Указание к заданию 4. Docker Swarm: Масштабируемое приложение на базе кластера Docker Swarm

Цель: изучить методы развертывания масштабируемых приложений на основе кластеров платформы Docker Swarm.

Критерии оценивания

Необходимо установить и развернуть Docker, Docker Compose на 3-х узлах в публичном облаке. Необходимо инициализировать создание Docker Swarm, создать 1 головной узел и присоединить к нему 2 рабочих узла.
Необходимо создать тестовую службу nginx с 2 репликами на разных узлах. Показать работу и доступность службы во внешней сети.
Необходимо создать и развернуть в кластере swarm службу приватного репозитория. Продемонстрировать ее доступность с узлов кластера. Необходимо создать простейшее демонстрационное приложение на Pyhon + Flask. Загрузить его в приватный репозиторий и продемонстрировать его развертывание в кластере. Показать возможности масштабирования сервиса.
Разработать и показать масштабируемость собственного веб-приложения, обеспечивающего обработку CSV-файлов и реализующего их асинхронную (отложенную) обработку. Веб-сервис должен предоставлять веб-API, обеспечивающий загрузку в сервис таблицы в виде CSV-файла для ее обработки и получения результата в виде N элементов из данного файла с максимальным значением значения данных по определенному полю данного CSV-файла (TOP-N). Первая строка CSV-файла содержит заголовки соответствующих столбцов таблицы. В качестве примера можно использовать данные со страницы https://www3.epa.gov/otaq/tcldata.htm (например, https://www3.epa.gov/fueleconomy/testcars/database/16tstcar.csv).

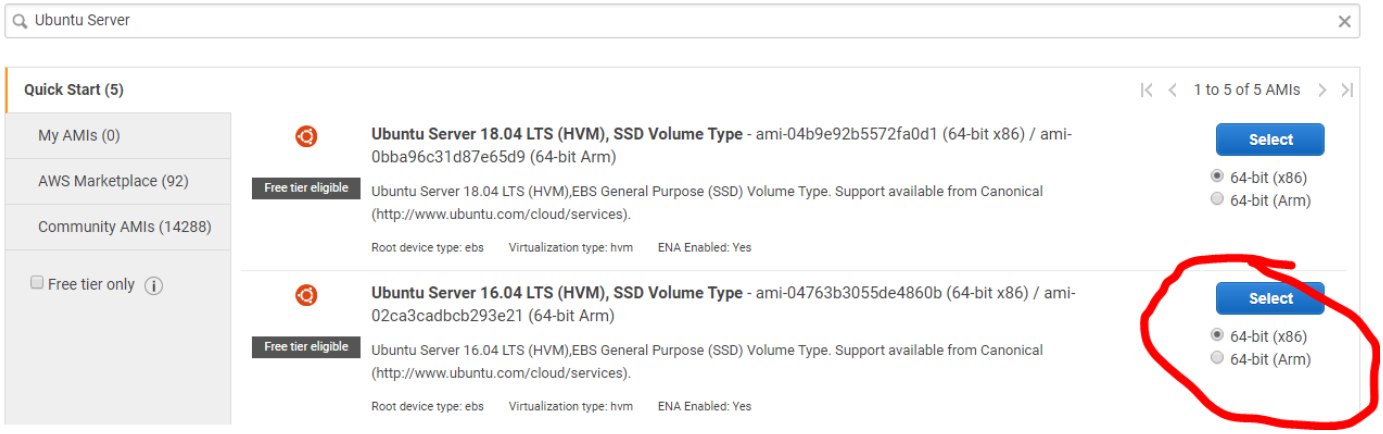
1.1 Развертывание Docker, Docker Compose и Docker Swarm

1.1.1 Начало работы

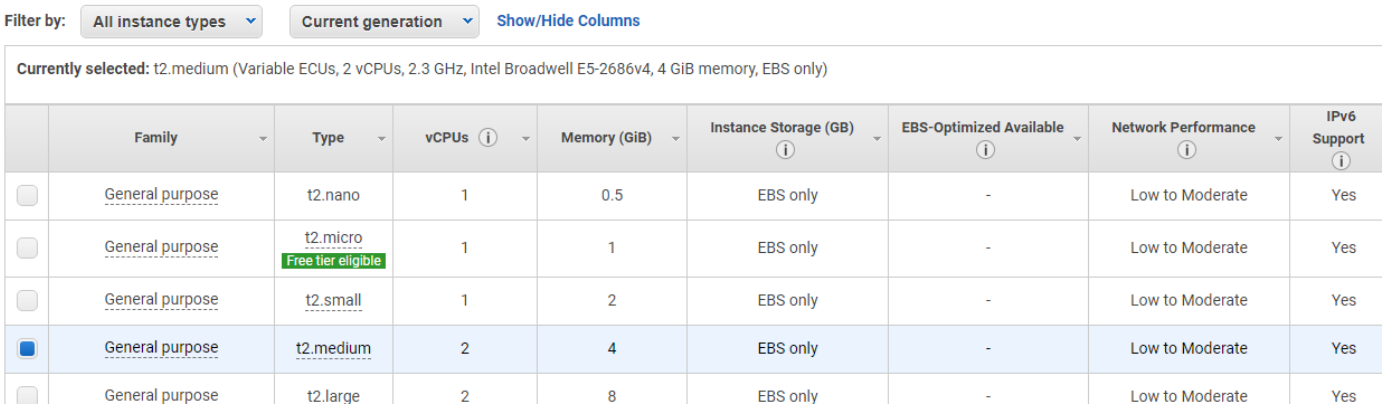
Установка Docker Swarm с Docker Compose на Ubuntu 16.04. В рамках данной лабораторной работы мы изучим, каким образом вы можете использовать технологии Docker Compose и Docker Swarm для развертывания многоконтейнерных приложений на базе облачных ресурсов, например, на базе Amason Web Services. В рамках этой лабораторной работы мы будем использовать образ Ubuntu ec2. Для начала

необходимо создать 3 новых виртуальных машины, которые и будут составлять наш кластер.

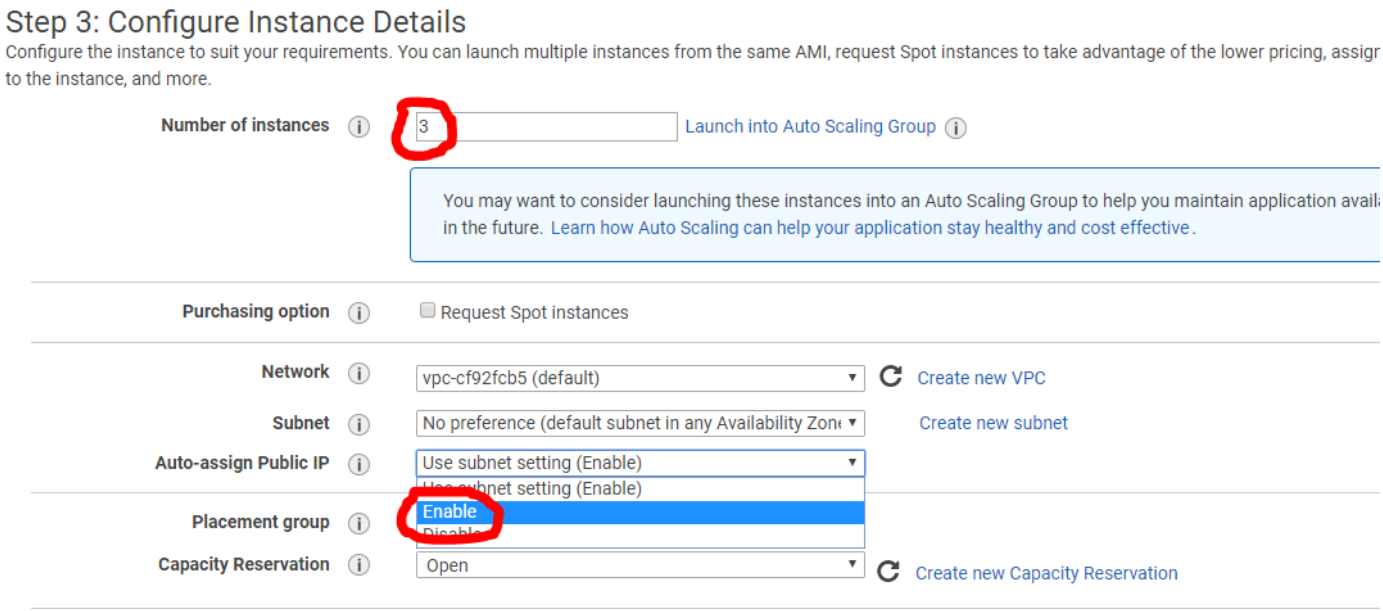
Создайте 3 машины с предустановленной Ubuntu Server 16.04.



Для работы необходимо использовать 3 машины класса t2.medium



Выбираем кнопку «Next: Configure Instance Details» и указываем, что мы хотим создать 3 машины, и отдельно в поле “Auto-assign Public IP” выбираем “Enable”.

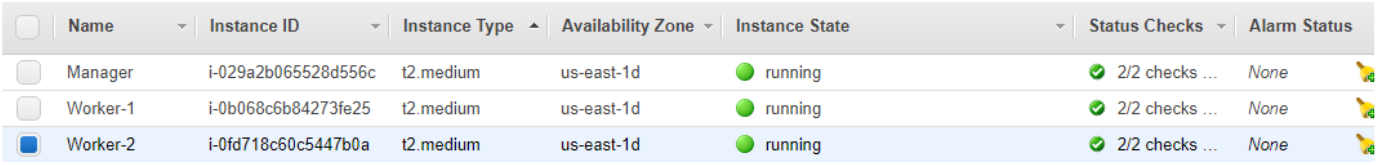


После этого нажимаем “Review and Launch”.

Не забудьте создать отдельный ключ для вновь-созданных образов. После того, как загрузите ключ на локальную машину, не забудьте сделать установку корректных прав на приватный ключ (предположим, что вы его назвали docker-swarm.pem)

```
$ chmod 400 docker-swarm.pem
```

После того, как машины будут созданы, дайте им имена в системе управления образами. Одну назовем Manager, другие Woker-1 и Worker-2 соответственно.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
Manager	i-029a2b065528d556c	t2.medium	us-east-1d	running	2/2 checks ...	None
Worker-1	i-0b068c6b84273fe25	t2.medium	us-east-1d	running	2/2 checks ...	None
Worker-2	i-0fd718c60c5447b0a	t2.medium	us-east-1d	running	2/2 checks ...	None

1.1.2 Установка Docker и Docker Compose

На машинах, которые вы создали не установлен Docker. Зайдем на машину Manager посредством терминала. Вместо «ManagerPublicIP» необходимо указать публичный адрес машины Manager в AWS.

```
$ ssh -i "docker-swarm.pem" ubuntu@ManagerPublicIP
```

Не забудьте указать имя пользователя «ubuntu» при подключении к узлу.

Для установки Docker выполним команды:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
$  sudo apt-get update
$  sudo apt-cache policy docker-ce
$  sudo apt-get install -y docker-ce
$  sudo systemctl status docker
```

Для выхода из окна информации нажмите q. Далее, для корректной работы с Docker нам необходимо прописать пользователя ubuntu в группу docker:

```
$  sudo usermod -a -G docker ubuntu
```

Для того, чтобы изменения вступили в силу, необходимо перезагрузить машину:

```
$  sudo reboot
```

После того, как машина перезагрузится, мы сможем зайти на нее и проверить корректность работы Docker:

```
$  docker info
```

Установим Docker Compose на узел на основе инструкций, представленных на сайте https://docs.docker.com/compose/install/

```
$  sudo curl -L "https://github.com/docker/compose/releases/download/1.24.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
$  sudo chmod +x /usr/local/bin/docker-compose
```

После успешной установки Docker Compose проверим его работу:

```
$ docker-compose --version
```

После успешной установки Docker и Docker Compose на машине Manager, необходимо повторить аналогичные процедуры, установив Docker и Docker Compose на машины Worker-1 и Worker-2.

1.1.3 Обновление настроек группы безопасности

Теперь нам нужно обновить группу безопасности, в которой находятся наши узлы. Мы должны обеспечить им возможность взаимодействия по служебным портам, по которым обеспечивается внутренняя коммуникация между узлами в сети Docker Swarm. Также, мы должны открыть порты для просмотра наших тестовых веб-приложений. Если правила будут указаны не корректно, то сеть overlay network, которая будет создаваться поверх Docker Swarm не сможет корректно транслировать запросы между узлами кластера. Создайте новую группу безопасности, и добавьте туда следующие группы правил:

1) Все входные TCP соединения по портам 22, 80, 5000, 8001 и 8080 должны приниматься с любых внешних адресов
2) Входные TCP соединения по портам 2376, 2377 должны приниматься с внутренних (приватных) IP-адресов всех машин, находящихся в кластере
3) Входные TCP **и UDP** соединения по портам 4789, 7946 должны приниматься с внутренних (приватных) IP-адресов всех машин, находящихся в кластере

Для примера, привожу структуру правил, что получились у меня в кластере.

Type	Protocol	Port Range	Source	Description
Custom TCP Rule	TCP	2377	172.31.45.50/32	Worker-1
Custom TCP Rule	TCP	2377	172.31.44.151/32	Worker-2
HTTP	TCP	80	0.0.0.0/0	
HTTP	TCP	80	::/0	
Custom TCP Rule	TCP	8001	0.0.0.0/0	
Custom TCP Rule	TCP	8001	::/0	
Custom TCP Rule	TCP	8080	0.0.0.0/0	
Custom TCP Rule	TCP	4789	172.31.45.50/32	Worker-1
Custom TCP Rule	TCP	4789	172.31.44.151/32	Worker-2
SSH	TCP	22	0.0.0.0/0	
SSH	TCP	22	::/0	
Custom UDP Rule	UDP	7946	172.31.45.50/32	Worker-1
Custom UDP Rule	UDP	7946	172.31.44.151/32	Worker-2
Custom UDP Rule	UDP	7946	172.31.47.114/32	Manager
Custom TCP Rule	TCP	5000	0.0.0.0/0	
Custom TCP Rule	TCP	5000	::/0	
Custom TCP Rule	TCP	2376	172.31.45.50/32	Worker-1
Custom TCP Rule	TCP	2376	172.31.44.151/32	Worker-2
Custom TCP Rule	TCP	7946	172.31.45.50/32	Worker-1
Custom TCP Rule	TCP	7946	172.31.44.151/32	Worker-2
Custom TCP Rule	TCP	7946	172.31.47.114/32	Manager
Custom UDP Rule	UDP	4789	172.31.45.50/32	Worker-1
Custom UDP Rule	UDP	4789	172.31.44.151/32	Worker-2
Custom UDP Rule	UDP	4789	172.31.47.114/32	Manager

Когда будете их применять, не забудьте заменить IP-адреса машин на приватные IP-адреса (Private IPs) ваших экземпляров. Не забудьте применить настройки данной группы безопасности к виртуальным машинам, которые будут входить в кластер.

В рамках данной лабораторной работы мы не будем полагаться на DNS, и для простоты, просто добавим данные об узлах, которые будут формировать кластер в файлы /etc/hosts на каждом узле. Необходимо отредактировать файл, указав там приватные IP-адреса всех узлов, которые будут входить в кластер (у вас они будут другими):

```
$ sudo nano /etc/hosts
172.31.47.114    manager
172.31.45.50    worker-1
```

```
172.31.44.151     worker-2
```

1.1.4 Инициализация роя Docker Swarm:

Теперь мы инициализируем рой Docker Swarm на узлах менеджера и, поскольку у нас есть более одного сетевого интерфейса, мы укажем опцию --advertise-addr где необходимо указать приватный IP нашего узла-менеджера.

В результате выполнения команды вам будет выдана строка, которую вы должны выполнить на узлах-рабочих, чтобы они зашли в рой.

```
[manager] $ docker swarm init --advertise-addr 172.31.47.114
Swarm initialized: current node (siqyf3yricsvjkzvej00a9b8h) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-06dn5vstf8vyn39yndfd4rozjurix2e0y3ccjgoooe1-0iv6vj6shlnjj8ffm1sxg45dn 172.31.47.114:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Если мы хотели бы добавить несколько менеджеров, нам нужно было бы запустить вышеупомянутую команду, чтобы добавить больше менеджеров.

1.1.5 Подключение рабочих узлов к менеджеру:

Теперь, чтобы присоединиться к рабочим узлам в рой, мы запустим команду docker swarm join, которую мы получили в этапе инициализации роя:

```
[worker-1] $ docker swarm join --token SWMTKN-1-0eith07xkcg93lzftuhjmxaxwfa6mbkjsmjzb3d3sx9cobc2zp-97s6xzdt27y2gk3kpm0cgo6y2 172.31.18.90:2377
This node joined a swarm as a worker.
```

Для присоединения второго рабочего узла к рою:

```
[worker-2] $ docker swarm join --token SWMTKN-1-0eith07xkcg93lzftuhjmxaxwfa6mbkjsmjzb3d3sx9cobc2zp-97s6xzdt27y2gk3kpm0cgo6y2 172.31.18.90:2377
This node joined a swarm as a worker.
```

Чтобы увидеть состояние узла, а также чтобы мы могли определить, активны/доступны ли узлы, из узла менеджера, посмотрим все узлы в рое:

```
[manager] $ docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
j14mte3v1jhtbm3pb2qrpgwp6     worker-1  Ready   Active
siqyf3yricsvjkzvej00a9b8h *   master    Ready   Active        Leader
srl5yzme5hxnzxal2t1efmwje     worker-2  Ready   Active
```

Если вы потеряли токен соединения, его можно получить, запустив следующую команду для токена менеджера:

```
$ docker swarm join-token manager -q
SWMTKN-1-67chzvi4epx28ii18gizcia8idfar5hokojz660igeavnrltf0-09ijujbnnh4v960b8xel58pmj
```

И следующая команда выдаст вам токен рабочего узла:

```
$ docker swarm join-token worker -q
SWMTKN-1-67chzvi4epx28ii18gizcia8idfar5hokojz660igeavnrltf0-acs21nn28v17uwhw0oqg5ibwx
```

На данный момент мы можем заметить, что у нас нет сервисов, запущенных в нашем рое:

```
[manager] $ docker service ls
ID  NAME  MODE  REPLICAS  IMAGE
```

1.2 Развертывание масштабируемого сервиса nginx

Давайте создадим сервис nginx с 2 репликами, что означает, что будет 2 контейнера nginx исполняться в нашем рое.

Если какой-либо из этих контейнеров остановится, они будут порождаться снова до тех пор, пока количество их реплик не достигнет числа, указанного в опции replica:

```
[manager] $ docker service create --name my-web --publish 8080:80 --replicas 2 nginx
```

Давайте взглянем на наш сервис nginx:

```
[manager] $ docker service ls
ID            NAME    MODE        REPLICAS  IMAGE
1okycpshfusq  my-web  replicated  2/2       nginx:latest
```

После того как мы видим, что количество реплик 2/2 наш сервис готов.

Чтобы узнать, на каких узлах работают наши контейнеры, которые составляют наш сервис, выполним следующую команду:

```
[manager] $ docker service ps my-web
ID            NAME      IMAGE         NODE      DESIRED STATE  CURRENT STATE           ERROR
PORTS
k0qqrh8s0c2d  my-web.1  nginx:latest  worker-1  Running        Running 30 seconds ago
nku9wer6tml1  my-web.2  nginx:latest  worker-2  Running        Running 30 seconds ago
```

Мы также можем получить более подробную информацию о нашем сервисе, используя опцию inspect:

```
[manager] $ docker service inspect my-web
[
    {
        "ID": "1okycpshfusqos8023gtvdf11",
        "Version": {
"Index": 22
        },
        "CreatedAt": "2017-06-21T07:58:15.177547236Z",
        "UpdatedAt": "2017-06-21T07:58:15.178864919Z",
        "Spec": {
"Name": "my-web",
"TaskTemplate": {
"ContainerSpec": {
"Image":
"nginx:latest@sha256:41ad9967ea448d7c2b203c699b429abe1ed5af331cd92533900c6d77490e0268",
"DNSConfig": {}
},
"Resources": {
"Limits": {},
"Reservations": {}
},
"RestartPolicy": {
"Condition": "any",
"MaxAttempts": 0
},
"Placement": {},
```

```
"ForceUpdate": 0
},
"Mode": {
"Replicated": {
"Replicas": 2
}
},
"UpdateConfig": {
"Parallelism": 1,
"FailureAction": "pause",
"MaxFailureRatio": 0
},
"EndpointSpec": {
"Mode": "vip",
"Порты":
{
"Protocol": "tcp",
"TargetPort": 80,
"PublishedPort": 8080,
"PublishMode": "ingress"
}
]
}
        },
        "Endpoint": {
"Spec": {
"Mode": "vip",
"Порты":
{
"Protocol": "tcp",
"TargetPort": 80,
"PublishedPort": 8080,
"PublishMode": "ingress"
}
]
},
"Порты":
{
"Protocol": "tcp",
"TargetPort": 80,
"PublishedPort": 8080,
"PublishMode": "ingress"
}
],
"VirtualIPs": [
{
"NetworkID": "wvim41tbf7tsmi7z49g56bisa",
"Addr": "10.255.0.6/16"
}
]
        },
        "UpdateStatus": {
"StartedAt": "0001-01-01T00:00:00Z",
"CompletedAt": "0001-01-01T00:00:00Z"
        }
    }
]
```

Мы можем получить информацию о портах конечных точек, используя опцию inspect и использовать параметр --format для фильтрации вывода:

```
[manager] $ docker service inspect --format="{{json .Endpoint.Ports}}" my-web  | python -m
json.tool
```

Если команда не выполняется, попробуйте установить python на узел командой

```
sudo apt install python
```

Из полученной информации мы найдем, что в поле PublishedPort будет указан порт, который мы открываем. В поле TargetPort будет указан порт, который установлен на прослушивание в контейнере:

```
[
    {
        "Protocol": "tcp",
        "PublishMode": "ingress",
        "PublishedPort": 8080,
        "TargetPort": 80
    }
]
```

Чтобы получить виртуальную информацию об Virual IP запустим команду:

```
[manager] $ docker service inspect --format="{{json .Endpoint.VirtualIPs}}" my-web  | python -
m json.tool
[
    {
        "Addr": "10.255.0.6/16",
        "NetworkID": "wvim41tbf7tsmi7z49g56bisa"
    }
]
```

Чтобы получить только опубликованный порт:

```
[manager] $ docker service inspect --format="{{range
.Endpoint.Ports}}{{.PublishedPort}}{{end}}" my-web
8080
```

Можем также представить список наших сетей:

```
[manager] $ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
1cb24ee7e385        bridge              bridge              local
503b2f0eda49        docker_gwbridge     bridge              local
fe679d82d502        host                host                local
wvim41tbf7ts        ingress             overlay             swarm
310b9219ec0c        none                null                local
```

Мы можем также провести инспекцию нашей сети для получения дополнительной информации:

```
[manager] $ docker network inspect ingress
[
    {
        "Name": "ingress",
        "Id": "wvim41tbf7tsmi7z49g56bisa",
        "Created": "2017-06-21T07:50:44.14232108Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
"Driver": "default",
"Options": null,
"Config": [
{
"Subnet": "10.255.0.0/16",
"Gateway": "10.255.0.1"
}
]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {
"ingress-sbox": {
"Name": "ingress-endpoint",
"EndpointID": "b6b2a8f3f233cfc77806718a9c47daf02ad128ba81c96d6382ff1d3799c3b5c1",
"MacAddress": "02:42:0a:ff:00:03",
"IPv4Address": "10.255.0.3/16",
"IPv6Address": ""
}
        },
        "Options": {
"com.docker.network.driver.overlay.vxlanid_list": "4096"
        },
        "Labels": {},
        "Peers": [
{
"Name": "master-59041a33bebc",
"IP": "172.31.18.90"
},
{
"Name": "worker-1-cfee817ddb5f",
```

```
"IP": "172.31.20.94"
},
{
"Name": "worker-2-40891fb1fa3f",
"IP": "172.31.21.50"
}
        ]
    }
]
```

Чтобы получить IP контейнера в кластере выполним команду:

```
[manager] $ docker service ps my-web
ID            NAME      IMAGE         NODE      DESIRED STATE  CURRENT STATE           ERROR
PORTS
k0qqrh8s0c2d  my-web.1  nginx:latest  worker-1  Running        Running 31 minutes ago
nku9wer6tml1  my-web.2  nginx:latest  worker-2  Running        Running 31 minutes ago
```

Например, мы хотели бы определить IP для нашего сервиса web.1 nginx на узле worker-1 (он может быть там не установлен, если нет, то проверьте worker-2):

```
[worker-1] $ docker ps
CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS              PORTS               NAMES
72e05af4e654
```

Затем выполним inspect на контейнере, чтобы получить IP-адрес:

```
[worker-1] $ docker inspect 72e05af4e654 --format "{{json
.NetworkSettings.Networks.ingress.IPAddress}}"
"10.255.0.8"
```

Или:

```
[worker-1] $ docker inspect 72e05af4e654 --format "{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}"
10.255.0.8
```

Для того, чтобы проверить корректность доступа к нашему приложению, с нашего локального компьютера

```
$ curl ManagerPublicIP:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
width: 35em;
margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
```

1.3 Создание демонстрационного веб-приложения:

В этом примере мы создадим приложение ***Python Flask Example***, которое распечатывает хост-имя контейнера (containerid), а также печатает UUID.

Это позволит нам увидеть, когда мы масштабируем службу, что веб-приложение будет сообщать нам разные имена хостов на каждом запросе HTTP, так как будет работать алгоритм балансировки нагрузки round-robin.

1.3.1 Создание приватного репозитория для сохранения наших образов:

Мы разместим приватный сервер репозитория в качестве службы в нашем рое. На нем будут храниться образы веб-приложений, что будут запускаться в нашем рое:

```
[manager] $ docker service create --name registry --publish "5000:5000" registry:2
```

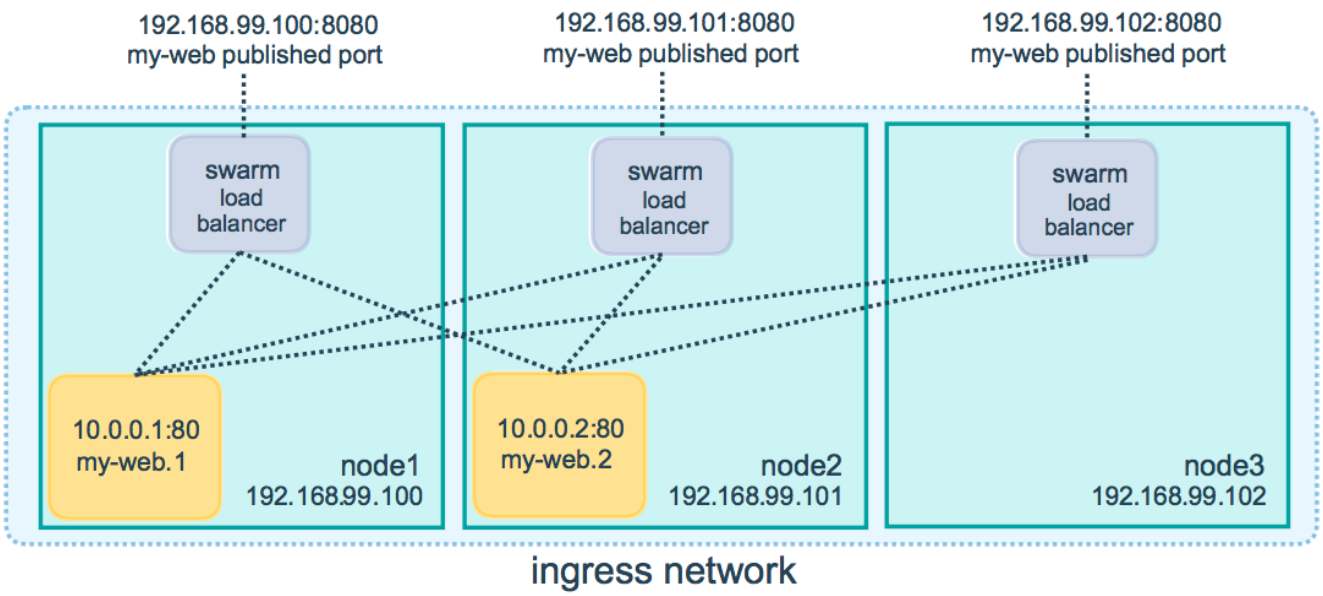
Попробуем сделать запрос в API из каждого узла нашего кластера на наш приватный репозиторий, чтобы убедиться в возможности доступа. «manager:5000» — это сетевое имя нашего узла-менеджера Docker Swarm. Получая такой запрос, по overlay network он будет пересылать его на те узлы, где развернут сервис registry:

```
$ curl manager:5000/v2/_catalog
{"repositories":[]}
```

Удивительная вещь здесь заключается в том, что Docker Swarm использует технологию ingress routing mesh, таким образом все узлы, которые являются частью

роя будут отвечать на запросы на открытый порт, даже если необходимый контейнер не находится на этом узле.

Например, мы видим, что наш контейнер находится на узле worker-1. Если мы получим публичные IP нашего узла manager node и нашего узла worker node, а затем сделаем HTTP-запросы на порт 5000 (в данном случае) этих IP, и то мы увидим, что запросы будут обработаны.



Режим Docker Engine swarm позволяет легко публиковать порты для сервисов, чтобы сделать их доступными для клиентов за пределами роя. Все узлы участвуют в сетке входящей маршрутизации. Маршрутизационная сетка позволяет каждому узлу в рое принимать соединения на опубликованных портах для любой службы, запущенной в рое, даже если на узле не запущена ни одна задача. Маршрутизатор mesh направляет все входящие запросы к опубликованным портам на доступных узлах в активный контейнер. Для работы этой сети необходимы следующие открытые порты

- Порт 7946 **TCP/UDP** для нахождения контейнеров.
- Port 4789 **UDP** для поддержки работы ingress сети.

1.3.2 Создание сети:

Создадим сеть, которая будет использовать overlay driver, который охватывает несколько узлов докеров:

```
[manager] $ docker network create --driver overlay --subnet 10.24.90.0/24 mynet
```

Теперь, когда наша сеть создана, давайте проинспектируем сеть, чтобы определить ее детали:

```
$ docker network inspect mynet
[
    {
        "Name": "mynet",
        "Id": "byzlzl4r2mwv4j9jisu7ellhs",
        "Created": "0001-01-01T00:00:00Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
"Driver": "default",
"Options": null,
"Config": [
{
"Subnet": "10.24.90.0/24",
"Gateway": "10.24.90.1"
}
]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": null,
        "Options": {
"com.docker.network.driver.overlay.vxlanid_list": "4103"
        },
        "Labels": null
    }
]
```

Когда сервис связан с этой сетью, с точки зрения разрешения dns, имя службы будет отвечать на виртуальный ip (vip) службы.

1.3.3 Создадим код нашего приложения, а также Dockerfile и docker-compose:

Создадим каталог, где будет храниться код нашего приложения:

```
$ mkdir flask-demo
$ cd flask-demo
```

Укажим зависимость Python в файле requirements.txt:

```
$ nano requirements.txt
flask
```

Создадим приложение Python Flask (app.py), которое будет показать название контейнера и случайную строку uuid:

```
$ cat app.py
from flask import Flask
import os
import uuid

app = Flask(__name__)

@app.route('/')
def index():
    hostname = os.uname()[1]
    randomid = uuid.uuid4()
    return 'Container Hostname: ' + hostname + ' , ' + 'UUID: ' + str(randomid)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5098)
```

Создадим Dockerfile, который будет служить инструкцией о том, как построить образ нашего веб-приложения:

```
$ nano Dockerfile
FROM python:3.4-alpine
ADD . /app
WORKDIR /app
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Создадим файл Docker Compose , который будет формировать образ с помощью нашего Dockerfile, указывая необходимые открытые порты и регистр образов. Также укажем, что приложение должно работать в рамках сети mynet:

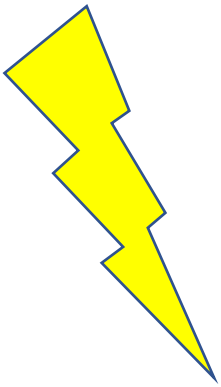
```
$ nano docker-compose.yml
version: '3'

services:
  web:
    image: master:5000/flask-app
    build: .
    ports:
      - "80:5098"

networks:
  default:
    external:
      name: mynet
```

Посмотрим на файл подробнее. Строка «`image: master:5000/flask-app`» означает, что образ необходимо будет взять из локального репозитория, доступного по адресу

«master:5000». Строки «ports:» «- "80:5098"» означают, что локальный порт приложения 5098 будет назначен на порт 80.

Проверим запуск docker compose локально на одном узле:

```
$ docker-compose up
 # do some testing
$ docker-compose down
```



Если приложение не может подключиться к сети mynet что мы создали для Docker Swarm, то необходимо пересобрать (удалить и создать заново) сеть mynet с параметром --attachable

```
[manager] $ docker network rm mynet
[manager] $ docker network create --driver overlay --attachable --subnet
10.24.90.0/24 mynet
```

Соберем образ, загрузим его в приватный репозиторий, который указан **В** docker-compose.yml :

```
$ docker-compose build
$ docker-compose push
Pushing web (master:5000/flask-app:latest)...
Pushing web (master:5000/flask-app:latest)...
The push refers to repository [master:5000/flask-app]
ERROR: Get https://master:5000/v2/: dial tcp: lookup master on 172.31.0.2:53: no such host
```

Если загрузка образа в репозиторий не происходит, это связано с тем, что docker не может сопоставить имя узла «master», указанное в docker-compose.yml с адресом, где развернут наш сервис репозитория. В файлах hosts мы назвали головной узел не «master» а «manager», поэтому укажем этот узел в файле в docker-compose.yml вместо `master:5000 -> manager:5000`

Пересоберем и попробуем еще раз загрузить наш образ в репозиторий

```
$ docker-compose build
$ docker-compose push
Pushing web (manager:5000/flask-app:latest)...
The push refers to repository [manager:5000/flask-app]
ERROR: Get https://manager:5000/v2/: http: server gave HTTP response to HTTPS client
```

Чтобы исправить эту проблему, на каждом узле необходимо создать файл `/etc/docker/daemon.json` и заполнить его со следующей информацией:

```
sudo nano /etc/docker/daemon.json
{ "insecure-registries":["manager:5000"] }
```

После этого перезапускаем docker командой :

```
sudo systemctl restart docker
```

Это необходимо сделать на всех узлах.

Теперь собираем всё еще разок.

```
$ docker-compose build
$ docker-compose push
Pushing web (manager:5000/flask-app:latest)...
The push refers to repository [manager:5000/flask-app]
d0df9f5805cc: Pushed
18d1f3d4d258: Pushed
62de8bcc470a: Pushed
58026b9b6bf1: Pushed
fbe16fc07f0d: Pushed
aabe8fddede5: Pushed
bcf2f368fe23: Pushed
latest: digest: sha256:6014c18bc6e0f93c32b742dfd6db16e18404ee6984b716ea4b7de155239a7ade size:
1786
```

Теперь, когда наш образ сохранен в нашем приватном репозитории, создадим службу из этого образа и опубликуем на порт `80` то, что будет развернуто внутри контейнера на порту `5098` (этот порт указан в коде).

Обратите внимание, что мы установили только 1 реплику, что означает, что каждый раз, когда мы делаем HTTP-запрос, нам будет возвращаться одно и тоже имя узла, так как у нас будет только 1 контейнер в нашем рое.

Мы также установили параметр --update-delay в значение 5 секунд что означает, что при обновлении нашего сервиса, каждая задача (контейнер) будет обновляться с 5-секундной задержкой между каждым из них.

1.3.4 Создание сервиса Flask-Demo:

```
[manager] $ docker service create --name flask-demo --network mynet --update-delay 5s --
publish 80:5098 --replicas 1 manager:5000/flask-app
```

1.3.5 Просмотр сервисов в нашем рое:

Просмотрим список сервисов в нашем рое и проверим, является ли число реплик таким же, как и то что было установлено при создании сервиса:

```
[manager] $ docker service ls
ID            NAME        MODE        REPLICAS  IMAGE
5y520y6fau5j  flask-demo  replicated  1/1       master:5000/flask-app:latest
xfk2z5s2ybcg  registry    replicated  1/1       registry:2
```

После того как мы видим, что желаемое число реплик достигнуто, мы можем выполнить команду ps, чтобы увидеть состояние и время работы, а также на каком узле контейнер живет:

```
$ docker service ps flask-demo
ID            NAME          IMAGE                          NODE      DESIRED STATE  CURRENT
STATE           ERROR  PORTS
u7ab0xhwrlzh  flask-demo.1  master:5000/flask-app:latest  worker-1  Running        Running 11
seconds ago
```

Проверим работу контейнера второй раз, осуществив вход на узел worker-1 и проверив, функционирует ли контейнер на указанном узле:

```
[worker-1] $ docker ps --filter "name=flask-demo" --format "table {{.ID}}\t{{.Names}}"
CONTAINER ID        NAMES
4e6bd7a8a2bf        flask-demo.1.u7ab0xhwrlzh3r8xme08ievop
```

Посмотрим на открытый порт:

```
$ docker service inspect --format="{{range .Endpoint.Ports}}{{.PublishedPort}}{{end}}" flask-demo
80
```

1.3.6 Тестирование нашего сервиса:

Теперь из клиента за пределами нашего роя мы сделаем HTTP-запрос к нашему менеджеру и IP-адресам рабочего узла:

```
[client] curl -XGET ManagerNodePublicIP
Container Hostname: 4e6bd7a8a2bf , UUID: 771cc6bd-b669-4f68-83c6-cbb30ad01380
[client] curl -XGET Worker1NodePublicIP
Container Hostname: 4e6bd7a8a2bf , UUID: c4f4115b-e848-4941-a399-b703afed249d
```

Мы видим, что нам нужно изменить код так, чтобы он заканчивал вывод символом новой строки

1.3.7 Обновление нашего кода приложения:

```
$ nano app.py
```

Измените код так, чтобы мы добавляли символ новой строки после нашего UUID:

Заменим строку

```
return 'Container Hostname: ' + hostname + ' , ' + 'UUID: ' + str(randomid)
```

На следующую:

```
return 'Container Hostname: ' + hostname + ' , ' + 'UUID: ' + str(randomid) + '\n'
```

1.3.8 Создадим новый образ из обновленного кода, загрузим его в частный репозиторий:

Теперь, когда наше приложение обновлено, мы должны собрать его новый образ и загрузить его в наш приватный репозиторий. Мы могли бы применить тег к нашему изображению, например `flask-app:v1` но поскольку мы решили их не использовать будем использовать тег `latest`, который отмечает последнюю сборку образа.

Если используется docker-compose, то тег должен быть указан в файле `docker-compose.yml`. Если docker-compose не используется, то тег необходимо указать через парамтер `docker build`:

```
$ docker-compose build
$ docker-compose push
Pushing web (manager:5000/flask-app:latest)...
The push refers to a repository [manager:5000/flask-app]
```

Как альтернативу, мы также можем создать образ без docker-compose:

```
$ docker build -t flask-app .
$ docker tag flask-app master:5000/flask-app
$ docker push manager:5000/flask-app
The push refers to a repository [manager:5000/flask-app]
```

Проверим, что мы можем увидеть обновленный образ в нашем частном репозитории:

```
$ docker images manager:5000/flask-app
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
manager:5000/flask-app   latest              3253d71edfa4        17 minutes ago      93.3 MB
manager:5000/flask-app   <none>              b7fd5b5e7892        51 minutes ago      93.3 MB
```

1.3.9 Обновление службы:

Применим механизм обновления rolling update для нашей службы:

```
$ docker service update --image manager:5000/flask-app:latest flask-demo
```

Убедимся, что служба работает:

```
$ docker service ls
ID            NAME        MODE        REPLICAS  IMAGE
5y520y6fau5j  flask-demo  replicated  1/1       manager:5000/flask-app:latest
xfk2z5s2ybcg  registry    replicated  1/1       registry:2
```

Запустим docker service ps с именем нашей службы, чтобы проверить, что предыдущий контейнер был остановлен, а новый контейнер с обновленным кодом был запущен:

```
$ docker service ps flask-demo
ID            NAME              IMAGE                          NODE      DESIRED STATE  CURRENT
STATE           ERROR  PORTS
ltvenwz4jidh  flask-demo.1      master:5000/flask-app:latest  worker-2  Running        Running
2 minutes ago
u7ab0xhwrlzh   \_ flask-demo.1  master:5000/flask-app:latest  worker-1  Shutdown
Shutdown 2 minutes ago
```

(Опционально:) Еще раз, мы можем войти на узел worker-2, чтобы убедиться, что мы можем увидеть container-id:

```
[worker-2] $ docker ps --filter "name=flask-demo" --format "table {{.ID}}\t{{.Names}}"
CONTAINER ID        NAMES
499b2a76da67        flask-demo.1.ltvenwz4jidhj7esc89ee6gzp
```

1.3.10 Тестирование обновленного веб-приложения:

Теперь давайте сделаем те же самые HTTP-запросы, чтобы проверить была ли обновлена служба:

```
[client] $ curl -XGET 52.214.150.151
Container Hostname: 499b2a76da67 , UUID: b1d534eb-e737-46ce-91e2-5800bf24a72e

[client] $ curl -XGET 52.214.150.151
Container Hostname: 499b2a76da67 , UUID: 6c47b0c5-7ffa-4501-a522-c430e420b12c
```

Мы можем видеть обновленный вывод, с переносом строки.

1.3.11 Масштабирование нашего приложения:

Проверим возможность масштабирования нашего приложения до 10 контейнеров:

```
$ docker service scale flask-demo=10
flask-demo scaled to 10
```

Давайте взглянем на наши количество реплик:

```
$ docker service ls
ID            NAME        MODE        REPLICAS  IMAGE
5y520y6fau5j  flask-demo  replicated  10/10     master:5000/flask-app:latest
```

```
xfk2z5s2ybcg  registry    replicated  1/1       registry:2
```

Мы видим, что наши задачи реплицируются, и мы можем увидеть, на каких узлах они работают:

```
$ docker service ps flask-demo
ID            NAME              IMAGE                          NODE      DESIRED STATE  CURRENT
STATE           ERROR  PORTS
ltvenwz4jidh  flask-demo.1      master:5000/flask-app:latest  worker-2  Running        Running
2 minutes ago
xyxi33keqdny   \_ flask-demo.1  master:5000/flask-app:latest  worker-1  Shutdown
Shutdown 2 minutes ago
u7ab0xhwrlzh   \_ flask-demo.1  master:5000/flask-app:latest  worker-1  Shutdown
Shutdown 6 minutes ago
yhnf8wyrimvb  flask-demo.2      master:5000/flask-app:latest  master    Running        Running
39 seconds ago
vxghtq86537m  flask-demo.3      master:5000/flask-app:latest  worker-1  Running        Running
37 seconds ago
f3p8ym23r4ir  flask-demo.4      master:5000/flask-app:latest  worker-1  Running        Running
37 seconds ago
q9hgb8z58ybd  flask-demo.5      master:5000/flask-app:latest  master    Running        Running
39 seconds ago
qdxef5d2bspt  flask-demo.6      master:5000/flask-app:latest  worker-1  Running        Running
37 seconds ago
a7wjn1lo9jbz  flask-demo.7      master:5000/flask-app:latest  worker-2  Running        Running
41 seconds ago
776oi10gyo32  flask-demo.8      master:5000/flask-app:latest  worker-1  Running        Running
37 seconds ago
u2mpr2t7aoj5  flask-demo.9      master:5000/flask-app:latest  master    Running        Running
39 seconds ago
kypazgcch5aa  flask-demo.10     master:5000/flask-app:latest  worker-2  Running        Running
41 seconds ago
```

Дополнительно посмотрим распределение контейнеров по узлам, где они работают:

```
[master] $ docker ps --filter "name=flask-demo" --format "table {{.ID}}\t{{.Names}}"
CONTAINER ID        NAMES
9e6b28ba711f        flask-demo.9.u2mpr2t7aoj5c5nggr2k3dvjf
4c3edef5c69b        flask-demo.2.yhnf8wyrimvbg937dt6tkjwb0
5a052e37ef73        flask-demo.5.q9hgb8z58ybd07cztmiqnqogy
```

А на узле worker-1:

```
[worker-1] $ docker ps --filter "name=flask-demo" --format "table {{.ID}}\t{{.Names}}"
CONTAINER ID        NAMES
499b2a76da67        flask-demo.6.qdxef5d2bspt2dtga4pdj6ab1
3bd92ade8242        flask-demo.4.f3p8ym23r4iredo7xqyt1yt7h
bd64090dc3f5        flask-demo.3.vxghtq86537mpxpo5t31zm1i4
3a5effab8827        flask-demo.8.776oi10gyo32h76czx7jqwkgf
```

Далее, на узле worker-2:

```
[worker-2] $ docker ps --filter "name=flask-demo" --format "table {{.ID}}\t{{.Names}}"
CONTAINER ID        NAMES
20e8148ab72f        flask-demo.10.kypazgcch5aaw4l04ha8r4elv
217c86d234de        flask-demo.7.a7wjn1lo9jbzv4p5hbgierp7g
a186ed70646c        flask-demo.1.ltvenwz4jidhj7esc89ee6gzp
```

Поскольку служба реплицируется на нескольких контейнерах, наше приложение должно теперь обслуживать запросы на основе разных контейнеров:

```
[client] $ curl -XGET 52.214.150.151
Container Hostname: 9e6b28ba711f , UUID: 16780535-8c5f-445a-a1dd-fdb10e07bc59

[client] $ curl -XGET 52.214.150.151
Container Hostname: 4c3edef5c69b , UUID: ea496a8f-b49d-470b-a0f4-e695810374aa
```

Давайте сделаем 10 запросов, чтобы увидеть имя хоста в ответах из контейнеров:

```
$ for x in {1..10}; do curl -XGET 52.214.150.151; done

Container Hostname: 3a5effab8827 , UUID: 00e3454e-4d0c-48d9-96ec-e8875c893b36
Container Hostname: 9e6b28ba711f , UUID: bf6da863-94a3-445d-b386-25e290004ed2
Container Hostname: 4c3edef5c69b , UUID: 1a611f14-0cc6-4495-b246-5bb4db65a002
Container Hostname: 5a052e37ef73 , UUID: 684f00dc-a17c-418a-bfb7-d2bac50d4c6e
Container Hostname: 20e8148ab72f , UUID: 85696153-d3d5-4f7a-b663-eb4e0c199555
Container Hostname: 217c86d234de , UUID: df7d2bd8-4cae-4150-a825-754e84709724
Container Hostname: a186ed70646c , UUID: 88cc84fe-44f7-4d4d-9f67-9479d52688d7
Container Hostname: 499b2a76da67 , UUID: e4d643e0-10e6-452f-abc7-f712fdf07cdd
Container Hostname: 3bd92ade8242 , UUID: ce23f3b2-c55d-4533-b164-5f60b8bf3aa0
Container Hostname: bd64090dc3f5 , UUID: 9e2bb8c2-848e-49ff-99d8-dff47644b74b
```

Сделаем 12 запросов. Теоретически мы должны увидеть 10 уникальных имен хоста:

```
$ for x in {1..12}; do curl -XGET 52.214.150.151; done

Container Hostname: 3a5effab8827 , UUID: 84a874b9-3767-4b07-82da-ba2ede0e2240
Container Hostname: 9e6b28ba711f , UUID: 0e1a7736-e89b-4c30-9739-97ef7b8a203a
Container Hostname: 4c3edef5c69b , UUID: 455df8e9-b465-4b58-865e-56d708148c26
Container Hostname: 5a052e37ef73 , UUID: 7478f449-ec48-4378-8cd1-fa0ae3465f78
Container Hostname: 20e8148ab72f , UUID: 25dd11fa-c09d-437a-b4d8-a12b7933a245
Container Hostname: 217c86d234de , UUID: b4e26634-037a-4694-b849-896c0bc89946
Container Hostname: a186ed70646c , UUID: 3af5619d-bfa9-4ac7-8190-0f11876b33b1
Container Hostname: 499b2a76da67 , UUID: e38bd20d-c9e7-4b1e-9f1f-a25574207558
Container Hostname: 3bd92ade8242 , UUID: 3caec655-a4f8-4956-a5b0-00a68db99233
Container Hostname: bd64090dc3f5 , UUID: 2c8cfe74-42ff-4463-b069-03fb3be5f290
Container Hostname: 3a5effab8827 , UUID: bb84adea-1dee-4b57-aea5-e086e0a979ce
Container Hostname: 9e6b28ba711f , UUID: 72706b15-544b-4a3f-b66c-5e1d3a363cc7
```

Запустим баш-скрипт для подсчета дубликатов:

```
$ for x in {1..10}; do curl -XGET 52.214.150.151; done >> run1.txt

$ for x in {1..12}; do curl -XGET 52.214.150.151; done >> run2.txt
```

Наш первый запуск (10 запросов):

```
[client] $ cat run1.txt | awk '{print $3}' | sort | uniq -c
      1 20e8148ab72f
      1 217c86d234de
      1 3a5effab8827
      1 3bd92ade8242
      1 499b2a76da67
      1 4c3edef5c69b
      1 5a052e37ef73
      1 9e6b28ba711f
      1 a186ed70646c
      1 bd64090dc3f5
```

Второй запуск (12 запросов):

```
[client] $ cat run2.txt | awk '{print $3}' | sort | uniq -c
      1 20e8148ab72f
      1 217c86d234de
      2 3a5effab8827
      1 3bd92ade8242
      1 499b2a76da67
      1 4c3edef5c69b
      1 5a052e37ef73
      2 9e6b28ba711f
      1 a186ed70646c
      1 bd64090dc3f5
```

Создадим программу на Python для случайного выбора одного из 3 узлов с докер-контейнерами для выполнения запроса HTTP GET и возврата IP-адреса Docker Host и идентификатора контейнера, обслуживающего ответ:

```
>>> import time
>>> import random
>>> import requests
>>> for x in xrange(20):
...     time.sleep(0.125)
...     docker_host = random.choice(['34.253.158.130', '34.251.150.142', '52.214.150.151'])
...     get_request = requests.get('http://' + str(docker_host)).content.split(' ')[2]
...     print("Host: {0}, Container: {1}".format(docker_host, get_request))
...
Host: 52.214.150.151, Container: 217c86d234de
Host: 52.214.150.151, Container: a186ed70646c
Host: 34.251.150.142, Container: bd64090dc3f5
Host: 34.251.150.142, Container: 3a5effab8827
Host: 52.214.150.151, Container: 499b2a76da67
```

```
Host: 52.214.150.151, Container: 3bd92ade8242
Host: 52.214.150.151, Container: bd64090dc3f5
Host: 52.214.150.151, Container: 3a5effab8827
Host: 34.253.158.130, Container: 9e6b28ba711f
Host: 52.214.150.151, Container: 9e6b28ba711f
Host: 52.214.150.151, Container: 4c3edef5c69b
Host: 34.251.150.142, Container: 4c3edef5c69b
Host: 34.251.150.142, Container: 9e6b28ba711f
Host: 34.253.158.130, Container: 4c3edef5c69b
Host: 34.253.158.130, Container: 5a052e37ef73
Host: 34.253.158.130, Container: 20e8148ab72f
Host: 34.251.150.142, Container: 5a052e37ef73
Host: 34.253.158.130, Container: 217c86d234de
Host: 52.214.150.151, Container: 5a052e37ef73
Host: 34.251.150.142, Container: 217c86d234de
```

1.4 Задание на самостоятельную реализацию

На основе изученных технологий работы с Docker, Docker Compose и Docker Swarm необходимо разработать и показать масштабируемость в среде AWS собственного веб-приложения, обеспечивающего обработку CSV-файлов и реализующего их асинхронную (отложенную) обработку.

Веб-сервис должен предоставлять веб-API, обеспечивающий загрузку в сервис таблицы в виде CSV-файла для ее обработки и получения результата в виде N элементов из данного файла с максимальным значением значения данных по определенному полю данного CSV-файла (TOP-N).

Первая строка CSV-файла содержит заголовки соответствующих столбцов таблицы.

В качестве примера можно использовать данные со страницы https://www3.epa.gov/otaq/tcldata.htm (например, https://www3.epa.gov/fueleconomy/testcars/database/16tstcar.csv).

1.5 Дополнительные возможности

1.5.1 Только менеджер планирования:

Мы можем настроить узел нашего менеджера только для обслуживания запросов и выполнения планирования:

```
$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
j14mte3v1jhtbm3pb2qrpgwp6    worker-1  Ready   Active
siqyf3yricsvjkzvej00a9b8h *  master    Ready   Active        Leader
srl5yzme5hxnzxal2t1efmwje    worker-2  Ready   Active
```

После просмотра наших узлов в рое, мы можем выбрать, какой узла мы хотим, чтобы слить

```
$ docker node update --availability drain manager
manager
```

Чтобы проверить наше изменение:

```
$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
j14mte3v1jhtbm3pb2qrpgwp6    worker-1  Ready   Active
siqyf3yricsvjkzvej00a9b8h *  master    Ready   Drain         Leader
srl5yzme5hxnzxal2t1efmwje    worker-2  Ready   Active
```

Проверка наш узла менеджера:

```
$ docker node inspect --pretty manager
ID:                     siqyf3yricsvjkzvej00a9b8h
Hostname:               master
Joined at:              2017-06-21 07:50:43.525020216 +0000 utc
Status:
 State:                 Ready
 Availability:          Drain
 Address:               172.31.18.90
Manager Status:
 Address:               172.31.18.90:2377
 Raft Status:           Reachable
 Leader:                Yes
Platform:
 Операционная система: linux
 Architecture:          x86_64
Ресурсы:
 Процессоров:                1
 Memory:                3.673 GiB
Plugins:
  Network:              bridge, host, macvlan, null, overlay
  Volume:               local
```

```
Engine Version:         17.03.1-ce
```

И теперь мы видим, что наши контейнеры работают только на рабочих узлах:

```
$ docker service ps flask-demo | grep Runn
lig9wj70c5m8  flask-demo.1      master:5000/newapp:v1      worker-1  Running        Running 5
minutes ago
zmmzcm6xxzzo колба-демо.2 мастер:5000/newapp:v1 работник-2 Бег Бег 25 секунд назад
a292huwkzr23 колба-демо.3 мастер:5000/newapp:v1 работник-2 Бег 5 минут назад
uuhbswsdvziq  flask-demo.4      master:5000/newapp:v1      worker-1  Running        Running 4
minutes ago
x4vrrs6fgpqz  flask-demo.5      master:5000/newapp:v1      worker-2  Running        Running 25
seconds ago
...
```

1.5.2 Удаление службы:

```
$ docker service rm flask-demo
```

1.5.3 Удаление сети:

```
$ docker network rm mynet
```

1.5.4 Выход из роя:

Чтобы удалить узлы из роя, запустите следующие из узлов:

```
[worker-1] $ docker swarm leave
[worker-2] $ docker swarm leave
```

Мы можем перечислить наши узлы, чтобы получить узлы, которые мы хотим удалить от менеджера:

```
[master] $ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
j14mte3v1jhtbm3pb2qrpgwp6    worker-1  Down    Active
siqyf3yricsvjkzvej00a9b8h *  master    Ready   Drain         Leader
srl5yzme5hxnzxal2t1efmwje    worker-2  Down    Active
```

Удаление узлов из списка менеджеров:

```
$ docker node rm worker-1
worker-1
```

```
$ docker node rm worker-2
worker-2

$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
siqyf3yricsvjkzvej00a9b8h *  master    Ready   Drain         Leader
```

1.6 Ресурсы:

- https://docs.docker.com/engine/swarm/

На базе https://sysadmins.co.za/docker-swarm-getting-started-with-a-3-node-docker-swarm-cluster-with-a-scalable-app/