  all  --  anywhere  anywhere      /* default/db-service: */ tcp to:10.244.1.2:3306
KUBE-SEP-JBWCWHQM57V2WN7  all  --  anywhere  anywhere      /* default/db-service: */
```

Ta rula mówi, że każdy ruch przychodzący do 10..103.132.104:3306 powinien być przekierowany do 10.244.1.2:3306. Jest to tNAT rule dodana do iptables. Tak samo będą tworzone rule w przypadku innego typu service np. NodePort. Informacje o tworzeniu ruli możemy też znaleźć w logach kube-proxy:

```
▶ cat /var/log/kube-proxy.log
I0307 04:29:29.883941 1 server_others.go:140] Using iptables Proxier.
I0307 04:29:29.912037 1 server_others.go:174] Tearing down inactive rules.
I0307 04:29:30.027360 1 server.go:448] Version: v1.11.8
I0307 04:29:30.049773 1 conntrack.go:98] Set sysctl 'net/netfilter/nf_conntrack_max' to 131072
I0307 04:29:30.049945 1 conntrack.go:52] Setting nf_conntrack_max to 131072
I0307 04:29:30.050701 1 conntrack.go:83] Setting conntrack hashsize to 32768
I0307 04:29:30.050701 1 proxier.go:294] Adding new service "default/db-service:3306" at 10.103.132.104:3306/TCP
```

Ćwiczenia:

Aby sprawdzić zakres IP, z którego są przypisywane adresy do nodów. Uruchamiamy `ip addr` i sprawdzamy zakres dla `eth0`.

Jeśli chcemy zobaczyć zakres IP przypisany do podów w clustrze to najlepiej sprawdzić logi network pluginu, ponieważ, to on tworzy wszystkie połączenia sieciowe - tutaj mamy weave, więc `kubectl logs weave_pod weave -n kube-system` i tam szukamy `ipalloc-range`.

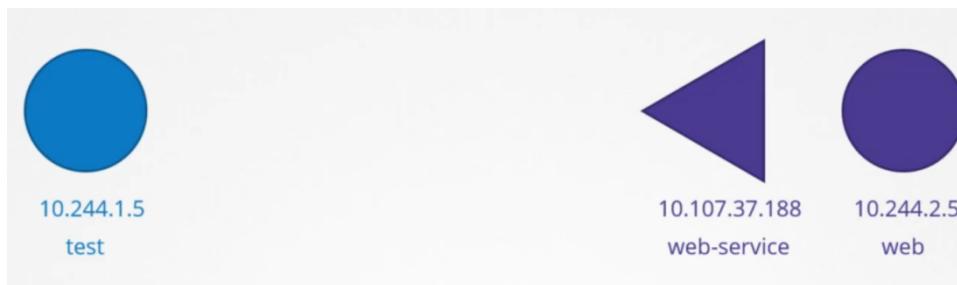
Jeśli chcemy zobaczyć zakres IP dla serviców to szukamy go w ustawieniach kube-api servera: `kubectl describe pod kube-apiserver-controlplane -n kube-system | grep ip-range`

Aby sprawdzić jakiego rodzaju proxy (userspace, iptables, czy ipvs) używa kube-proxy trzeba wejść w logi jego poda i tam to znaleźć.

DNS in Kubernetes

Mamy cluster z trzema nodami, a w nich pody i service. Każdy node ma swój przypisany hostname. W trakcie tworzenia clustra Kubernetes tworzy także wewnętrzny Kube DNS server. Założmy, że ustawienia sieciowe są już zrobione i wszystkie pody mogą się ze sobą łączyć pomiędzy nodami.

Przykładowo mamy poda, który się chce łączyć z innym podem, na którym jest web server. Tworzymy więc service, aby umożliwić połączenie. Elementy te mają adresy IP i hostname:



Zawsze, gdy tworzony jest service, Kube DNS server tworzy A record łącząc hostname service z jego adresem IP, więc wewnątrz clustra każdy pod może się do niego zwrócić po nazwie. W tym przypadku zakładamy, że oba pody są w tym samym namespace, więc niebieski pod może się połączyć z web serverem używając zwykłej nazwy service.

Jeśli jednak niebieski pod jest w innym namespace:



To przy połączeniu trzeba dodać na koniec nazwę namespace:

`web-service.apps`

Dla każdego namespace Kube DNS server tworzy subdomain. Wszystkie service są natomiast zgrupowane w inny subdomain - `svc`.

Podsumowując:

`web-service` - to nazwa service,

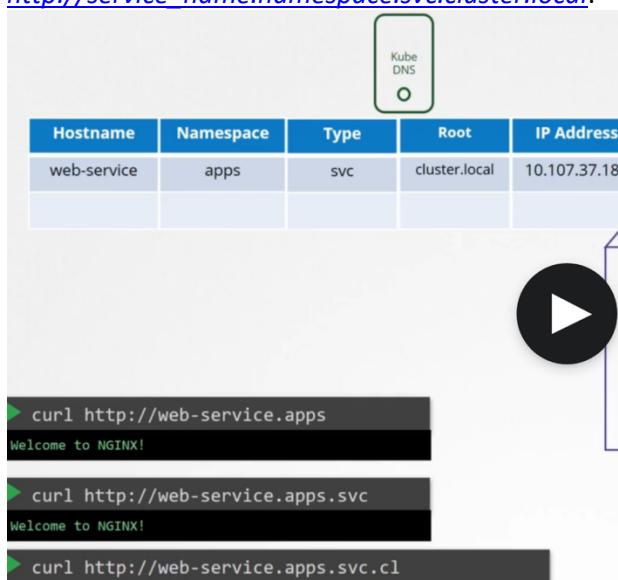
`apps` - nazwa namespace. Pamiętamy, że dla każdego namespace DNS server tworzy subdomain z nazwą namespace. Są tu więc zgrupowane wszystkie service i pody dla danego namespace,

`svc` - w tej subdomenie są zgrupowane wszystkie service,

`cluster.local` - jest to root domain, w której są zgrupowane wszystkie service i pody w clustrze.

Na tej podstawie, do każdego service można się łączyć poprzez:

http://service_name.namespace.svc.cluster.local:



Jest to FQDN dla service.

W przypadku podów, A record w DNS nie jest tworzony domyślnie, ale można to włączyć. Nie jest jednak używany wtedy hostname poda, ale Kubernetes tworzy nazwę na podstawie jego adresu IP zamieniając kropki na minusy: 10.244.2.5 -> 10-244-2-5. Na tej podstawie FQDN dla poda wygląda następująco:

http://pod_name.namespace.pod.cluster.local:

10-244-2-5	apps	pod	cluster.local	10.244.2.5
------------	------	-----	---------------	------------

CoreDNS in Kubernetes

Jeśli chcielibyśmy ręcznie ustawić komunikację pomiędzy podami, to najlepiej jest stworzyć DNS server i tam wpisywać nazwy podów z ich adresami IP, a następnie we wszystkich podach ustawić ten server w `/etc/resolv.conf`:

```
▶ cat >> /etc/resolv.conf
nameserver      10.96.0.10
```

W taki sposób Kubernetes automatycznie dodaje wpisy dla service, a dla podów tworzy ich nazwę z adresu IP zamieniając kropki na myślniki. Nazwa serwera DNS tworzonego w clustrze przez Kuberntesesa to CoreDNS (kube-DNS dla starszych wersji).

CoreDNS jest zdeployowany jako pod w kube-system namespace (najczęściej jest kilka podów jako replicaset). Uruchamia on binarkę Coredns (`./Coredns`), która używa `/etc/coredns/Corefile` jako plik konfiguracyjny (ścieżkę tego pliku można znaleźć robiąc `describe` na coredns deployment):

```
▶ cat /etc/coredns/Corefile
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        upstream
        fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    proxy . /etc/resolv.conf
    cache 30
    reload
}
```

W tym pliku jest wiele pluginów, służących do pokazywania zdrowia poda, monitorowania metryk, cache itp. Plugin kubernetes służy do komunikacji z Kuberntesem, jest tu ustawiona top level domain dla clustra - w tym przypadku `cluster.local` (oznacza to, że każdy wpis w CoreDNS będzie się zawierał w tej domenie). Ustawienie pods w pluginie odpowiada za tworzenie rekordów dla podów - pamiętamy, że jest to wyłączone domyślnie, ale tu można to włączyć. Każdy rekord, którego CoreDNS nie jest w stanie rozwiązać, jest przesyłany do serwera DNS ustawionego w `/etc/resolv.conf`. Corefile jest też przesyłany do poda jako ConfigMap object, dzięki temu jeśli chcemy modyfikować ustawienia, wystarczy modyfikować ConfigMap.

```
▶ kubectl get configmap -n kube-system
NAME          DATA   AGE
coredns       1      168d
```

Należy też pamiętać, że pody muszą wiedzieć, jak dostać się do CoreDNS. Gdy CoreDNS jest tworzony, automatycznie dodany jest też jego service, który udostępnia go podom. Domyślnie nazywa się on `kube-dns`:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP, 53/TCP

Adres IP tego serwisu jest automatycznie dodany jako DNS server w każdym podzie. Za ten automatyczny proces jest odpowiedzialny kubelet, w jego konfiguracji możemy zobaczyć ustawiony CoreDNS:

```
▶ cat /var/lib/kubelet/config.yaml
```

...
clusterDNS:
- 10.96.0.10
clusterDomain: cluster.local

Jeśli teraz byśmy próbowali wyświetlić informacje np. o service komendą `host` lub `nslookup` to wyświetli się FQDN service. Dzieje się tak dlatego, że do plików `/etc/resolv.conf` jest też automatycznie dodany search entry:

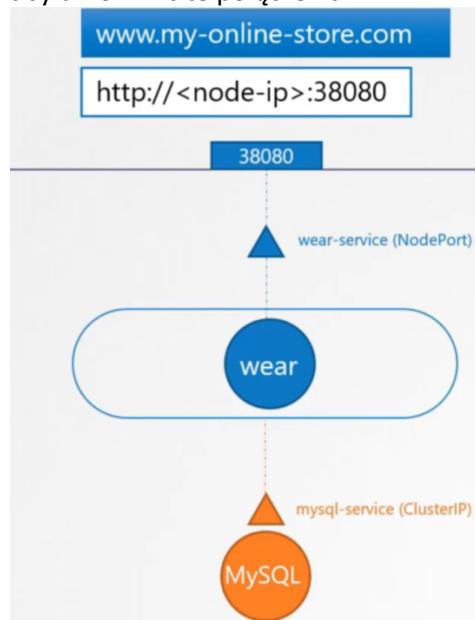
```
▶ cat /etc/resolv.conf
```

nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local

Pozwala to na znalezienie service używając dowolnej z tych nazw. Nie działa to jednak dla poda, dla poda trzeba używać wyłącznie FQDN.

Ingress

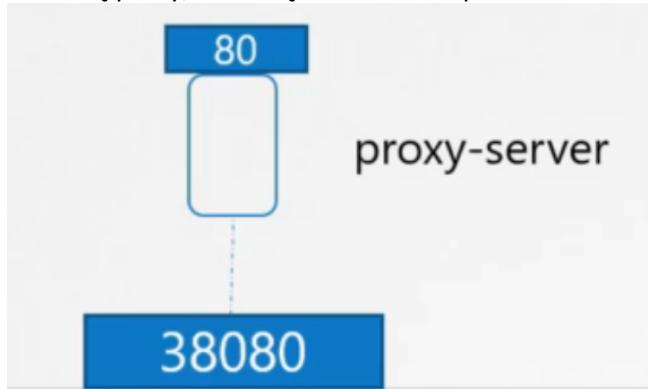
Różnica pomiędzy service i ingress. Założymy, że mamy aplikację w podzie i musi ona się łączyć z innym podem, w którym jest baza danych, oraz musi być dostępna z zewnątrz. Tworzymy więc dwa serwisy, aby umożliwić te połączenia:



Jeśli ruch wzrośnie w replicasecie możemy dołożyć dodatkowe pody.

Są jednak dodatkowe aspekty takiego połączenia, np. nie chcemy aby użytkownicy musieli pamiętać port 38080, tylko żeby mogli używać domyślnego 80. Niestety jednak nie jest to możliwe w service

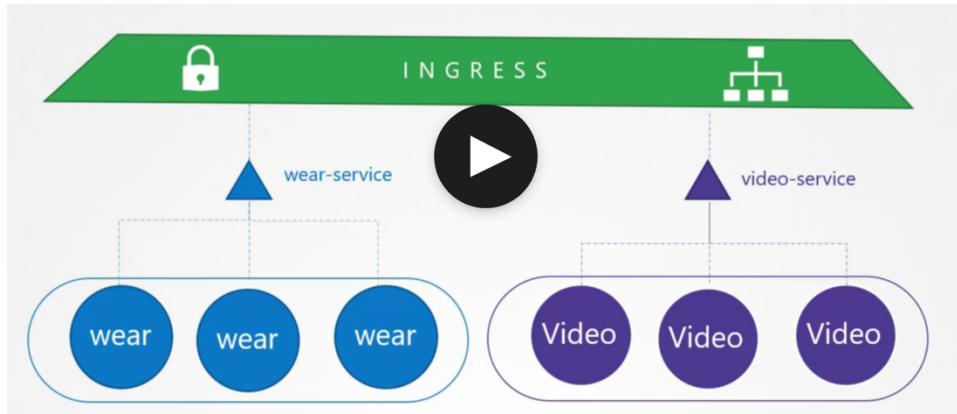
NodePort, bo może on tylko przypisać duże porty (powyżej 30000). Dodajemy więc dodatkową warstwę proxy, która będzie zmieniać port.



Na koniec trzeba skonfigurować DNS, aby指owało bezpośrednio do proxy servera. Aplikacja teraz będzie działać po linku z zewnątrz. Ta sama konfiguracja jest możliwa w Cloudzie, wtedy wybieramy typ service LoadBalancer i będzie on tworzył automatycznie Network Load Balancer w Cloudzie do zmieniania portu.

W przypadku jednak, gdyby aplikacja się rozwijała i doszłyby dodatkowe serwisy, dostępne pod innym linkiem, musielibyśmy dodawać dodatkowe resourcy w cloudzie, aby to wszystko obsłużyć. Dodatkowo wymagany byłby SSL. Gdybyśmy chcieli to wszystko mieć jednak wewnętrz Kuberntesu to trzeba użyć Ingress.

Ingress pozwala użytkownikom na dostęp do aplikacji poprzez jeden URL, a my możemy go skonfigurować, aby różne zapytania szły do różnych serwisów. W tym samym momencie SSL będzie też zaimplementowane. O Ingress można myśleć jak o Load Balancerze warstwy 7 zajmującym się rozdzielaniem ruchu:



Należy jednak pamiętać, że mając Ingress, trzeba też dodać jeden dodatkowy service (NodePort lub LoadBalancer), aby aplikacja była dostępna poza Kuberntesem.

Deployment Ingressu składa się z 2 kroków. Najpierw tworzymy deployment znanego proxy (GCE, Nginx, HAProxy, trafik, Istio), a potem robimy konfigurację. Krok pierwszy nazywany jest Ingress Controller, a konfiguracja Ingress Resources (tworzymy ją przy pomocy plików yaml).

Ingress Controller to tak naprawdę nie tylko proxy, ale ma też dodatkową inteligencję pozwalającą na monitorowanie resourców i plików yaml.

Ingress Controller (tu Nginx) tworzymy jako deployment składający się z kilku podów:

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-
                  controller/nginx-ingress-controller:0.21.0
      args:
        - /nginx-ingress-controller
        - --configmap=$(POD_NAMESPACE)/nginx-configuration
      env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      ports:
        - name: http
          containerPort: 80
        - name: https
          containerPort: 443

```

W momencie deploymentu, Nginx potrzebuje kilku argumentów, aby móc prawidłowo działać. Potrzebna jest np. ścieżka do nginx-ingress-controllera.

Dodatkowo, aby sam Nginx działał musimy mu ustawić ścieżki do logów, protokoły ssl i keep-alive probe. Tworzymy więc ConfigMap i ustawiamy go w argumentach, aby to zrealizować:

```

kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration

```

Należy też podać nazwy podów i namespace, gdzie będzie deployment. Na końcu trzeba sprecyzować porty, których Ingress Controller będzie używał. Są to 80 i 443.

Dodatkowo, aby wystawić Ingress poza sieć Kuberntesa tworzymy service typu NodePort i łączymy go labelką z Ingress Controllerem (wygenerowanie pliku yaml do service: `kubectl expose deployment ingress-controller --type=NodePort --port=80 --name=ingress --dry-run=client -o yaml > ingress.yaml`):

```

apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress

```

Ostatnią rzeczą jest stworzenie ServiceAccount dla Ingress Controllera z odpowiednimi rolami i role bindings. To konto będzie umożliwiało controllerowi monitorowanie Ingress resources (to jest ta wbudowana inteligencja Ingress Controllera).

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-ingress-serviceaccount

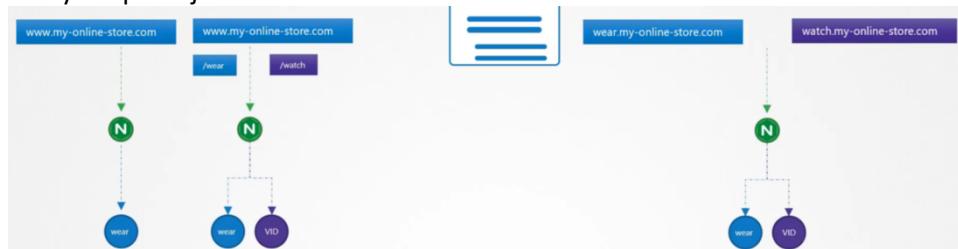
```

[Roles](#)

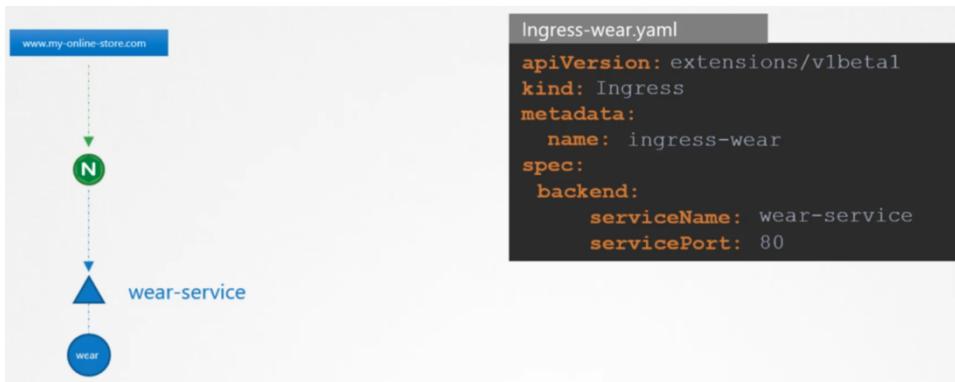
[ClusterRoles](#)

[RoleBindings](#)

Teraz tworzymy Ingress Resources. Jest to zbiór roli i konfiguracji ustawiony na Ingress Controllerze. Na podstawie tych ustawień przychodzący ruch będzie wysyłany w inne miejsca. Można go kierować ustawiając konkretny URL jako filtr, jakąś wartość po znaku slash, lub kierować różne domeny do różnych aplikacji:

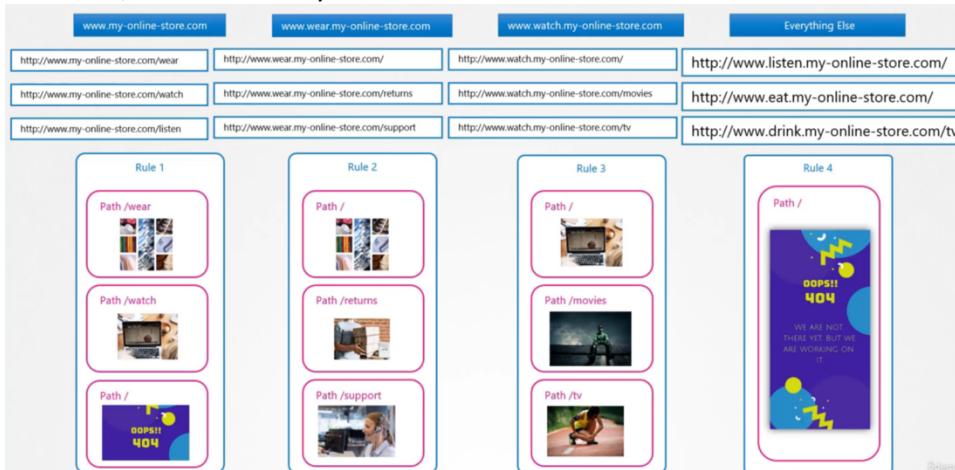


Każdy Ingress Resource jest tworzony plikiem yaml. W najprostszym przypadku kierowania ruchu z danego URL do aplikacji musimy w sekcji spec ustawić backend service:

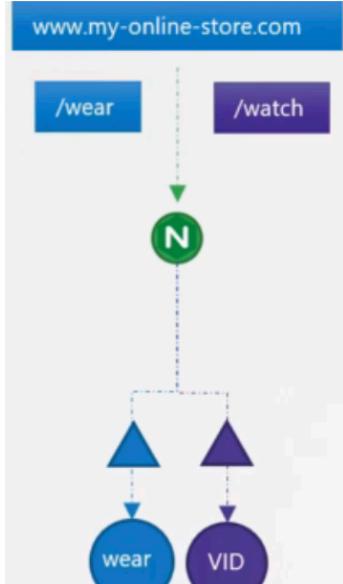


`kubectl get ingress` - wyświetli wszystkie Ingress Resources

Jeśli chcemy troche zmodyfikować ruch, musimy stworzyć więcej routing ruli. Założmy, że mamy 4 rule: 1 - my-online-store.com; 2- wear.my-online-store.com; 3- watch.my-online-store.com; 4- reszta. Dodatkowo w każdej ruli możemy ustawić różne ścieżki np. /wear będzie przesyłać do 2 linku, /watch do 3 linku, a inne do strony 404 Not Found. Tak samo można ustawić ścieżki w innych rulach:



Jeśli mamy rozróżnienie na różne ścieżki wewnętrz ruli to wszystko to też ustawimy w sekcji spec pliku yaml. Tu mamy jedną rulę, bo w tym przypadku zajmujemy się ruchem do my-online-store.com. Dla:



Będzie:

```

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
    paths:
    - path: /wear
      backend:
        serviceName: wear-service
        servicePort: 80
    - path: /watch
      backend:
        serviceName: watch-service
        servicePort: 80

```

kubectl describe ingress nazwa - pokaże więcej informacji o resource.

```

▶ kubectl describe ingress ingress-wear-watch
Name:           ingress-wear-watch
Namespace:      default
Address:
Default backend: default-http-backend:80 (<none>)
Rules:
Host  Path  Backends
*   /wear  wear-service:80 (<none>)
     /watch  watch-service:80 (<none>)
Annotations:
Events:
  Type  Reason  Age   From          Message
  Normal CREATE  14s  nginx-ingress-controller  Ingress default/ingress-wear-watch

```

Ważna informacja to default backend. Jest to URL, do którego zostanie przekierowany użytkownik, gdy będzie się próbował łączyć nieznanym linkiem, więc trzeba pamiętać, żeby tam coś ustawić i tą aplikację zdeployować.

Ostatni typ konfiguracji to przekierowania na podstawie domen:



Jako że mamy dwie domeny, w pliku yaml trzeba stworzyć dwie rule i ruch będzie rozdzielany przez pole host:

```

Ingress-wear-watch.yaml

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:

rules:
- host: wear.my-online-store.com
  http:
    paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
- host: watch.my-online-store.com
  http:
    paths:
      - backend:
          serviceName: watch-service
          servicePort: 80

```

Najnowsze zmiany w Ingress:



W nowych wersjach Ingress może być tworzony komendą:

Format - kubectl create ingress <ingress-name> --rule="host/path=service:port"

Example - kubectl create ingress ingress-test --rule="wear.my-online-store.com/wear=wear-service:80"*

Ingress Annotations and rewrite-target

Załóżmy, że aplikacja /watch wyświetla video streaming pod adresem <http://<watch-service>:<port>> a aplikacja /wear <http://<wear-service>:<port>>

Chcemy skonfigurować Ingress aby osiągnąć następujące przekierowanie:

<http://<ingress-service>:<ingress-port>/watch> --> <http://<watch-service>:<port>>

<http://<ingress-service>:<ingress-port>/wear> --> <http://<wear-service>:<port>>

Gdy użytkownik kliknie na URL po lewej będzie przekierowany na URL po prawej. Należy zauważyć, że aplikacje nie mają skonfigurowanego połączenia URL/path, a ścieżki /wear i /watch są skonfigurowane na Ingress Controllerze. Bez opcji rewrite-target byłby następujący wynik:

<http://<ingress-service>:<ingress-port>/watch> --> <http://<watch-service>:<port>/watch>

<http://<ingress-service>:<ingress-port>/wear> --> <http://<wear-service>:<port>/wear>

Aplikacje do których chcemy się łączyć mogą nie mieć skonfigurowanych path /wear i /watch, więc jako wynik dostalibyśmy 404 Not Found. Aby to naprawić, chcemy "nadpisać" URL, gdy request jest wysyłany do aplikacji wear i watch. Używamy do tego opcji rewrite-target, która zamienia to co jest w rules->http->paths->path (w przykładzie /pay) wartością w rewrite-target.

Przykładowy plik yaml:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  namespace: critical-space
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /pay
        backend:
          serviceName: pay-service
          servicePort: 8282
```

Tu /pay zamieniane jest na /

Jest też przykład z użyciem hosta:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
  name: rewrite
  namespace: default
spec:
  rules:
  - host: rewrite.bar.com
    http:
      paths:
      - backend:
          serviceName: http-svc
          servicePort: 80
        path: /something(/|$)(.*)
```

Tu `/something(/|$)(.*)` jest zamieniane na `/$2`

Design and Install Kubernetes Cluster

Design a Kubernetes Cluster

Pytania, które trzeba sobie zadać przed tworzeniem clustra to:

Cel clustra (edukacja - pojedynczy cluster stworzony przez kubeadm lub cloudowy provider, testowanie - cluster z kilkoma nodami i też tworzony przez kubeadm lub cloudowy provider, produkcja - high availability cluster z wieloma nodami stworzony przez kubeadm lub cloudowy provider z zapewnieniem dużej ilości resourców do stworzenia. W AWS lub GCP są tabelki z gotowymi rozmiarami clutrów w zależności od tego, ile ich potrzebujemy);

Czy będzie on hostowany w cloudzie czy on prem;

Jakiego rodzaju pracę będzie wykonywał - ile aplikacji i jakiego rodzaju, jakie wymagania sprzętowe podów i jak duży ruch będzie obsługiwany. Jeśli dużo operacji to warto wziąć SSD storage, jeśli dużo sieciowych połączeń, to network base storage, warto używać też persistent volumes dla wielu podów, Node Selectors i labelować dyski. Jeśli mamy bardzo dużego clustra to warto też rozważyć wyłączenie ETCD z Master noda i stworzenie dla niego osobnych nodów.

Choosing Kubernetes Infrastructure

Kubernetes może być zdeployowany na prywatnym laptopie, wirtualnych serwerach organizacji, lub w cloudzie.

Jeśli instalujemy na prywatnym sprzęcie to trzeba mieć Linuxa i ręcznie instalować binarki i ustawiać lokalny cluster. Jeśli mamy Windowsa, to trzeba na nim zainstalować jakiś VirtualBox żeby móc instalować Kubernetes na Linuxie. Żeby nie robić wszystkiego ręcznie możemy skorzystać z automatycznych tooli jak np. Minikube, który w szybki sposób tworzy mały pojedynczy cluster na VirtualBoxie i sam go konfiguruje. Można też użyć kubeadm, ale tutaj już trzeba mieć skonfigurowanego hosta Linuxowego np. na VM, pozwala on jednak mieć więcej niż jeden node.

Jeśli mamy cluster na on-premie, to sami musimy stworzyć VM i użyć różnych narzędzi i skryptów, aby konfigurować Kubernetes. Samemu trzeba jednak zarządzać VM i patchować je. Przykłady rozwiązań on-prem to OpenShift (jest to open source container application zainstalowany na Kubernetesie, pozwala też na łatwą integrację z CI/CD pipelines), Cloud Foundry Container Runtime (open source project pozwalający na tworzenie high available cluster przy pomocy toola Bosh), Vmware Cloud PKS, Vagrant. Przed uchuumieniem tych tooli, trzeba mieć jednak swoje VM odpowiednio skonfigurowane.

Jeśli chcemy cluster w cloudzie (managed-solution) to wtedy cloud jest odpowiedzialny za tworzenie i zarządzanie VM i sam konfiguruje, patchuje Kuberntesa. Przykłady to Google Container Engine (jest to kubernetes-as-a-service), OpenShift Online, Azure Kubernetes Service, Amazon Elastic Container Service for Kubernetes (EKS).

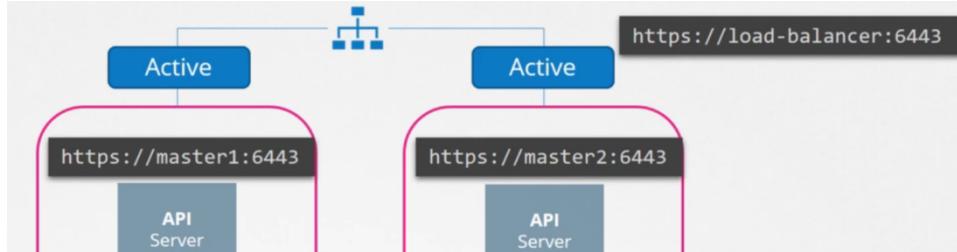
My w ćwiczeniach będziemy deployować na laptopie na VirtualBoxie. Będzie jeden master node i 2 worker nody.

Configure High Availability

Co się stanie gdy panie nam master node? Istniejące aplikacje będą dalej działać, ale gdy któryś kontener się wyłączy, to nie zostaną już stworzone nowe, dodatkowo nie będzie można się komunikować z clustrem przez kubectl, dlatego na prodzie dobrze jest mieć kilka master nodów, dzięki temu każdy komponent Kuberntesa będzie miał swój duplikat.

Jeśli mamy np. dwa master nody, to na każdym z nich pracują te same komponenty. Np. będą dwa Kube-api servery, które mogą pracować razem w trybie Active-Active. Wiemy, że kubectl łączy się z kube-api serverem, aby wykonać dane polecenia, a dokładne ustawienie tego, jak ma się łączyć jest w

pliku kube-config (przykładowo mamy <https://master1:6443>). Jeśli mamy dwa mastery, to najlepiej jest dodać load balancer (może to być Nginx, HAProxy itp.) i ustawić kubectl, tak by wysyłał tam polecenia, a load balancer będzie je rozdzielał pomiędzy nody.



W przypadku Controller Managera i Schedulera, nie mogą one pracować razem na dwóch masterach, bo mogłyby przez przypadek zduplikować dane polecenie. Są one w trybie Active-Standby.



Wybieranie, który z nich jest aktywny odbywa się przez leader election process. Np. gdy tworzymy Controller Manager, możemy nadać mu opcję --leader-elect, która domyślnie ma wartość true. Jak taki proces będzie już działał, będzie starał się zablokować inny kube-controller-manager object. Którykolwiek master node szybciej zaktualizuje informacje i zablokuje przeciwny proces, on stanie się aktywny, a drugi będzie w Standby. Następnie odbywa się tak zwany lease, czyli normalna praca, w której jeden jest Active, a drugi Passive, po upłynięciu czasu ustawionego w opcji --leader-elect-lease-duration (domyślnie 15 sekund). W międzyczasie jednak aktywny proces aktualnia swój status domyślnie dzięki opcji --leader-elect-renew-deadline (domyślnie 10 sekund). Dodatkowo każdy z procesów stara się zostać leaderem domyślnie co 2 sekundy przez opcję --leader-elect-retry-period. Dzięki temu, jeśli obecny master padnie, ten rezerwowy przejmie jego miejsce.

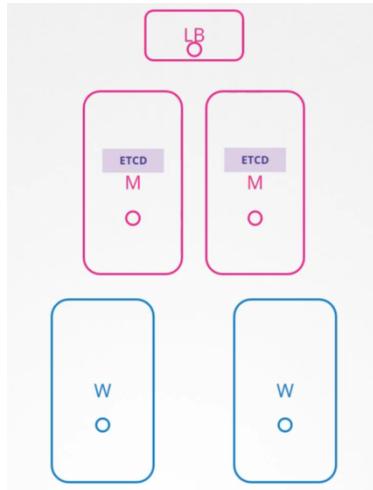
W przypadku ETCD są dwie opcje infrastruktury. Pierwsza - Stacked Topology, gdzie ETCD jest częścią istniejących master nodów. Jest to łatwiejsze w konfiguracji i potrzebuje mniej resurów, ale jak padnie master to ETCD razem z nim. Druga opcja to External ETCD Topology, gdzie ETCD jest na swoich własnych nodach. Jest to bezpieczniejsze, ale wymaga więcej pracy i serwerów. Pamiętamy, że jedynie kube-api server łączy się z ETCD, więc w jego konfiguracji trzeba ustawić, gdzie hostujemy ETCD:

```
cat /etc/systemd/system/kube-apiserver.service
```

```
[Service]
ExecStart=/usr/local/bin/kube-apiserver \
--advertise-address=${INTERNAL_IP} \
--allow-privileged=true \
--apiserver-count=3 \
--etcd-cafile=/var/lib/kubernetes/ca.pem \
--etcd-certfile=/var/lib/kubernetes/kubernetes.pem \
--etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \
--etcd-servers=https://10.240.0.10:2379,https://10.240.0.11:2379
```

W konfiguracji mamy podane dwa adresy jeśli ETCD jest na dwóch różnych nodach. Jest to dobre ustawienie, bo, każdy z nodów jest w stanie wykonać pracę zadaną dla ETCD.

Na podstawie tych informacji, możemy uaktualnić nasze zadanie i dodamy drugi node masterowy i load balancer do rozdzielania ruchu pomiędzy kube-api serverami.



ETCD in High Availability

ETCD jest to baza danych przechowująca dane o całym clustrze w postaci key-value. Informacje tutaj są przechowywane w postaci dokumentów. Każdy wpis ma swój dokument, w którym są klucze i wartości:

Key	Value
Name	John Doe
Age	45
Location	New York
Salary	5000

Dokumenty te są przechowywane w postaci plików, które mogą mieć różny format (yaml lub json) i zmiana jednego z nich nie wpływa na inne.

Najczęściej mamy kilka kopii ETCD na różnych serwerach, tak by w przypadku awarii jednej z nich, mieć cały czas inną. Pomimo kilku kopi, ETCD cały czas uaktualnia wszystkie z nich, aby w każdej chwili mieć na wszystkich takie same informacje, ponieważ kube-api server może się łączyć z każdą z nich. W praktyce działa to tak, że wybierany jest jeden node ETCD, który staje się leaderem i tylko on obsługuje operację write. Zapisuje on zmiany i przesyła je do innych nodów, które są followersami. Proces kończy się dopiero, jak dana informacja znajdzie się już na wszystkich nodach. Jeśli inny node dostanie zadanie write, to przesyła je on do lidera. Co w przypadku awarii jednego z nodów? Write zostanie uznane jako udane w przypadku, gdy lider będzie w stanie zapisać informację na większości (quorum) nodów (dlatego zalecane jest mieć przynajmniej 3 nody, bo jak jeden padnie to większością jest 2, a jak mamy 2 nody i jeden padnie to już nie ma większości i te operacje nie będą wykonane. Zawsze warto mieć nieparzystą liczbę nodów). Jak zepsute nody znowu wstaną, to oczywiście nowe dane też będą u nich zaktualizowane.

Aby umożliwić dystrybucję danych pomiędzy nody ETCD korzysta z protokołu RAFT. Protokół ten na początku wysyła losowy timer request (node ma odpowiedzieć po jakimś czasie) do wszystkich nodów i ten który pierwszy odpowie do wszystkich zostanie liderem. Następnie wysyła on co jakiś czas informacje do innych nodów, że dalej jest liderem, jeśli przestanie (np. jak padnie) to proces wyboru lidera zaczyna się od nowa.

Aby zainstalować ETCD należy pobrać binarkę, rozpakować ją, stworzyć strukturę folderów, skopiować certyfikaty dla ETCD:

```
▶ wget -q --https-only \
  "https://github.com/coreos/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz"

▶ tar -xvf etcd-v3.3.9-linux-amd64.tar.gz

▶ mv etcd-v3.3.9-linux-amd64/etcd* /usr/local/bin/

▶ mkdir -p /etc/etcd /var/lib/etcd

▶ cp ca.pem kubernetes-key.pem kubernetes.pem /etc/etcd/
```

Następnie skonfigurować etcd service. Ważną opcją jest initial-cluster peer, dzięki temu ETCD wie, że jest częścią clustra i gdzie inne nody są.

```
etcd.service
ExecStart=/usr/local/bin/etcd \
--name ${ETCD_NAME} \
--cert-file=/etc/etcd/kubernetes.pem \
--key-file=/etc/etcd/kubernetes-key.pem \
--peer-cert-file=/etc/etcd/kubernetes.pem \
--peer-key-file=/etc/etcd/kubernetes-key.pem \
--trusted-ca-file=/etc/etcd/ca.pem \
--peer-trusted-ca-file=/etc/etcd/ca.pem \
--peer-client-cert-auth \
--client-cert-auth \
--initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \
--listen-peer-urls https://${INTERNAL_IP}:2380 \
--listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://${INTERNAL_IP}:2379 \
--initial-cluster-token etcd-cluster-0 \
--initial-cluster peer-1=https://${PEER1_IP}:2380,peer-2=https://${PEER2_IP}:2380 \
--initial-cluster-state new \
--data-dir=/var/lib/etcd
```

Następnie można używać etcdctl do komunikacji i przechowywania dancyh. Są dwie wersje jego API (2-domyślna i 3). Ważne żeby ustawić zmienną środowiskową, która to określi:

```
▶ export ETCDCTL_API=3
```

etcdctl put key value - dodanie informacji do bazy danych

etcdctl get key - wyświetlenie danych

etcdctl get --prefix --keys-only - wyświetlenie wszystkich kluczy.

Install Kubernetes with Kubeadm

Introduction to Deployment with kubeadm

Kubeadm tool pozwala na automatyczną instalację wszystkich komponentów Kuberntesa, ustawienie ich konfiguracji, certyfikatów itp. Na początku musimy mieć wirtualne maszyny (przyszłe nody), na których skonfigurujemy Kubernetes cluster. Trzeba będzie też wybrać jednego z nich jako mastera, a inne jako worker nody. Drugim krokiem jest instalacja toola do konteneryzacji na wszystkich hostach (np. Docker). Następnie instalujemy kubeadm tool na wszystkich nodach. Potem inicjalizujemy master node i instalujemy jego komponenty. Trzeba też wypełnić wymagania sieciowe - stworzyć sieć do komunikacji pomiędzy masterem i worker nodami (Pod network). Na końcu przyłączamy worker nody.

Dokumentacja: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/>

Provision VMs with Vagrant

Przed rozpoczęciem trzeba mieć zainstalowane VirtualBox i Vagrant. Użyjemy pliku Vagrant do automatyzacji tworzenia master noda i 2 workerów. Jest on stworzony tutaj: [GitHub - kodekloudhub/certified-kubernetes-administrator-course: Certified Kubernetes Administrator - CKA Course](#)

Klonujemy to repozytorium do siebie: *git clone URL*

Następnie otwieramy skopiowane foldery. Możemy sprawdzić Vagrantfile, czy ma odpowiednie ustawienia nodów, zakres adresów IP. Komendą *vagrant status* możemy sprawdzić status naszych VM (oczywiście na razie ich nie ma jeszcze). Teraz komendą *vagrant up* uruchamiamy tworzenie VM. Po stworzeniu VM możemy się na nie zalogować przez *vagrant ssh nazwa_VM*. Warto sprawdzić, czy wszystkie działają.

Deployment with Kubeadm

Po stworzeniu VM możemy pójść do dokumentacji kubeadm i sprawdzić, czy mamy wszystkie wymagania spełnione (odpowiedni system, 2GB ramu na każdej maszynie, 2CPU lub więcej, połączenie sieciowe pomiędzy wszystkimi maszynami w clustrze i odpowiednie porty pootwierane, hostname, MAC address dla każdego noda itp).

Dodatkowo, aby iptables widział bridged traffic musimy mieć załadowany br_netfilter module na każdym node: *sudo modprobe br_netfilter*.

Następnie uruchamiamy na każdym node gotowe, podane w dokumentacji komendy, aby ustawić parametry w kernelu:

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sudo sysctl --system
```

Następnie instalujemy Dockera i jego komponenty i repozytorium na wszystkich nodach zgodnie z instrukcją. Do tego będą potrzebne root privileges. Trzeba też zainstalować Docker daemon i stworzyć katalog dla serwisu Dockera. Na końcu restartujemy Dockera po zainstalowaniu. Aby sprawdzić, czy działa uruchamiamy: *systemctl status docker.service*.

Teraz można przystąpić do instalacji kubeadm, kubelet i kubectl na wszystkich nodach.

Teraz możemy przystąpić do kreacji clustra przy pomocy kubeadm. Na początku inicjujemy controlplane node (pierwszy krok jest potrzebny, gdy tworzymy HA master node, tutaj mamy tylko pojedynczy). W kolejnym kroku wybieramy pod network add-on i jaki zakres IP będzie przydzielony podom. Jest tu wiele opcji do wyboru takich jak Calico, Cilium, Weave Net, Kube-router itp. Trzeci krok

możemy też pominąć, bo używamy Dockera, a w czwartym trzeba sprecyzować adres IP kube-api servera jako api server advertize address. Jest to ustawienie stałego IP, na którym kube-api server będzie nasłuchiwał ruchu z innych nodów (najczęściej jest to static IP master node - możemy go sprawdzić komendą `inconfig enp0s8`).

Po ustawieniu tych parametrów uruchamiamy `kubeadm init` na master node wraz z tymi parametrami:

```
(root@kubemaster) ~ # kubeadm init --pod-network-cidr 10.244.0.0/16 --apiserver-advertise-address=192.168.56.2
```

Po uruchomieniu kubeadm będzie instalował certyfikaty i wszystkie komponenty.

Po instalacji trzeba przystąpić do post installation tasks. Output kubeadm podpowie nam jeszcze jakie komendy mamy uruchomić jako normalny użytkownik (nie root) - jest to między innymi kopiowanie pliku admin.conf.

Następnie trzeba zdeployować pod network w clustrze (tu używamy WeaveNet) - jak to zrobimy będzie można dołączyć inne nody komenda podana w outpcie:

Then you can join any number of worker nodes by running the following on each as root:

```
kubeadm join 192.168.56.2:6443 --token 51...zsu8haydiaqekji \
    --discovery-token-ca-cert-hash sha256:1c297ce7b60d33494589059744332349a87d42acde2a
e5b0d67cdaf936b11a8
```

Idziemy do pod network w dokumentacji i wybieramy networking solution. Następnie kopujemy komende, która wszystko skonfiguruje (uruchamiamy ją tylko na masterze).

Potem dołączamy worker nody komendą z outputu.

Teraz jak uruchomimy `kubectl get nodes` to zobaczymy wszystkie nody, po jakimś czasie nody będą ready i będzie można testować ich działanie np. tworząc pody.

Dokumentacja: [Installing kubeadm | Kubernetes](#)

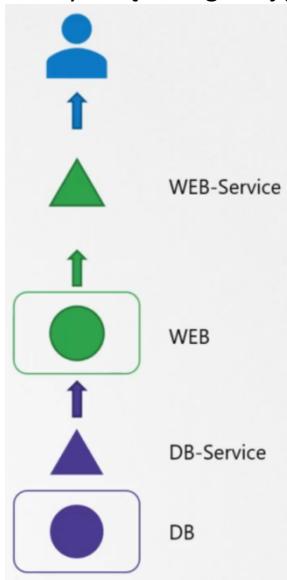
Jeśli chcemy zainstalować konkretną wersję kubeadm, kubelet i kubectl to trzeba to sprecyzować w komendzie:

```
sudo apt-get install -y kubelet=1.21.0-00 kubeadm=1.21.0-00 kubectl=1.21.0-00
```

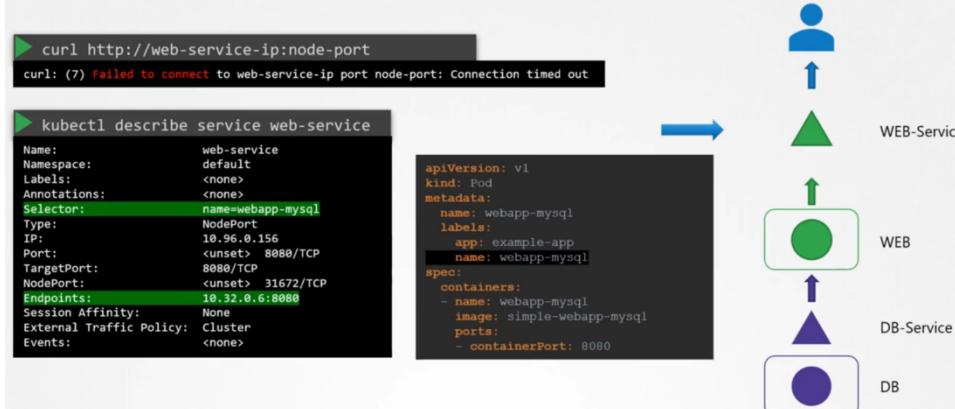
Troubleshooting

Application failure

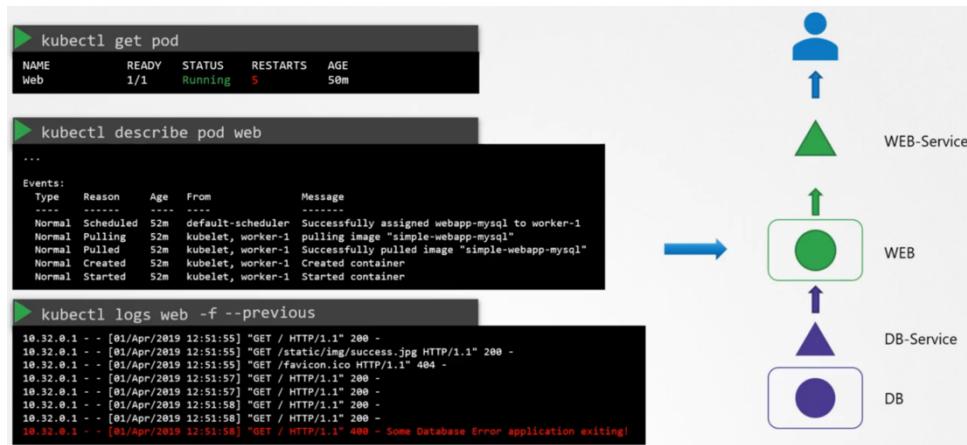
Mamy taką konfigurację podów:



Przykładowo użytkownicy mówią, że aplikacja webowa nie jest dostępna. Dobrze jest sprawdzić, czy web service jest dostępny pod IP Node-Port service przy pomocy curla. Następnie sprawdzić konfigurację service, czy ma dobrze skonfigurowane endpointy dla web poda. Warto też sprawdzić selectory ustawione na service i na podzie, czy się zgadzają:



Następnie dobrze sprawdzić, czy pod w ogóle działa, czy się restartuje, sprawdzić jego eventy i logi. Jeśli aplikacja się restartuje i błąd jest na końcu tuż przed restartem i znika, to dobrze jest oglądać logi przy pomocy flagi -f, albo użyć opcji --previous żeby zobaczyć logi poprzedniego poda:



Na końcu w ten sam sposób warto sprawdzić db service i db poda.

Control Plane Failure

Warto sprawdzić statusy nodów i podów w clustrze, szczególnie warto sprawdzić, czy wszystkie pody w kube-system namespace działają (jeśli mieliśmy deployment z kubeadm). Jeśli jednak komponenty są zdeployowane jako serwisy, to trzeba też sprawdzić ich statusy:

```

► service kube-apiserver status
● kube-apiserver.service - Kubernetes API Server
  Loaded: loaded (/etc/systemd/system/kube-apiserver.service; enabled; vendor preset: enabled)
  Active: active (running) since Wed 2019-03-20 07:57:25 UTC; 1 weeks 1 days ago
    Docs: https://github.com/kubernetes/kubernetes
 Main PID: 15767 (kube-apiserver)
   Tasks: 13 (limit: 2362)

► service kube-controller-manager status
● kube-controller-manager.service - Kubernetes Controller Manager
  Loaded: loaded (/etc/systemd/system/kube-controller-manager.service; enabled; vendor preset: enabled)
  Active: active (running) since Wed 2019-03-20 07:57:25 UTC; 1 weeks 1 days ago
    Docs: https://github.com/kubernetes/kubernetes
 Main PID: 15771 (kube-controller)
   Tasks: 10 (limit: 2362)

► service kube-scheduler status
● kube-scheduler.service - Kubernetes Scheduler
  Loaded: loaded (/etc/systemd/system/kube-scheduler.service; enabled; vendor preset: enabled)
  Active: active (running) since Fri 2019-03-29 01:45:32 UTC; 11min ago
    Docs: https://github.com/kubernetes/kubernetes
 Main PID: 28390 (kube-scheduler)
   Tasks: 10 (limit: 2362)

► service kubelet status
● kubelet.service - Kubernetes Kubelet
  Loaded: loaded (/etc/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Active: active (running) since Wed 2019-03-20 14:22:06 UTC; 1 weeks 1 days ago
    Docs: https://github.com/kubernetes/kubernetes
 Main PID: 1281 (kubelet)
   Tasks: 24 (limit: 1152)

► service kube-proxy status
● kube-proxy.service - Kubernetes Kube Proxy
  Loaded: loaded (/etc/systemd/system/kube-proxy.service; enabled; vendor preset: enabled)
  Active: active (running) since Wed 2019-03-20 14:21:54 UTC; 1 weeks 1 days ago
    Docs: https://github.com/kubernetes/kubernetes
 Main PID: 794 (kube-proxy)
   Tasks: 7 (limit: 1152)

```

Następnie dobrze też sprawdzić logi komponentów controlplane (logów z podów jeśli mamy kubeadm, lub logów systemowych jeśli są to serwisy).

```
▶ kubectl logs kube-apiserver-master -n kube-system
I0401 13:45:38.190735      1 server.go:703] external host was not specified, using 172.17.0.117
I0401 13:45:38.194290      1 server.go:145] Version: v1.11.3
I0401 13:45:38.819705      1 plugins.go:158] Loaded 8 mutating admission controller(s) successfully in the following order:
Namespacelifecycle,LimitRanger,ServiceAccount,NodeRestriction,Priority,DefaultTolerationSeconds,DefaultStorageClass,MutatingAdmissionWebhook.
I0401 13:45:38.819741      1 plugins.go:161] Loaded 6 validating admission controller(s) successfully in the following order:
LimitRanger,ServiceAccount,Priority,PersistentVolumeClaimResize,ValidatingAdmissionWebhook,ResourceQuota.
I0401 13:45:38.821372      1 plugins.go:158] Loaded 8 mutating admission controller(s) successfully in the following order:
Namespacelifecycle,LimitRanger,ServiceAccount,NodeRestriction,Priority,DefaultTolerationSeconds,DefaultStorageClass,MutatingAdmissionWebhook.
I0401 13:45:38.821410      1 plugins.go:161] Loaded 6 validating admission controller(s) successfully in the following order:
LimitRanger,ServiceAccount,Priority,PersistentVolumeClaimResize,ValidatingAdmissionWebhook,ResourceQuota.
I0401 13:45:38.985453      1 master.go:234] Using reconciler: lease
W0401 13:45:40.900380      1 genericapiserver.go:319] Skipping API batch/v2alpha1 because it has no resources.
W0401 13:45:41.376077      1 genericapiserver.go:319] Skipping API rbac.authorization.k8s.io/v1alpha1 because it has no resources.
W0401 13:45:41.381736      1 genericapiserver.go:319] Skipping API scheduling.k8s.io/v1alpha1 because it has no resources.
```

```
▶ sudo journalctl -u kube-apiserver
Mar 20 07:57:25 master-1 systemd[1]: Started Kubernetes API Server.
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.553377 15767 flags.go:33] FLAG: --address="127.0.0.1"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558273 15767 flags.go:33] FLAG: --admission-control="[]"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558325 15767 flags.go:33] FLAG: --admission-control-config-file=""
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558339 15767 flags.go:33] FLAG: --advertise-address="192.168.5.11"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558353 15767 flags.go:33] FLAG: --allow-privileged="true"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558365 15767 flags.go:33] FLAG: --alsologtostderr="false"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558413 15767 flags.go:33] FLAG: --anonymous-auth="true"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558425 15767 flags.go:33] FLAG: --api-audiences="[]"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558442 15767 flags.go:33] FLAG: --apiserver-count="3"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558454 15767 flags.go:33] FLAG: --audit-dynamic-configuration="false"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558464 15767 flags.go:33] FLAG: --audit-log-batch-buffer-size="10000"
Mar 20 07:57:25 master-1 kube-apiserver[15767]: [0320 07:57:25.558474 15767 flags.go:33] FLAG: --audit-log-batch-max-size="1"
```

Worker Node Failure

Sprawdzamy statusy nodów, jeśli jakiś jest NotReady, to sprawdzamy go dokładnie komendą describe:

```
▶ kubectl get nodes
NAME     STATUS   ROLES   AGE   VERSION
worker-1  Ready   <none>  8d    v1.13.0
worker-2  NotReady <none>  8d    v1.13.0

▶ kubectl describe node worker-1
...
Conditions:
Type        Status  LastHeartbeatTime          Reason           Message
----        ----   ----                  ----          -----
OutOfDisk  False   Mon, 01 Apr 2019 14:30:33 +0000  KubeletHasSufficientDisk  kubelet has sufficient disk space available
MemoryPressure  False   Mon, 01 Apr 2019 14:30:33 +0000  KubeletHasSufficientMemory  kubelet has sufficient memory available
DiskPressure  False   Mon, 01 Apr 2019 14:30:33 +0000  KubeletHasNoDiskPressure  kubelet has no disk pressure
PIDPressure  False   Mon, 01 Apr 2019 14:30:33 +0000  KubeletHasSufficientPID  kubelet has sufficient PID available
Ready       True    Mon, 01 Apr 2019 14:30:33 +0000  KubeletReady            kubelet is posting ready status. AppArmor enabled
```

Każdy node ma conditions, mówią one o tym, z czym node ma problem np. OutOfDisk, MemoryPressure, DiskPressure lub PIDPressure (za dużo procesów). Kolumna LastHeartBeat mówi nam, kiedy ostatni raz to działało.

Dobrze jest też sprawdzić CPU, dysk noda komendami top i df -h.

```
▶ top
top - 14:43:56 up 3 days, 19:02, 1 user, load average: 0.35, 0.29, 0.21
Tasks: 112 total, 1 running, 72 sleeping, 0 stopped, 0 zombie
%Cpu(s): 3.9 us, 1.7 sy, 0.1 ni, 94.3 id, 0.0 wa, 0.0 hi, 0.1 si, 0.0 st
KiB Mem : 1009112 total, 74144 free, 736608 used, 198360 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 129244 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  34 root      20   0      0      0      0 S  5.9  0.0   0:13.14 kswapd0
28826 root      20   0 1361320 383208  3596 S  5.9 38.0   0:46.95 mysqld
  1 root      20   0    78260   5924  3192 S  0.0  0.6   0:21.88 systemd
  2 root      20   0      0      0      0 S  0.0  0.0   0:00.02 kthreadd
  4 root      20  -20     0      0      0 I  0.0  0.0   0:00.00 kworker/0:0H
```

```
▶ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            481M   0  481M  0% /dev
tmpfs           99M  1000K  98M  1% /run
/dev/sda1       9.7G  5.3G  4.5G  55% /
tmpfs           493M   0  493M  0% /dev/shm
tmpfs           5.0M   0  5.0M  0% /run/lock
tmpfs           493M   0  493M  0% /sys/fs/cgroup
tmpfs           99M   0  99M  0% /run/user/1000
```

Warto też sprawdzić status i logi kubeleta (poda lub systemowe). Dodatkowo warto sprawdzić certyfikaty kubeleta, czy nie są wygaśnięte, czy są w odpowiedniej grupie i czy są wydane przez CA:

```
openssl x509 -in /var/lib/kubelet/worker-1.crt -text

Certificate:
Data:
    Version: 3 (0x2)
    Serial Number:
        ff:e0:23:9d:fc:78:03:35
Signature Algorithm: sha256WithRSAEncryption
Issuer: CN = KUBERNETES-CA
Validity
    Not Before: Mar 20 08:09:29 2019 GMT
    Not After : Apr 19 08:09:29 2019 GMT
Subject: CN = system:node:worker-1, O = system:nodes
Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    Public-Key: (2048 bit)
        Modulus:
            00:b4:28:0c:60:71:41:06:14:46:d9:97:58:2d:fe:
            a9:c7:6d:51:cd:1c:98:b9:5e:e6:e4:02:d3:e3:71:
            58:a1:60:fe:cb:e7:9b:4b:86:04:67:b5:4f:da:d6:
            6c:08:3f:57:e9:70:59:57:48:6a:ce:e5:d4:f3:6e:
            b2:fa:8a:18:7e:21:60:35:8f:44:f7:a9:39:57:16:
            4f:4e:1e:b1:a3:77:32:c2:ef:d1:38:b4:82:20:8f:
            11:0e:79:c4:d1:9b:f6:82:c4:08:84:84:68:d5:c3:
            e2:15:a0:ce:23:3c:8d:9c:b8:dd:fc:3a:cd:42:ae:
            5e:1b:80:2d:1b:e5:5d:1b:c1:fb:be:a3:9e:82:ff:
```

Sprawdzanie logów i uruchamianie serwisów systemowych:

<code>systemctl</code>	<code>status</code>	<code>kubelet</code>
<code>systemctl</code>	<code>start</code>	<code>kubelet</code>

`journalctl -u kubelet`

Należy pamiętać, że kubelet łączy się z controlplane po porcie 6443.

Network Troubleshooting

Kubernetes używa pluginów CNI do skonfigurowania sieci, a za uruchomienie tych pluginów jest odpowiedzialny kubelet, dlatego też w jego konfiguracji ustawiamy te dwa parametry:

-cni-bin-dir - tu kubelet szuka pluginów do uruchomienia

-network-plugin - tu precyzujemy, którego pluginu ma użyć

Jest wiele dostępnych pluginów:

-Weave Net - jedyny opisany w dokumentacji, instalacja: `kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"`

-Flannel - instalacja: `kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/2140ac876ef134e0ed5af15c65e414cf26827915/Documentation/kube-flannel.yml`

-Calico - instalacja: `curl https://docs.projectcalico.org/manifests/calico.yaml -O` i potem `kubectl apply -f calico.yaml`.

Kubernetes używa też CoreDNS, który służy jako DNS dla całego clustra. W dużych clustrach użycie pamięci przez CoreDNS wzrasta przez liczbę podów i service. Dodatkowymi czynnikami wpływającymi na zużycie są rozmiar DNS answer cache oraz liczba poleceń otrzymanych przez CoreDNS nainstancję (rate of queries received).

Kubernetes ma następujące resourcy dla CoreDNS:

-service account - coredns

-cluster-roles - coredns i kube-dns

-clusterrolebindings - coredns i kube-dns

```
-deployment - coredns  
-configmap - coredns  
-service - kube-dns
```

Ważną konfiguracją dla CoreDNS jest Corefile plugin, zdefiniowany jako *configmap*. Port 53 jest używany do DNS resolution.

Tak wygląda ustawienie głównej domeny dla clustra, tutaj cluster.local:

```
kubernetes cluster.local in-addr.arpa ip6.arpa {  
    pods insecure  
    fallthrough in-addr.arpa ip6.arpa  
    ttl 30  
}
```

Troubleshooting problemów z CoreDNS

Jeśli pody coredns są w stanie pending, to warto sprawdzić, czy network plugin jest zainstalowany. Jeśli mają status CrashLoopBackOff lub Error, to warto sprawdzić, czy mamy na nodach SELinux ze starszą wersją Dockera. Jeśli tak, to trzeba zupgradować Dockera do nowej wersji, wyłączyć SELinux, zmodyfikować coredns deployment i tam ustawić allowPrivilegeEscalation=ture

```
kubectl -n kube-system get deployment coredns -o yaml | sed 's/allowPrivilegeEscalation: false/allowPrivilegeEscalation: true/g' | kubectl apply -f -
```

Jeśli CoreDNS pod i kube-dns service działają dobrze, trzeba sprawdzić, czy kube-dns ma dobre endpointy: *kubectl -n kube-system get ep kube-dns*. Może tu być pusto, lub coś złego ustawione.

Kube-proxy to sieciowe proxy działające na każdym nodzie w clustrze. Zajmuje się ona podtrzymywaniem network ruli w nodach. Pozwalają one na komunikację pomiędzy podami, a serwisami z wewnętrz lub zewnętrz clustra. Kube-proxy możemy znaleźć jako deamonset z podem na każdym clustrze. Kube-proxy jest też odpowiedzialna za monitorowanie serviców i endpointów połączonych z każdym z service. Jak ktoś będzie się chciał połączyć z service, to kube-proxy przesyła to polecenie do poda będącego za service.

Jeśli uruchomimy *kubectl describe ds kube-proxy -n kube-system* zobaczymy, że binarka kube-proxy ma następujące komendy wewnątrz kontenera kube-proxy:

Command:

```
/usr/local/bin/kube-proxy  
--config=/var/lib/kube-proxy/config.conf  
--hostname-override=$(NODE_NAME)
```

Pobiera ona konfigurację z pliku konfiguracyjnego i możemy nadpisać hostname nazwą noda, na którym działa pod. W pliku konfiguracyjnym są też zdefiniowane clusterCIDR, kube-proxy mode, ipvs, iptables, bindaddress, kube-config itp.

Troubleshooting związany z Kube-proxy

Sprawdzić, czy kube-proxy pod działa. Warto też sprawdzić jego logi.

Sprawdzić, czy configmap jest dobrze zdefiniowany i czy jest dobrze ustawiony dla binarki w pliku konfiguracyjnym.

Sprawdzić, czy kube-config jest ustawiony w pliku konfiguracyjnym.

Sprawdzić, czy kube-proxy działa na clustrze:

```
# netstat -plan | grep kube-proxy
```

tcp	0	0 0.0.0.0:30081	0.0.0.0:*	LISTEN	1/kube-
tcp	0	0 127.0.0.1:10249	0.0.0.0:*	LISTEN	1/kube-
tcp	0	0 172.17.0.12:33706	172.17.0.12:6443		ESTABLISHED
1/kube-proxy					
tcp6	0	0 :::10256	:::*	LISTEN	1/kube-
proxy					

Zadanie:

W pierwszym zadaniu nie było CNI pluginia zainstalowanego i przez to w każdym service nie było endpointa ustawionego. Wszystkie pody były running, ale połączenia brakowało. Łatwo zauważać ten brak CNI, bo nie było poda weave w kube-system namespace.

W drugim zadaniu pod kube-proxy był zcrashowany. W jego logach był błąd, że nie może on znaleźć pliku konfiguracyjnego. Trzeba porównać ten plik ustawiony w configmap i w daemonsecie. Okazało się,

że zła ścieżka była w daemonsecie.
kubectl get configmap -n kube-system
kubectl get daemonset -n kube-system

Other Topics

Introduction to YAML

Jest to format, w którym mamy często pliki konfiguracyjne. Przykładowa reprezentacja danych:

Key Value Pair	Array / Lists	Dictionary / Map
Fruit: Apple Vegetable: Carrot Liquid: Water Meat: Chicken	Fruits: - Orange - Apple - Banana Vegetables: - Carrot - Cauliflower - Tomato	Banana: Calories: 105 Fat: 0.4 g Carbs: 27 g Grapes: Calories: 62 Fat: 0.3 g Carbs: 16 g

Należy pamiętać, że w Key/Value pair zawsze potrzebna jest spacja po dwukropku. W przypadku listy poszczególne wartości precyzujemy od myślników. W słowniku natomiast musimy mieć takie same wcięcia do wszystkich wartości będących na tym samym poziomie. Jeśli coś jest bardziej wcięte od poprzedniej linijki, to znaczy, że jest to określeniem, wartością należącą do poprzedniej linijki.

Możemy mieć też połączone struktury danych np. lista słowników:

Key Value / Dictionary / Lists
Fruits: - Banana: Calories: 105 Fat: 0.4 g Carbs: 27 g - Grape: Calories: 62 Fat: 0.3 g Carbs: 16 g

Jeśli mamy wiele cech danego obiektu to najlepiej je wszystkie umieścić w słowniku. Jeśli te cechy mają swoje podcechy, to będziemy mieć słownik w słowniku.

Jeśli natomiast mamy wiele obiektów tego samego typu to najlepiej je przechowywać w liście. Jeśli będziemy chcieli dodać cechy poszczególnych aut to wyjdzie nam lista słowników.

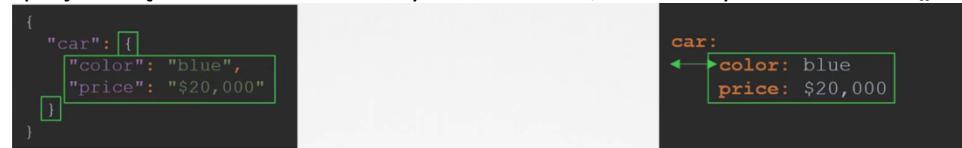
Należy pamiętać, że kolejność obiektów w słowniku nie ma znaczenia, natomiast kolejność obiektów w liście jest istotna.

Dictionary / Map		Dictionary / Map
Banana: Calories: 105 Fat: 0.4 g Carbs: 27 g	=	Banana: Calories: 105 Carbs: 27 g Fat: 0.4 g
Array / List	≠	Array / List
Fruits: - Orange - Apple - Banana	≠	Fruits: - Orange - Banana - Apple

Dodatkowo każda linijka zaczynająca się od # jest uznawana jako komentarz.

Introduction to JSON PATH

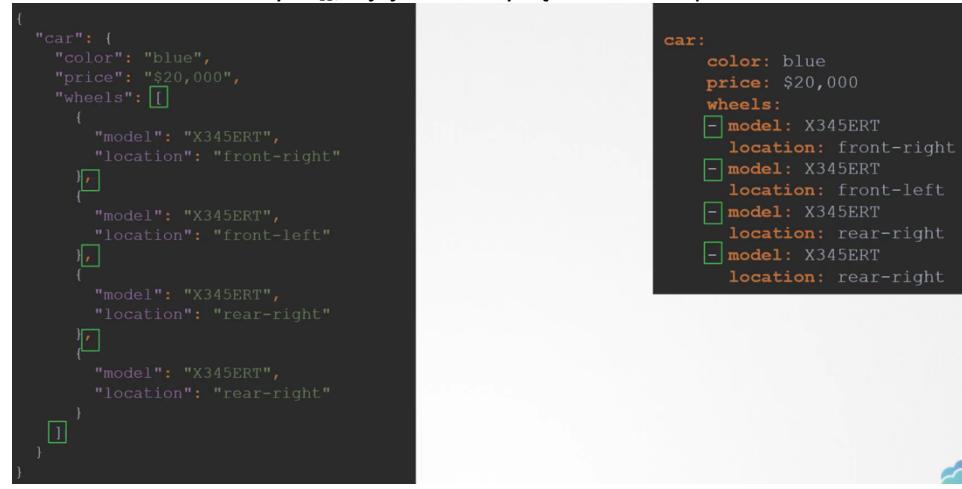
Yaml i JSON służą do tego samego, czyli do reprezentowania danych. Różnice są takie, że yaml używa spacji i wcięć do rozróżnienia danych w słowniku, a JSON używa nawiasów i {}.



```
{  
    "car": [  
        {"color": "blue",  
         "price": "$20,000"}  
    ]  
}
```

```
car:  
  color: blue  
  price: $20,000
```

Do wyróżnienia obiektów na liście w yaml używamy myślników, a w JSON lista jest oznaczona nawiasem kwadratowym [], a jej elementy są oddzielone przecinkami.



```
{  
    "car": {  
        "color": "blue",  
        "price": "$20,000",  
        "wheels": [  
            {  
                "model": "X345ERT",  
                "location": "front-right"  
            },  
            {  
                "model": "X345ERT",  
                "location": "front-left"  
            },  
            {  
                "model": "X345ERT",  
                "location": "rear-right"  
            },  
            {  
                "model": "X345ERT",  
                "location": "rear-right"  
            }  
        ]  
    }  
}
```

```
car:  
  color: blue  
  price: $20,000  
  wheels:  
    - model: X345ERT  
      location: front-right  
    - model: X345ERT  
      location: front-left  
    - model: X345ERT  
      location: rear-right  
    - model: X345ERT  
      location: rear-right
```

Istnieje wiele programów do konwertowania plików z formatu json na yaml.

JSON PATH to język zapytań do wyświetlania danych zapisanych w postaci json. Należy napisać odpowiednie query, aby dostać odpowiednie dane. Wpisanie nazwy np. obiektu ze słownika wyświetli jego dane, a wpisanie nazwa.cecha wyświetli wartość danej cechy tego obiektu. Należy pamiętać jeszcze o tzw root elemencie pliku json. Jak nie mamy podanej nazwy - tutaj w pierwszej linijce jest tylko { to musimy się odnieść do root elementu poprzez dodanie \$ przed każdym query:

QUERY	RESULT
<pre>[\$.car]</pre>	<pre>{ "color": "blue", "price": "\$20,000" }</pre>
<pre>[\$.bus]</pre>	<pre>{ "color": "white", "price": "\$120,000" }</pre>
<pre>\$.car.color</pre>	<pre>"blue"</pre>
<pre>\$.bus.price</pre>	<pre>"\$120,000"</pre>

Wyświetlony wynik będzie nawiasach kwadratowych [].

W przypadku listy (oznaczonej []), gdy chcemy wyświetlić dany element odnosimy się do jego pozycji zaczynając od 0 np. [1]. Znowu należy pamiętać o \$ jako root element. Jeśli chcemy wyświetlić kilka obiektów, to oddzielamy je przecinkiem w nawiasie kwadratowym.

QUERY	RESULT
<pre>{ "car": { "color": "blue", "price": "\$20,000" }, "bus": { "color": "white", "price": "\$120,000" } }</pre>	<p>Get car details \$.car</p> <pre>{ "color": "blue", "price": "\$20,000" }</pre> <p>Get bus details \$.bus</p> <pre>{ "color": "white", "price": "\$120,000" }</pre> <p>Get car's color \$.car.color</p> <pre>"blue"</pre> <p>Get bus's price \$.bus.price</p> <pre>">\$120,000"</pre>

Na tej podstawie możemy łączyć słowniki i listy w jednym query. Należy pamiętać, że kropka jest dla słownika a nawias kwadratowy dla listy:

DATA	QUERY	RESULT
<pre>{ "car": { "color": "blue", "price": "\$20,000", "wheels": [{ "model": "X345ERT", "location": "front-right" }, { "model": "X346GRX", "location": "front-left" }, { "model": "X236DEM", "location": "rear-right" }, { "model": "X987XMV", "location": "rear-right" }] } }</pre>	<p>Get the model of the 2nd wheel \$.car.wheels[1].model</p>	<pre>"X346GRX"</pre>

Jeśli chcemy dodać jakieś kryterium do naszego query, np. z listy liczb chcemy wyświetlić tylko te, które są większe niż 40. Dajemy wtedy wewnątrz nawiasu kwadratowego ?(), które odpowiada za operator if. Aby odwołać się do sprawdzania każdego elementu po kolei używamy znaku @ dodając tutaj >40.

DATA	QUERY	RESULT
<pre>[12, 43, 23, 12, 56, 43, 93, 32, 45, 63, 27, 8, 78]</pre>	<p>Get all numbers greater than 40 \$[Check if each item in the array > 40]</p> <p>Check if => ? ()</p> <p>\$[?(each item in the list > 40)]</p> <p>each item in the list => @</p> <p>\$[?(@ > 40)]</p>	<pre>[43, 56, 43, 93, 45, 63, 78]</pre>

W ten sam sposób można używać innych kryteriów np. @==40, @!=40, @in[40,43,45]. Na tej podstawie możemy też napisać query do poprzedniego zadania z samochodem:

```
$.car.wheels[?( @.location == "rear-right") ].model
```

JSON PATH - Wild Cards

Wild Card * pomaga nam wyświetlić wszystkie wartości danej cechy dla wszystkich obiektów. Mamy przykład słownika z dwoma obiektami: car i bus. Jeśli chcemy wyświetlić kolory ich obu użyjemy właśnie Wild Cardu (\$. *.color):

DATA	QUERY	RESULT
<pre>{ "car": { "color": "blue", "price": "\$20,000" }, "bus": { "color": "white", "price": "\$120,000" } }</pre>	Get car's color \$.car.color	["blue"]
	Get bus's color \$.bus.color	["white"]
	Get all colors \$.*.color	["blue", "white"]

"*" oznacza wszystkie obiekty wewnętrz s³ownika.

W przypadku listy, gdzie mamy ponumerowane obiekty te¶ mo¿emy wyświetlić np. warto¶ci modelu wszystkich kó³:

DATA	QUERY	RESULT
<pre>[{ "model": "X345ERT", "location": "front-right" }, { "model": "X346ERT", "location": "front-left" }, { "model": "X347ERT", "location": "rear-right" }, { "model": "X348ERT", "location": "rear-right" }]</pre>	Get 1st wheel's model \${[0]}.model	["X345ERT"]
	Get 4th wheel's model \${[3]}.model	["X348ERT"]
	Get all wheels' model \${[*]}.model	["X345ERT", "X346ERT", "X347ERT", "X348ERT"]

Mo¿emy te¶ łączyæ ten Wild Card w s³ownikach i listach jednocze¶nie uzyskujac np. wszystkie modele ze wszystkich pojazdów:

DATA	QUERY	RESULT
<pre>{ "car": { "color": "blue", "price": 20000, "wheels": [{ "model": "X345ERT" }, { "model": "X346ERT" }] }, "bus": { "color": "white", "price": 120000, "wheels": [{ "model": "Z227KLJ" }, { "model": "Z226KLJ" }] } }</pre>	Get car's 1st wheel model \$.car.wheels[0].model	["X345ERT"]
	Get car's all wheel model \$.car.wheels[*].model	["X345ERT", "X346ERT"]
	Get bus's wheel models \$.bus.wheels[*].model	["Z227KLJ", "Z226KLJ"]
	Get all wheels' models \$.*.wheels[*].model	["X345ERT", "X346ERT", "Z227KLJ", "Z226KLJ"]

JSON PATH - Lists

Gdy w li¶cie chcemy wyświetlić kilka elementów to oddzielamy je przecinkiem, ale mo¿emy te¶ wyświetlić ciąg elementów np. od pierwszego do czwartego: \${[0:4]} w tym przypadku zostanie

wyświetlone pierwsze cztery elementy (należy pamiętać, że element 4 jest piąty w kolejności, ale on już nie będzie wyświetlony).

Jeśli podamy w query `$[0:8:2]` to będziemy mieli wyświetlany co drugi element od pierwszego do ósmego. Dodatkowa liczba po drugim dwukropku to tzw skok.

Jeśli chcemy wyświetlić ostatni element z listy to użyjemy `$[-1]`. Gdybyśmy chcieli wyświetlić przedostatni to zamienimy na `-2` itd. Ta postać nie działa jednak w każdej odsłonie JSON PATH, czasami trzeba to sprecyzować jako `$[-1:0]`

Jeśli chcielibyśmy np. wyświetlić trzy ostatnie elementy to można dodać `$[-3:0]`.

JSON PATH - Kubernetes

JSON PATH jest używany w Kubernetesie, gdy potrzebujemy obejrzeć dane setek resourców np. na prodzie mamy 100 podów i chcemy zobaczyć na raz ich status, potem wyfiltrować tylko nie działające itp. Używając kubectl i JSON PATH osiągniemy to w prosty sposób.

Kubectl, gdy chce uzyskać jakieś informacje, pyta o nie kube-api server, który zwraca je w postaci .json. Kubectl następnie zamienia je w czytelny format. W związku z tą zamianą, wiele informacji jest ukrytych i można użyć flagi `-o wide`, aby je pokazać, ale cały czas nie wyświetli to informacji np. o taints, image, CPU itp. Takie informacje możemy uzyskać tylko komendą `descirbe`, lub właśnie używając JSON PATH.

Aby użyć JSON PATH w Kubernetesie musimy:

1. Znaleźć komendę kubernetesową, które pokaże nam żądane informacje (np. `kubectl get nodes`, `pods` itp.)
2. Dodać flagę `-o json`, aby mieć output w JSON
3. Stworzyć JSON PATH query.
4. Dodać JSON PATH query do zapytania kubectl.

Przykładowo:

```
{  
    "apiVersion": "v1",  
    "kind": "List",  
    "items": [  
        {  
            "apiVersion": "v1",  
            "kind": "Pod",  
            "metadata": {  
                "name": "nginx-5557945897-gznjp",  
            },  
            "spec": {  
                "containers": [  
                    {  
                        "image": "nginx:alpine",  
                        "name": "nginx"  
                    }  
                ],  
                "nodeName": "node01"  
            }  
        }  
    ]  
}
```

`kubectl get pods -o=jsonpath='{.items[0].spec.containers[0].image}'`

Należy pamiętać, że tu nie musimy dodawać \$ jako symbolu roota, bo kubectl dodaje to automatycznie.

Można też łączyć ze sobą kilka query w jednym zapytaniu (\n to oznaczenie nowej lini - entera, \t to tab, czyli wcięcie):

```
▶ kubectl get nodes -o=jsonpath='{.items[*].metadata.name} {"\n"} {.items[*].status.capacity.cpu}'  
master node01  
4 4
```

Aby inaczej zmodyfikować wynik możemy też użyć pętli:

```
▶ kubectl get nodes -o=jsonpath=  
'{range .items[*]}{.metadata.name} {"\t"} {.status.capacity.cpu}{"\n"}{end}'
```

{range.items[*]} to początek pętli mówiący, że będzie iteracja przez wszystkie itemy. W środku mamy komendę dla każdego itemu i na koniec {end}.

Ten sam wynik można też uzyskać opcją custom columns dla kubectl. Wzór to *kubectl get nodes -o=custom-columns=nazwa_kolumny:json_path*

Przykładowo taki będzie output dla wyświetlenia nodów w kolumnie:

```
▶ kubectl get nodes -o=custom-columns=NODE:.metadata.name  
NODE  
master  
node01
```

Aby uzyskać wynik z poprzedniej metody, musimy dodać drugą kolumnę:

```
▶ kubectl get nodes -o=custom-columns=NODE:.metadata.name,CPU:.status.capacity.cpu  
NODE CPU  
master 4  
node01 4
```

JSON PATH może być też użyte do sortowania outputu (jest to flaga --sort-by). Przykładowo tu mamy output kubectl get nodes posortowany po nazwie:

```
▶ kubectl get nodes --sort-by=.metadata.name  
NAME STATUS ROLES AGE VERSION  
master Ready master 5m v1.11.3  
node01 Ready <none> 5m v1.11.3
```

Test Notes

Gdy chcemy aby kontener miał określone przywileje systemowe po stworzeniu to dodajemy securityContext->capabilities

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: super-user-pod
    name: super-user-pod
spec:
  containers:
    - image: busybox:1.28
      name: super-user-pod
      command: ["sleep"]
      args: ["4800"]
    securityContext:
      capabilities:
        add: ["SYS_TIME"]
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Tworzenie Usera

Najpierw na podstawie pliku.csr tworzymy CertificateSigningRequest. Aby otrzymać wartość pola request uruchamiamy: `cat myuser.csr | base64 | tr -d "\n"`. Należy chyba dodać też te grupy i usages.

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: john-developer
spec:
  request:
    LS0tLS1CRUDJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUIJQ1ZEQ0NBVhdDQVFb0R6RU5NQXNHQTFVRUF3d0VhbTlvYmpDQ0FTSXdEUVIKS29aSWh2Y05BUUVCQIFBRApnZ0VQQURDQ0FRb0NnZ0VCQU5LQmVEVE56aUxSdUt3YWhiSEtVb3BKOEFKbU1ZVm1CSElxN0tPT1FZOC9HTzg2CktHcVIDaE5JZUtLK2RvMjRBbVMxNGY3UGJKdTIVQXI5TDNLT0Q5TIBwZzgzcVltV2s2NkZwM0lvQjFDTHk0eVQKNnZxMGhBM2pVWIE5ZUw3Rzdxb21oalAxRmNFVHVkUGFJQnNGN2MxNmVKM2FWOFpPY1VadklwZm5jZDZhNWloegowTm5Wa3FIS0JLNWRYY0FDa08yaHA0ZTZLa2xTeUNmbWdMbmdjcXord2F6cnhRNXBvcUdiaEJTNlJOdnpWb0hxCmwzcGZjU052bVFkcWUwVXEzek9BQ0dBRENEbEFSdk56WWZlQzNDNnhWWkN4Vkd0VnVERkx5Q1dPNUi5L0Y4M04KYTi2QWdnQlRkWklybGl2djZoR29yOUN2ZVI5T25XaERwYVRoRkY4QOF3RUFBYUFBTUEwR0NTcUdTSWIzRFFFQgpDd1VBQTRJQkFRQ01oSzEzWWZJYzc4VjNGdVU2RjRISU9USW5vSHAxNzhvUXdESWxzQVYvbkwyeWUrYnNNZ2tpCk1pQTIXZjFrVnpvM05qQ20yVDdnQ3bJFpaa2FqSSNQ1NZNytTVHpEbXIRNDFHZUlrvWN6QRndms3ckJ1emoKMkdWL0w2dzMzUkJwekd3clpMZUk1SlRSXQ3cWtxUEhoaVFXSmI0b3JkMGpkQzRxLzdxeTQweVVWVURFbFILYwp5QnRySDUzTFI0akg0M1VZSJFmYUpPVWt5NktsWIFIvxZmdGIETjZ2YWwxZkxIU2FGc04zaVFoOVJ1WFhqbGYxCkJmU3RGNkRobERjbkl1eW9hMEw4SCtiZ0VLTEIHbHJLc3JbjVoQzFCb1krYWxyYTJrN1k0QXBvUnlKemQwWFICO DcwSTF1MzJvajhNQ1E1dGIBZVR1MTQwenVGQ3VrdHcKLS0tLS1FTkQgQ0VSVEIGSUNBVEUgUkVRVU VTVC0tLS0tCg==
  signerName: kubernetes.io/kube-apiserver-client
  usages:
```

- client auth
- digital signature
- key encipherment
- groups:
- system:authenticated

Następnie akceptujemy ten request: `kubectl certificate approve myuser`. Możemy też go wyświetlić przez: `kubectl get csr`. Teraz możemy stworzyć rolę i role binding dla nowego usera (pamiętać o namespace!):

```
kubectl create role developer --resource=pods --verb=create,list,get,update,delete --
namespace=development
kubectl create rolebinding developer-role-binding --role=developer --user=john --
namespace=development
```

Ostatecznie można też sprawdzić, czy ma określone przywileje:

```
kubectl auth can-i update pods --as=john --namespace=development
```

[Certificate Signing Requests | Kubernetes](#)

Testowanie połączenia do poda i service przy użyciu danego image

Najpierw tworzymy poda i jego service:

```
kubectl run nginx-resolver --image=nginx
```

```
kubectl expose pod nginx-resolver --name=nginx-resolver-service --port=80 --target-port=80 --type=ClusterIP
```

Do testowania połączenia tworzymy poda, ale z flagą --rm, czyli że będzie on usunięty od razu po teście. Po dwóch myślnikach -- wpisujemy komendę do uruchomienia:

```
kubectl run test-nslookup --image=busybox:1.28 --rm -it --restart=Never -- nslookup nginx-resolver-service
```

To samo robimy z podem, tylko trzeba znaleźć jego IP:

```
kubectl get pod nginx-resolver -o wide
```

```
kubectl run test-nslookup --image=busybox:1.28 --rm -it --restart=Never -- nslookup ip_poda
```

[Create static Pods | Kubernetes](#)

Upgradowanie wersji deploymentu do nowszej i zapisywanie tych zmian

Najpierw tworzymy deployment:

```
kubectl run nginx-deploy --image=nginx:1.16 --replicas=1 --record (flaga record powoduje, że będziemy zapisywac zmiany)
```

`Kubectl rollout history deployment nginx-deploy` pokaże nam wszystkie zmiany.

```
Kubectl set image deployment/nazwa_deploymentu container=nowa_wersja_image - ta komenda zmieni nam image na nowy, w naszym przypadku będzie to: kubectl set image deployment/nginx-deploy nginx-deploy=nginx:1.17 --record
```

Service Account

Pody autentykują się do API Servera przy pomocy ServiceAccount i jeśli nie sprecyzujemy go to domyślne SA dla namespace jest używane w momencie tworzenia podów.

Tworzenie SA:

```
kubectl create serviceaccount pvviewer
```

Tworzenie clusterrole z odpowiednimi privilege:

```
kubectl create clusterrole pvviewer-role --resource=persistentvolumes --verb=list
```

Tworzenie clusterrolebinding:

```
kubectl create clusterrolebinding pvviewer-role-binding --clusterrole=pvviewer-role --serviceaccount=default:pvviewer
```

Plik yaml na tworzenie poda:

```
apiVersion:v1
kind:Pod
metadata:
labels:
  run: pvviewer
  name: pvviewer
spec:
containers:
-image: redis
  name: pvviewer
  serviceAccountName: pvviewer
```

Plik yaml na tworzenie poda z kilkoma kontenerami:

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-pod
spec:
  containers:
    -image: nginx
      name: alpha
      env:
        -name: name
        value: alpha
    -image: busybox
      name: beta
      command: ["sleep", "4800"]
      env:
        -name: name
        value:beta
```

Robienie Network Policy gdzie jest ingress dla wszystkich połączeń (nie ma tu żadnych ustawień from:)

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ingress-to-nptest
  namespace: default
spec:
  podSelector:
    matchLabels:
      run: np-test-1
  policyTypes:
    - Ingress
  ingress:
    - ports:
        - protocol: TCP
          port: 80
```

```
Master $ kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="ExternalIP")].address}'
```