Указания к заданию 3. Многоконтейнерные окружения.

Цель: изучить принципы развертывания многоконтейнерных окружений в публичных облаках.

Критерии оценивания

Необходимо локально развернуть многоконрейнерное приложение вручную, настроив подсистему сетей Docker
Необходимо локально развернуть многоконтейнерное приложение используя утилиту Docker Compose.
В среде AWS необходимо развернуть многоконтейнерное приложение. Необходимо продемонстрировать публичную доступность и процесс работы веб-приложения в среде AWS.

Методические указания

1.1 Многоконтейнерные окружения

В прошлом разделе мы увидели, как легко и просто запускать приложения с помощью Докера. Мы начали с простого статического сайта, а потом запустили Flask-приложение. Оба варианта можно было запускать локально или в облаке, несколькими командами. Общая черта этих приложений: каждое из них работало **в одном контейнере**.

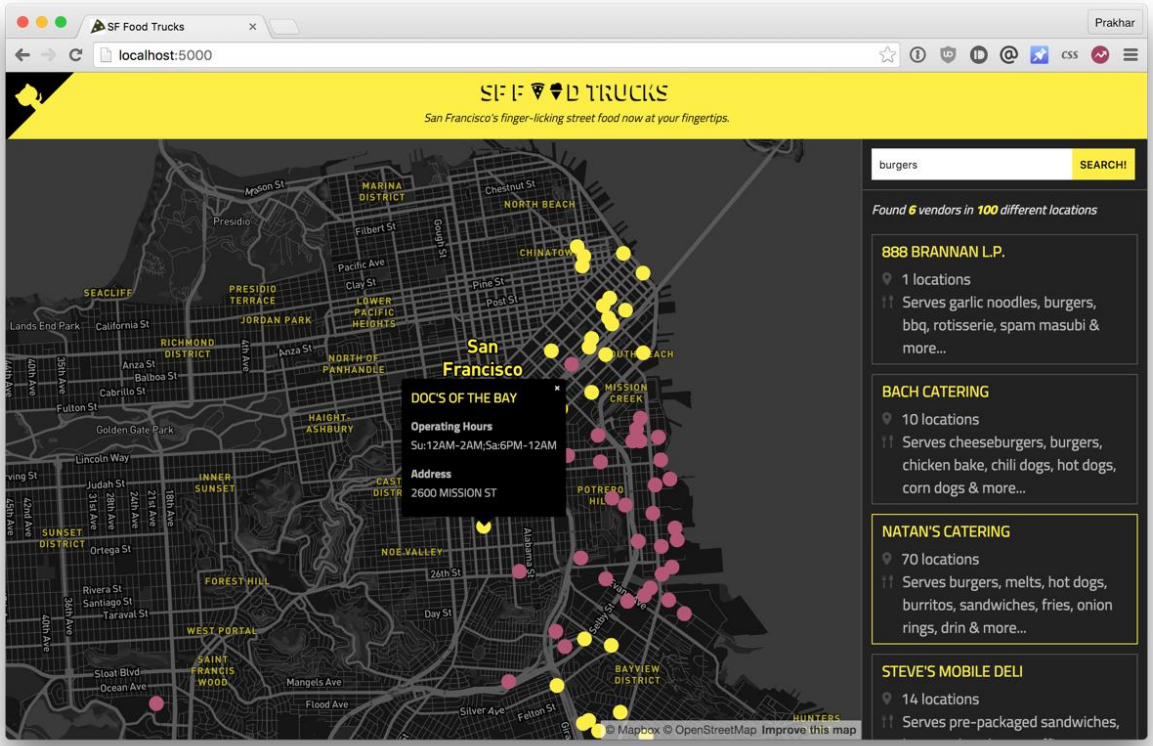
Если у вас есть опыт управления сервисами в продакшене, то вы знаете, что современные приложения обычно не такие простые. Почти всегда есть база данных (или другой тип постоянного хранилища). Системы вроде Redis и Memcached стали практически обязательной частью архитектуры веб-приложений. Поэтому, в этом разделе мы научимся "докеризировать" приложения, которым требуется несколько запущенных сервисов.

В частности, мы увидим, как запускать и управлять **многоконтейнерными** Докер-окружениями. Почему нужно несколько контейнеров, спросите вы? Ну, одна из главных идей Докера в том, что он предоставляет изоляцию. Идея совмещения процесса и его зависимостей в одной песочнице (называемой контейнером) и делает Докер мощным инструментом.

Аналогично тому, как приложение разбивают на части, стоит содержать отдельные **сервисы** в отдельных контейнерах. Разным частям скорее всего требуются разные ресурсы, и требования могут расти с разной скоростью. Если мы разделим эти части и поместим в разные контейнеры, то каждую часть приложения можно строить, используя наиболее подходящий тип ресурсов. Это также хорошо совмещается с идеей микро сервисов. Это одна из причин, по которой Докер (и любая другая технология контейнеризации) находится на передовой современных микро сервисных архитектур.

1.1.1 SF Food Trucks

Приложение, которое мы переведем в Докер, называется SF Food Trucks. Цель была сделать что-то полезное (и похожее на настоящее приложение из реального мира), что-то, что использует как минимум один сервис, но не слишком сложное для этого пособия.



Бэкэнд приложения написан на Питоне (Flask), а для поиска используется Elasticsearch - свободная программная поисковая система, которая может быть развернута внутри отдельного контейнера.

Код нашего приложения находится на Github. Мы используем это приложение, чтобы научиться запускать и развертывать много-контейнерное окружение.

В нашем приложении есть бэкэнд на Flask и сервис Elasticsearch. Очевидно, что можно поделить приложение на два контейнера: один для Flask, другой для Elasticsearch (ES). Если приложение станет популярным, то можно будет добавлять новые контейнеры в нужном месте, смотря где будет узкое место.

Отлично, значит нужно два контейнера. Мы уже создавали Flask-контейнер в прошлом разделе. А для Elasticsearch… давайте посмотрим, есть ли что-нибудь в хабе:

```
$ docker search elasticsearch
NAME                                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATE
D
elasticsearch                       Elasticsearch is a powerful open source se...   697       [OK]
itzg/elasticsearch                  Provides an easily configurable Elasticsea...   17                   [OK]
tutum/elasticsearch                 Elasticsearch image - listens in port 9200.     15                   [OK]
barnybug/elasticsearch              Latest Elasticsearch 1.7.2 and previous re...   15                   [OK]
digitalwonderland/elasticsearch     Latest Elasticsearch with Marvel & Kibana       12                   [OK]
monsantoco/elasticsearch            ElasticSearch Docker image                      9                    [OK]
```

Не удивительно, но существуют официальный образ для Elasticsearch. Чтобы запустить ES, нужно всего лишь выполнить docker run, и вскоре у нас будет локальный, работающий контейнер с одним узлом ES.

```
$ docker run -dp 9200:9200 elasticsearch
d582e031a005f41eea704cdc6b21e62e7a8a42021297ce7ce123b945ae3d3763
```

Если образ Elasticsearch не загружается, попробуйте запустить образ с конкретной версией, например:

```
$ docker run -dp 9200:9200 elasticsearch:6.8.3
d582e031a005f41eea704cdc6b21e62e7a8a42021297ce7ce123b945ae3d3763
```

Проверим, что образ запустился и отвечает на запросы

```
$ curl 127.0.0.1:9200
{
  "name" : "Ultra-Marine",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.1.1",
    "build_hash" : "40e2c53a6b6c2972b3d13846e450e66f4375bd71",
    "build_timestamp" : "2015-12-15T13:05:55Z",
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}
```

Заодно давайте запустим контейнер с Flask. Для начала необходимо создать клон репозитория с приложением, расположенном по адресу https://github.com/prakhar1989/FoodTrucks на вашем устройстве.

Рассмотрим, как там создан Dockerfile. В прошлой секции мы использовали образ python:3-onbuild в качестве базового. Однако, в этом раз, кроме установки зависимостей через pip, нам нужно, чтобы приложение генерировало минимизированный Javascript-файл для продакшена. Для этого понадобится Nodejs. Так что нужен свой билд с нуля, поэтому начнем с базового образа ubuntu.

Замечание: если оказывается, что существующий образ не подходит для вашей задачи, то спокойно создавайте свой образ на основе другого базового образа. В большинстве случаем, для образов на Docker Hub можно найти соответствующий Dockerfile на Github. Почитайте существующие Докерфайлы — это один из лучших способов научиться делать свои образы.

Наш Dockerfile для Flask-приложения выглядит следующим образом:

```
# start from base
FROM ubuntu:14.04
MAINTAINER Prakhar Srivastav <prakhar@prakhar.me>

# install system-wide deps for python and node
RUN apt-get -yqq update
RUN apt-get -yqq install python-pip python-dev
RUN apt-get -yqq install nodejs npm
RUN ln -s /usr/bin/nodejs /usr/bin/node

# copy our application code
ADD flask-app /opt/flask-app
WORKDIR /opt/flask-app

# fetch app specific deps
RUN npm install
RUN npm run build
RUN pip install -r requirements.txt

# expose port
EXPOSE 5000

# start app
CMD [ "python", "./app.py" ]
```

Тут много всего нового. Вначале указан базовый образ Ubuntu LTS, потом используется пакетный менеджер apt-get для установки зависимостей, в частности — Python и Node. Флаг yqq нужен для игнорирования вывода и автоматического выбора "Yes" во всех местах. Также создается символическая ссылка для бинарного файла node. Это нужно для решения проблем обратной совместимости.

Потом мы используем команду ADD для копирования приложения в нужную директорию в контейнере — /opt/flask-app. Здесь будет находиться весь наш код. Мы также

устанавливаем эту директорию в качестве рабочей, так что следующие команды будут выполняться в контексте этой локации. Теперь, когда наши системные зависимости установлены, пора установить зависимости уровня приложения. Начнем с Node, установки пакетов из npm и запуска команды сборки, как указано в нашем файле package.json. В конце устанавливаем пакеты Python, открываем порт и определяем запуск приложения с помощь CMD, как в предыдущем разделе.

Наконец, можно собрать образ и запустить контейнер. Перейдите в папку, где расположен Dockerfile и запустите сбор образа, присвоив ему имя формата «<ваш логин DockerHub>/ foodtrucks-web». Не забудьте точку в конце. Она означает что создание образа будет происходить из корня того каталога, в котором вы находитесь. Создание образа займет значительное время.

```
$ docker build --tag <ваш_логин>/foodtrucks-web .
```

Если вы не указали имя образа через -t, то безымянный образ можно запустить, узнав его ID через команду docker images.

```
C:\Users\name\Documents\Docker> docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
name/foodtrucks-web     latest              8da52c6ad837        18 seconds ago      572MB
ubuntu                  latest              2ca708c1c9cc        3 weeks ago         64.2MB
elasticsearch           6.8.3               1d0fd79266e6        5 weeks ago         800MB
```

При первом запуске нужно будет больше времени, так как клиент Докера будет скачивать образ ubuntu, запускать все команды и готовить образ. Повторный запуск docker build после последующих изменений будет практически моментальным. Давайте попробуем запустить приложение

```
$ docker run -P <ваш_логин>/foodtrucks-web
Unable to connect to ES. Retying in 5 secs...
Unable to connect to ES. Retying in 5 secs...
Unable to connect to ES. Retying in 5 secs...
Out of retries. Bailing out...
```

Упс! Наше приложение не смогло запуститься, потому что оно не может подключиться к Elasticsearch. Как сообщить одному контейнеру о другом и как заставить их взаимодействовать друг с другом? Ответ — в следующей секции.

1.1.2 Сети Docker

Перед тем, как обсудить возможности Докера для решения описанной задачи, давайте посмотрим на возможные варианты обхода проблемы. Думаю, это поможет нам оценить удобство той функциональности, которую мы вскоре изучим.

Ладно, давайте запустим docker ps, что тут у нас:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
e931ab24dedc        elasticsearch       "/docker-entrypoint.s" 2 seconds ago       Up 2 seconds 0.0.0.0:9200->9200/tcp, 9
300/tcp   cocky_spence
```

Итак, у нас есть контейнер ES по адресу и порту 0.0.0.0:9200, и мы можем напрямую обращаться к нему. Если можно было бы сообщить нашему приложению подключаться к этому адресу, то оно сможет общаться с ES, верно? Давайте взглянем на код на Питоне нашего сервера (\flask-app\app.py), где описано подключение (строка 8).

```
es = Elasticsearch(host='es')
```

Нужно сообщить Flask-контейнеру, что контейнер ES запущен на хосте 0.0.0.0 (порт по умолчанию 9200), и все заработает, да? К сожалению, нет, потому что IP 0.0.0.0 это адрес для доступа к контейнеру с **хост-машины**, то есть с моего компьютера. Другой контейнер

не сможет обратиться по этому адресу. Ладно, если не этот адрес, то какой другой адрес нужно использовать для работы с контейнером ES?

Это хороший момент, чтобы изучить работу сети в Докере. После установки Докер автоматически создает три сети:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
075b9f628ccc        none                null
be0f7178486c        host                host
8022115322ec        bridge              bridge
```

Сеть **bridge** — это сеть, в которой контейнеры запущены по умолчанию. Это значит, что когда я запускаю контейнер ES, он работает в этой сети bridge. Чтобы удостовериться, давайте проверим:

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "8022115322ec80613421b0282e7ee158ec41e16f565a3e86fa53496105deb2d7",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
        },
        "Containers": {
            "e931ab24dedc1640cddf6286d08f115a83897c88223058305460d7bd793c1947": {
                "EndpointID": "66965e83bf7171daeb8652b39590b1f8c23d066ded16522daeb0128
c9c25c189",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        }
    }
]
```

Видно, что контейнер e931ab24dedc находится в секции Containers. Также виден IP-адрес, выданный этому контейнеру — 172.17.0.2. Именно этот адрес мы и искали? Давайте проверим: запустим Flask-приложение и попробуем обратиться к нему по IP с помощью команды curl изнутри контейнера (если curl отсутствует в контейнере, то его придется установить):

```
$ docker run -it --rm <ваше имя>/foodtrucks-web bash
root@35180ccc206a:/opt/flask-app# curl 172.17.0.2:9200
bash: curl: command not found
root@35180ccc206a:/opt/flask-app# apt-get -yqq install curl
root@35180ccc206a:/opt/flask-app# curl 172.17.0.2:9200
{
  "name" : "Jane Foster",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.1.1",
    "build_hash" : "40e2c53a6b6c2972b3d13846e450e66f4375bd71",
    "build_timestamp" : "2015-12-15T13:05:55Z",
```

```
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}
root@35180ccc206a:/opt/flask-app# exit
```

Сейчас все должно быть понятно. Мы запустили контейнер в интерактивном режиме с процессом bash. Флаг --rm нужен для удобства, благодаря нему контейнер автоматически удаляется после выхода. Мы попробуем curl, но нужно сначала установить его. После этого можно удостовериться, что по адресу 172.17.0.2:9200 на самом деле можно обращаться к ES! Супер!

Несмотря на то, что мы нашли способ наладить связь между контейнерами, существует несколько проблем с этим подходом:

1. Придется добавлять записи в файл `/etc/hosts` внутри Flask-контейнера, чтобы приложение понимало, что имя хоста `es` означает `172.17.0.2`. Если IP-адрес меняется, то придется вручную менять запись.
2. Так как сеть *bridge* используется всеми контейнерами по умолчанию, этот метод **не безопасен**.

Но есть хорошие новости: в Докере есть отличное решение этой проблемы. Докер позволяет создавать собственные изолированные сети. Это решение также помогает справиться с проблемой /etc/hosts, сейчас увидим как.

Во-первых, давайте создадим свою сеть:

```
$ docker network create foodtrucks
1a3386375797001999732cb4c4e97b88172d983b08cd0addfcb161eed0c18d89

$ docker network ls
NETWORK ID          NAME                DRIVER
1a3386375797        foodtrucks          bridge
8022115322ec        bridge              bridge
075b9f628ccc        none                null
be0f7178486c        host                host
```

Команда network create создает новую сеть *bridge*. Нам сейчас нужен именно такой тип. Существуют другие типы сетей, и вы можете почитать о них в официальной документации.

Теперь у нас есть сеть. Можно запустить наши контейнеры внутри сети с помощью флага --net. Давайте так и сделаем, но сначала остановим контейнер с ElasticSearch, который был запущен в сети bridge по умолчанию.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             S
TATUS              PORTS                              NAMES
e931ab24dedc        elasticsearch       "/docker-entrypoint.s"   4 hours ago         U
p 4 hours          0.0.0.0:9200->9200/tcp, 9300/tcp   cocky_spence

$ docker stop e931ab24dedc
e931ab24dedc

$ docker run -dp 9200:9200 --net foodtrucks --name es elasticsearch
2c0b96f9b8030f038e40abea44c2d17b0a8edda1354a08166c33e6d351d0c651

$ docker network inspect foodtrucks
[
    {
        "Name": "foodtrucks",
        "Id": "1a3386375797001999732cb4c4e97b88172d983b08cd0addfcb161eed0c18d89",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
```

```
            "Config": [
                {}
            ]
        },
        "Containers": {
            "2c0b96f9b8030f038e40abea44c2d17b0a8edda1354a08166c33e6d351d0c651": {
                "EndpointID": "15eabc7989ef78952fb577d0013243dae5199e8f5c55f1661606077
d5b78e72a",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

Мы сделали то же, что и раньше, но на этот раз дали контейнеру название es. Перед тем, как запускать контейнер с приложением, давайте проверим что происходит, когда запуск происходит в сети.

```
$ docker run -it --rm --net foodtrucks <ваше имя>/foodtrucks-web bash

root@53af252b771a:/opt/flask-app# curl es:9200
bash: curl: command not found

root@53af252b771a:/opt/flask-app# apt-get -yqq install curl

root@53af252b771a:/opt/flask-app# curl es:9200
{
  "name" : "Doctor Leery",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.1.1",
    "build_hash" : "40e2c53a6b6c2972b3d13846e450e66f4375bd71",
    "build_timestamp" : "2015-12-15T13:05:55Z",
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}

root@53af252b771a:/opt/flask-app# ls
app.py  node_modules  package.json  requirements.txt  static  templates  webpack.confi
g.js

root@53af252b771a:/opt/flask-app# python app.py
Index not found...
Loading data in elasticsearch ...
Total trucks loaded:  733
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

root@53af252b771a:/opt/flask-app# exit
```

Магическим образом (посредством встроенного DNS-сервера, на самом деле) Докер сделал имя нашего образа «es» доступным в среде другого контейнера, и поэтому es:9200можно использовать в приложении — этот адрес корректно направляет запросы в контейнер ES. Отлично! Давайте теперь запустим Flask-контейнер по-настоящему:

```
$ docker run -d --net foodtrucks -p 5000:5000 --name foodtrucks-web <ваше имя>/foodtru
cks-web
2a1b77e066e646686f669bab4759ec1611db359362a031667cacbe45c3ddb413

$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED
STATUS              PORTS                              NAMES
```

```
2a1b77e066e6        prakhar1989/foodtrucks-web   "python ./app.py"        2 seconds ag
o       Up 1 seconds        0.0.0.0:5000->5000/tcp             foodtrucks-web
2c0b96f9b803        elasticsearch                "/docker-entrypoint.s"   21 minutes a
go      Up 21 minutes       0.0.0.0:9200->9200/tcp, 9300/tcp   es

$ curl -I 0.0.0.0:5000
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 3697
Server: Werkzeug/0.11.2 Python/2.7.6
Date: Sun, 10 Jan 2016 23:58:53 GMT
```

Зайдите на http://0.0.0.0:5000, и увидите приложение в работе. Опять же, может показаться, что было много работы, но на самом деле мы ввели всего 4 команды чтобы с нуля дойти до работающего приложения. Я собрал эти команды в bash-скрипт.

```
#!/bin/bash

# build the flask container
docker build -t <ваше имя>/foodtrucks-web .

# create the network
docker network create foodtrucks

# start the ES container
docker run -d --net foodtrucks -p 9200:9200 -p 9300:9300 --name es elasticsearch

# start the flask app container
docker run -d --net foodtrucks -p 5000:5000 --name foodtrucks-web <ваше имя>/foodtruck
s-web
```

Теперь представьте, что хотите поделиться приложением с другом. Или хотите запустить на сервере, где установлен Докер. Можно запустить всю систему с помощью одной команды!

```
$ git clone https://github.com/prakhar1989/FoodTrucks
$ cd FoodTrucks
$ ./setup-docker.sh
```

Вот и все! По-моему, это невероятно крутой и мощный способ распространять и запускать приложения!

1.1.3 Docker Compose

До этого момента мы изучали клиент Докера. Но в экосистеме Докера есть несколько других инструментов с открытым исходным кодом, которые хорошо взаимодействуют с Докером. Некоторые из них это:

1. Docker Machine позволяет создавать Докер-хосты на своем компьютере, облачном провайдере или внутри дата-центра.
2. Docker Compose — инструмент для определения и запуска много-контейнерных приложений.
3. Docker Swarm — нативное решение для кластеризации.

В этом разделе мы поговорим об одном из этих инструментов — Docker Compose, и узнаем, как он может упростить работу с несколькими контейнерами.

У Docker Compose довольно интересная предыстория. Примерно два года назад компания OrchardUp запустила инструмент под названием Fig. Идея была в том, чтобы создавать изолированные рабочие окружения с помощью Докера. Проект очень хорошо восприняли на Hacker News - я смутно помню, что читал о нем, но не особо понял его смысла.

Первый комментарий на самом деле неплохо объясняет, зачем нужен Fig и что он делает.

На самом деле, смысл Докера в следующем: запускать процессы. Сегодня у Докера есть неплохое API для запуска процессов: расшаренные между контейнерами (иными словами, запущенными образами) разделы или директории (shared volumes), перенаправление портов с хост-машины в контейнер, вывод логов, и так далее.

Но больше ничего: Докер сейчас работает только на уровне процессов. Несмотря на то, что в нем содержатся некоторые возможности оркестрации нескольких контейнеров для создания единого "приложения", в Докере нет ничего, что помогало бы с управлением такими группами контейнеров как одной сущностью. И вот зачем нужен инструмент вроде Fig: чтобы обращаться с группой контейнеров как с единой сущностью. Чтобы думать о "запуске приложений" (иными словами, "запуске оркестрированного кластера контейнеров") вместо "запуска контейнеров".

Оказалось, что многие пользователи Докера согласны с такими мыслями. Постепенно, Fig набрал популярность, Docker Inc. заметили, купили компанию и назвали проект Docker Compose.

Итак, зачем используется *Compose*? Это инструмент для простого определения и запуска многоконтейнерных Докер-приложений. В нем есть файл docker-compose.yml, и с его помощью можно одной командой поднять приложение с набором сервисов.

Давайте посмотрим, сможем ли мы создать файл docker-compose.yml для нашего приложения SF-Foodtrucks и проверим, способен ли он на то, что обещает.

Но вначале нужно установить Docker Compose. Есть у вас Windows или Mac, то Docker Compose уже установлен — он идет в комплекте с Docker Toolbox. На Linux можно установить Docker Compose следуя простым инструкциям на сайте документации. Compose написан на Python, поэтому можно сделать просто pip install docker-compose. Проверить работоспособность так:

```
$ docker-compose version
docker-compose version 1.7.1, build 0a9ab35
docker-py version: 1.8.1
CPython version: 2.7.9
OpenSSL version: OpenSSL 1.0.1j 15 Oct 2014
```

Теперь можно перейти к следующему шагу, то есть созданию файла docker-compose.yml. Синтаксис yml-файлов очень простой, и в репозитории уже есть пример, который мы будем использовать (файл docker-compose.yml)

```
version: "2"
services:
  es:
    image: elasticsearch
  web:
    image: prakhar1989/foodtrucks-web
    command: python app.py
    ports:
      - "5000:5000"
    volumes:
      - .:/code
```

На родительском уровне мы задали название неймспейса для наших сервисов: es и web. К каждому сервису можно добавить дополнительные параметры, среди которых image— обязательный. Для es мы указываем доступный на Docker Hub образ elasticsearch. Для Flask-приложения — тот образ, который мы создали самостоятельно в начале этого раздела.

С помощью других параметров вроде command и ports можно предоставить информацию о контейнере. volumes отвечает за локацию монтирования, где будет находиться код в контейнере web. Это опциональный параметр, он полезен, если нужно обращаться к логам и так далее. Подробнее о параметрах и возможных значениях можно прочитать в документации.

Замечание: Нужно находиться в директории с файлом docker-compose.yml чтобы запускать большую часть команд Compose.

Отлично! Файл готов, давайте посмотрим на docker-compose в действии. Но вначале нужно удостовериться, что порты свободны. Так что если у вас запущены контейнеры Flask и ES, то пора их остановить:

```
$ docker stop $(docker ps -q)
39a2f5df14ef
2a1b77e066e6
```

Теперь можно запускать docker-compose. Перейдите в директорию с приложением Foodtrucks и выполните команду docker-compose up.

```
$ docker-compose up
Creating network "foodtrucks_default" with the default driver
Creating foodtrucks_es_1
Creating foodtrucks_web_1
Attaching to foodtrucks_es_1, foodtrucks_web_1
es_1  | [2016-01-11 03:43:50,300][INFO ][node                     ] [Comet] version[2.
1.1], pid[1], build[40e2c53/2015-12-15T13:05:55Z]
es_1  | [2016-01-11 03:43:50,307][INFO ][node                     ] [Comet] initializi
ng ...
es_1  | [2016-01-11 03:43:50,366][INFO ][plugins                  ] [Comet] loaded [],
sites []
es_1  | [2016-01-11 03:43:50,421][INFO ][env                      ] [Comet] using [1]
data paths, mounts [[/usr/share/elasticsearch/data (/dev/sda1)]], net usable_space [16
gb], net total_space [18.1gb], spins? [possibly], types [ext4]
es_1  | [2016-01-11 03:43:52,626][INFO ][node                     ] [Comet] initialize
d
es_1  | [2016-01-11 03:43:52,632][INFO ][node                     ] [Comet] starting .
..
es_1  | [2016-01-11 03:43:52,703][WARN ][common.network           ] [Comet] publish ad
dress: {0.0.0.0} is a wildcard address, falling back to first non-loopback: {172.17.0.
2}
es_1  | [2016-01-11 03:43:52,704][INFO ][transport                ] [Comet] publish_ad
dress {172.17.0.2:9300}, bound_addresses {[::]:9300}
es_1  | [2016-01-11 03:43:52,721][INFO ][discovery                ] [Comet] elasticsea
rch/cEk4s7pdQ-evRc9MqS2wqw
es_1  | [2016-01-11 03:43:55,785][INFO ][cluster.service          ] [Comet] new_master
{Comet}{cEk4s7pdQ-evRc9MqS2wqw}{172.17.0.2}{172.17.0.2:9300}, reason: zen-disco-join(e
lected_as_master, [0] joins received)
es_1  | [2016-01-11 03:43:55,818][WARN ][common.network           ] [Comet] publish ad
dress: {0.0.0.0} is a wildcard address, falling back to first non-loopback: {172.17.0.
2}
es_1  | [2016-01-11 03:43:55,819][INFO ][http                     ] [Comet] publish_ad
dress {172.17.0.2:9200}, bound_addresses {[::]:9200}
es_1  | [2016-01-11 03:43:55,819][INFO ][node                     ] [Comet] started
es_1  | [2016-01-11 03:43:55,826][INFO ][gateway                  ] [Comet] recovered
[0] indices into cluster_state
es_1  | [2016-01-11 03:44:01,825][INFO ][cluster.metadata         ] [Comet] [sfdata] c
reating index, cause [auto(index api)], templates [], shards [5]/[1], mappings [truck]
es_1  | [2016-01-11 03:44:02,373][INFO ][cluster.metadata         ] [Comet] [sfdata] u
pdate_mapping [truck]
es_1  | [2016-01-11 03:44:02,510][INFO ][cluster.metadata         ] [Comet] [sfdata] u
pdate_mapping [truck]
es_1  | [2016-01-11 03:44:02,593][INFO ][cluster.metadata         ] [Comet] [sfdata] u
pdate_mapping [truck]
es_1  | [2016-01-11 03:44:02,708][INFO ][cluster.metadata         ] [Comet] [sfdata] u
pdate_mapping [truck]
es_1  | [2016-01-11 03:44:03,047][INFO ][cluster.metadata         ] [Comet] [sfdata] u
pdate_mapping [truck]
```

```
web_1 |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Перейдите по IP чтобы увидеть приложение. Круто, да? Всего лишь пара строк конфигурации и несколько Докер-контейнеров работают в унисон. Давайте остановим сервисы и перезапустим в detached mode:

```
web_1 |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
Killing foodtrucks_web_1 ... done
Killing foodtrucks_es_1 ... done

$ docker-compose up -d
Starting foodtrucks_es_1
Starting foodtrucks_web_1

$ docker-compose ps
      Name                     Command               State            Ports
-------------------------------------------------------------------------------------
foodtrucks_es_1    /docker-entrypoint.sh elas ...   Up      9200/tcp, 9300/tcp
foodtrucks_web_1   python app.py                    Up      0.0.0.0:5000->5000/tcp
```

Не удивительно, но оба контейнера успешно запущены. Откуда берутся имена? Их Compose придумал сам. Но что насчет сети? Его Compose тоже делаем сам? Хороший вопрос, давайте выясним.

Для начала, остановим запущенные сервисы. Их всегда можно вернуть одной командой:

```
$ docker-compose stop
Stopping foodtrucks_web_1 ... done
Stopping foodtrucks_es_1 ... done
```

Заодно, давайте удалим сеть foodtrucks, которую создали в прошлый раз. Эта сеть нам не потребуется, потому что *Compose* автоматически сделает все за нас.

```
$ docker network rm foodtrucks
$ docker network ls
NETWORK ID          NAME                DRIVER
4eec273c054e        bridge              bridge
9347ae8783bd        none                null
54df57d7f493        host                host
```

Класс! Теперь в этом чистом состоянии можно проверить, способен ли *Compose* на волшебство.

```
$ docker-compose up -d
Recreating foodtrucks_es_1
Recreating foodtrucks_web_1
$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED
STATUS              PORTS                     NAMES
f50bb33a3242        prakhar1989/foodtrucks-web   "python app.py"          14 seconds a
go      Up 13 seconds       0.0.0.0:5000->5000/tcp   foodtrucks_web_1
e299ceeb4caa        elasticsearch                "/docker-entrypoint.s"   14 seconds a
go      Up 14 seconds       9200/tcp, 9300/tcp       foodtrucks_es_1
```

Пока все хорошо. Проверим, создались ли какие-нибудь сети:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
0c8b474a9241        bridge              bridge
293a141faac3        foodtrucks_default  bridge
b44db703cd69        host                host
0474c9517805        none                null
```

Видно, что Compose самостоятельно создал сеть foodtrucks_default и подсоединил оба сервиса в эту сеть, так, чтобы они могли общаться друг с другом. Каждый контейнер для сервиса подключен к сети, и оба контейнера доступны другим контейнерам в сети. Они доступны по hostname, который совпадает с названием контейнера. Давайте проверим, находится ли эта информация в /etc/hosts.

```
$ docker ps
```

```
CONTAINER ID        IMAGE                          COMMAND                  CREATED
STATUS              PORTS                      NAMES
bb72dcebd379        prakhar1989/foodtrucks-web   "python app.py"          20 hours ago
Up 19 hours         0.0.0.0:5000->5000/tcp   foodtrucks_web_1
3338fc79be4b        elasticsearch                "/docker-entrypoint.s"   20 hours ago
Up 19 hours         9200/tcp, 9300/tcp       foodtrucks_es_1

$ docker exec -it bb72dcebd379 bash
root@bb72dcebd379:/opt/flask-app# cat /etc/hosts
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.18.0.2  bb72dcebd379
```

Упс! Оказывается, файл понятия не имеет о es. Как же наше приложение работает? Давайте попингуем его по названию хоста:

```
root@bb72dcebd379:/opt/flask-app# ping es
PING es (172.18.0.3) 56(84) bytes of data.
64 bytes from foodtrucks_es_1.foodtrucks_default (172.18.0.3): icmp_seq=1 ttl=64 time=
0.049 ms
64 bytes from foodtrucks_es_1.foodtrucks_default (172.18.0.3): icmp_seq=2 ttl=64 time=
0.064 ms
^C
--- es ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.049/0.056/0.064/0.010 ms
```

Вуаля! Работает! Каким-то магическим образом контейнер смог сделать пинг хоста es. Оказывается, Docker 1.10 добавили новую сетевую систему, которая производит обнаружение сервисов через DNS-сервер. Если интересно, то почитайте подробнее о предложении и release notes.

На этом наш тур по Docker Compose завершен. С этим инструментом можно ставить сервисы на паузу, запускать отдельные команды в контейнере и даже масштабировать систему, то есть увеличивать количество контейнеров. Также советую изучать некоторые другие примеры использования Docker Compose.

Надеюсь, я продемонстрировал как на самом деле просто управлять многоконтейнерной средой с Compose. В последнем разделе мы задеплоим все на AWS!