



Научитесь легко создавать и развертывать свои распределенные приложения в облаке с помощью Docker

Написано и разработано [Прахаром Шриваставом](#)



[Star](#) 5,402

## ВВЕДЕНИЕ

Что такое Docker?

Википедия определяет [Docker](#) как

*проект с открытым исходным кодом, который автоматизирует развертывание программных приложений внутри контейнеров путем предоставления дополнительного уровня абстракции и автоматизации виртуализации на уровне операционной системы в Linux.*

Вау! Это полный бред. Проще говоря, Docker - это инструмент, который позволяет разработчикам, системным администраторам и т.д. чтобы легко развертывать свои приложения в изолированной

среде (называемой *контейнерами*) для запуска в операционной системе хоста, то есть Linux. Ключевым преимуществом Docker является то, что он позволяет пользователям **упаковать приложение со всеми его зависимостями в стандартизованный модуль** для разработки программного обеспечения. В отличие от виртуальных машин, контейнеры не требуют больших накладных расходов и, следовательно, позволяют более эффективно использовать базовую систему и ресурсы.

## Что такое контейнеры?

Сегодня отраслевым стандартом является использование виртуальных машин (VM) для запуска программных приложений. Виртуальные машины запускают приложения внутри гостевой операционной системы, которая работает на виртуальном оборудовании, работающем от основной операционной системы сервера.

Виртуальные машины отлично подходят для обеспечения полной изоляции процессов приложений: существует очень мало способов, которыми проблема в операционной системе хоста может повлиять на программное обеспечение, работающее в гостевой операционной системе, и наоборот. Но такая изоляция обходится дорого – вычислительные затраты, затрачиваемые на виртуализацию оборудования для использования гостевой ОС, значительны.

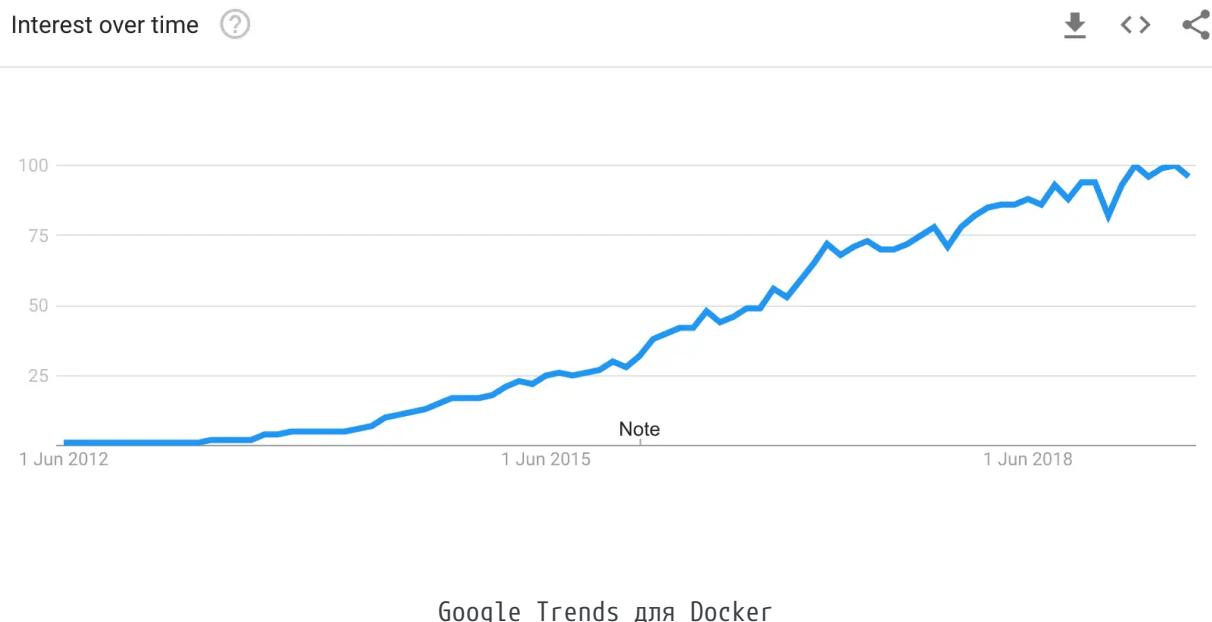
В контейнерах используется другой подход: используя низкоуровневую механику операционной системы хоста, контейнеры обеспечивают большую часть изоляции виртуальных машин при незначительной вычислительной мощности.

## Зачем использовать контейнеры?

Контейнеры предлагают логический механизм упаковки, с помощью которого приложения могут быть абстрагированы от среды, в

которой они фактически выполняются. Такое разделение позволяет легко и последовательно развертывать приложения на основе контейнеров, независимо от того, является ли целевая среда частным центром обработки данных, общедоступным облаком или даже персональным ноутбуком разработчика. Это дает разработчикам возможность создавать предсказуемые среды, которые изолированы от остальных приложений и могут запускаться где угодно.

С точки зрения операций, помимо мобильности, контейнеры также предоставляют более детальный контроль над ресурсами, повышая эффективность вашей инфраструктуры, что может привести к лучшему использованию ваших вычислительных ресурсов.



Благодаря этим преимуществам контейнеры (и Docker) получили широкое распространение. Такие компании, как Google, Facebook, Netflix и Salesforce, используют контейнеры для повышения производительности больших инженерных групп и улучшения использования вычислительных ресурсов. Фактически, Google приписал контейнеры тому, что они избавили от необходимости в целом центре обработки данных.

## Чему меня научит этот учебник?

Цель этого руководства - стать универсальным средством для того, чтобы не запачкать руки Docker. Помимо демистификации ландшафта Docker, это даст вам практический опыт создания и развертывания ваших собственных веб-приложений в облаке. Мы будем использовать [Amazon Web Services](#) для развертывания статического веб-сайта и двух динамических веб-приложений на [EC2](#) с использованием [Elastic Beanstalk](#) и [Elastic Container Service](#). Даже если у вас нет предварительного опыта развертывания, этого руководства должно быть достаточно, чтобы начать работу.

## ПРИСТУПАЯ К РАБОТЕ

Этот документ содержит серию из нескольких разделов, каждый из которых объясняет определенный аспект Docker. В каждом разделе мы будем вводить команды (или писать код). Весь код, используемый в руководстве, доступен в [репозитории Github](#).

*Примечание: В этом руководстве используется версия Docker 18.05.0-ce. Если вы обнаружите, что какая-либо часть руководства несовместима с будущей версией, пожалуйста, поднимите [вопрос](#). Спасибо!*

## Предварительные требования

В этом учебном пособии не требуется никаких специальных навыков, кроме элементарного удобства работы с командной строкой и текстовым редактором. В этом учебном пособии используется `git clone` для локального клонирования репозитория. Если в вашей системе не установлен Git, либо

установите его, либо не забудьте вручную загрузить zip-файлы с Github. Предыдущий опыт разработки веб-приложений будет полезен, но не обязателен. По мере прохождения руководства мы будем использовать несколько облачных сервисов. Если вы заинтересованы в дальнейшем, пожалуйста, создайте учетную запись на каждом из этих веб-сайтов:

- Веб-сервисы Amazon
- Docker Hub

## Настройка вашего компьютера

Настройка всех инструментов на вашем компьютере может оказаться непростой задачей, но, к счастью, поскольку Docker стал стабильным, запустить Docker в вашей любимой ОС стало очень просто.

Еще несколько выпусков назад запуск Docker в OSX и Windows был довольно сложной задачей. Однако в последнее время Docker значительно инвестировала в улучшение взаимодействия своих пользователей с этими операционными системами, поэтому запустить Docker сейчас - проще простого. Руководство по *началу работы* по Docker содержит подробные инструкции по настройке Docker на [Mac](#), [Linux](#) и [Windows](#).

После завершения установки Docker протестируйте установку Docker, выполнив следующее:

```
$ docker run hello-world
```

Hello from Docker.

This message shows that your installation appears to be working correctly.

...

# ПРИВЕТ, МИР

## Играем с Busybox

Теперь, когда у нас все настроено, пришло время запачкать руки. В этом разделе мы собираемся запустить контейнер **Busybox** в нашей системе и познакомиться с `docker run` командой.

Чтобы начать, давайте запустим следующее в нашем терминале:

```
$ docker pull busybox
```

*Примечание: В зависимости от того, как вы установили docker в своей системе, вы можете увидеть permission denied ошибку после выполнения приведенной выше команды. Если вы используете Mac, убедитесь, что движок Docker запущен. Если вы используете Linux, то добавляйте в свои docker команды префикс sudo . В качестве альтернативы вы можете создать группу docker, чтобы избавиться от этой проблемы.*

`pull` Команда извлекает **образ** busybox из **реестра Docker** и сохраняет его в нашей системе. Вы можете использовать команду `docker images`, чтобы просмотреть список всех изображений в вашей системе.

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED
busybox             latest   c51f86c28340  4 weeks ago
```

## Запуск Docker

Отлично! Теперь давайте запустим **контейнер Docker** на основе этого изображения. Для этого мы собираемся использовать

команду всемогущий `docker run`.

```
$ docker run busybox
$
```

Подождите, ничего не произошло! Это ошибка? Ну, нет. За кулисами произошло много всего. При вызове `run` клиент Docker находит изображение (в данном случае `busybox`), загружает контейнер и затем запускает команду в этом контейнере. При запуске `docker run busybox` мы не предоставили команду, поэтому контейнер загрузился, выполнил пустую команду и затем вышел. Ну, да - своего рода облом. Давайте попробуем что-нибудь более захватывающее.

```
$ docker run busybox echo "hello from busybox"
hello from busybox
```

Приятно - наконец-то мы видим некоторые выходные данные. В этом случае клиент Docker послушно выполнил `echo` команду в нашем контейнере `busybox`, а затем вышел из него. Если вы заметили, все это произошло довольно быстро. Представьте, что вы загружаете виртуальную машину, запускаете команду и затем завершаете ее работу. Теперь вы знаете, почему говорят, что контейнеры - это быстро! Хорошо, теперь пришло время ознакомиться с `docker ps` командой. Команда `docker ps` показывает вам все контейнеры, которые запущены в данный момент.

CONTAINER ID	IMAGE	COMMAND	CREATED

Поскольку контейнеры не запущены, мы видим пустую строку. Давайте попробуем более полезный вариант: `docker ps -a`

CONTAINER ID	IMAGE	COMMAND	CREATED
305297d7a235	busybox	"uptime"	11 minutes ago

 ff0a5c3750b9	busybox	"sh"	12 minutes ago
14e5bd11d164	hello-world	"/hello"	2 minutes ago

Итак, то, что мы видим выше, - это список всех контейнеров, которые мы запускали. Обратите внимание, что столбец **STATUS** показывает, что эти контейнеры завершили работу несколько минут назад.

Вам, вероятно, интересно, есть ли способ запустить в контейнере не только одну команду. Давайте попробуем это сейчас.:

```
$ docker run -it busybox sh
/ # ls
bin dev etc home proc root sys tmp usr var
/ # uptime
05:45:21 up 5:58, 0 users, load average: 0.00, 0.01, 0.04
```

Выполнение `run` команды с `-it` флагами привязывает нас к интерактивному `tty` в контейнере. Теперь мы можем запускать в контейнере столько команд, сколько захотим. Потратьте некоторое время на выполнение ваших любимых команд.

**Опасная зона:** Если вам хочется приключений, вы можете попробовать `rm -rf bin` в контейнере. Убедитесь, что вы запускаете эту команду в контейнере, а **не** на своем ноутбуке / десктопе. При выполнении этого любые другие команды, такие как `ls`, `uptime` не будут работать. Как только все перестанет работать, вы можете выйти из контейнера (введите `exit` и нажмите `Enter`), а затем запустить его снова с помощью `docker run -it busybox sh` команды. Поскольку `Docker` каждый раз создает новый контейнер, все должно начать работать заново.

На этом завершается увлекательное знакомство с командой `mighty docker run`, которую, скорее всего, вы будете использовать чаще всего. Имеет смысл потратить некоторое время на освоение

с ней. Чтобы узнать больше о `run`, используйте `docker run --help`, чтобы просмотреть список всех поддерживаемых им флагов. По мере продвижения мы увидим еще несколько вариантов `docker run`.

Прежде чем мы двинемся дальше, давайте быстро поговорим об удалении контейнеров. Мы видели выше, что мы все еще можем видеть остатки контейнера даже после выхода из него с помощью запуска `docker ps -a`. На протяжении всего этого руководства вы будете запускать `docker run` несколько раз, и оставление ненужных контейнеров будет занимать место на диске.

Следовательно, как правило, я очищаю контейнеры, как только заканчиваю с ними. Чтобы сделать это, вы можете запустить `docker rm` команду. Просто скопируйте идентификаторы контейнеров сверху и вставьте их рядом с командой.

```
$ docker rm 305297d7a235 ff0a5c3750b9  
305297d7a235  
ff0a5c3750b9
```

При удалении вы увидите, что идентификаторы будут возвращены вам. Если вам нужно удалить кучу контейнеров за один раз, копирование идентификаторов может быть утомительным. В этом случае вы можете просто запустить -

```
$ docker rm $(docker ps -a -q -f status=exited)
```

Эта команда удаляет все контейнеры со статусом `exited`. На случай, если вам интересно, `-q` флаг возвращает только числовые идентификаторы и `-f` фильтрует выходные данные на основе предоставленных условий. Последнее, что будет полезно, - это `--rm` флаг, который можно передать `docker run`, который автоматически удаляет контейнер после выхода из него. Для одноразовых запусков `docker --rm` flag очень полезен.

В более поздних версиях Docker для достижения того же эффекта можно использовать команду `docker container prune`.

```
$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
4a7f7eebae0f63178aff7eb0aa39f0627a203ab2df258c1a00b456cf20063
f98f9c2aa1eaf727e4ec9c0283bcaa4762fdbda7f26191f26c97f64090360

Total reclaimed space: 212 B
```

Наконец, вы также можете удалить изображения, которые вам больше не нужны, запустив `docker rmi`.

## Терминология

В последнем разделе мы использовали много специфичного для Docker жаргона, который может кого-то сбить с толку. Итак, прежде чем мы пойдем дальше, позвольте мне прояснить некоторую терминологию, которая часто используется в экосистеме Docker.

- *Изображения* - схемы нашего приложения, которые составляют основу контейнеров. В приведенной выше демонстрации мы использовали `docker pull` команду для загрузки образа `busybox`.
- *Контейнеры* - Создаем из образов Docker и запускаем само приложение. Создаем контейнер, используя `docker run` что мы и сделали, используя загруженный образ `busybox`. Список запущенных контейнеров можно просмотреть с помощью команды `docker ps`.
- *Docker Daemon* - фоновая служба, работающая на хосте, которая управляет сборкой, запуском и распространением контейнеров Docker. Демон - это процесс, который выполняется в операционной системе, с которой взаимодействуют клиенты.
- *Docker Client* - инструмент командной строки, который позволяет пользователю взаимодействовать с демоном. В более общем плане могут существовать и другие формы



клиентов, такие как [Kitematic](#), которые предоставляют пользователям графический интерфейс.

- *Docker Hub* - [реестр](#) образов Docker. Реестр можно представить как каталог всех доступных образов Docker. При необходимости можно разместить свои собственные реестры Docker и использовать их для извлечения изображений.
- 

## ВЕБ-ПРИЛОЖЕНИЯ С DOCKER

Отлично! Итак, мы рассмотрели `docker run`, поиграли с контейнером Docker, а также освоили некоторую терминологию. Вооруженные всеми этими знаниями, мы теперь готовы приступить к реальным действиям, то есть к развертыванию веб-приложений с помощью Docker!

### Статические сайты

Давайте начнем с простых шагов. Первое, на что мы собираемся обратить внимание, это то, как мы можем запустить предельно простой статический веб-сайт. Мы собираемся извлечь образ Docker из Docker Hub, запустить контейнер и посмотреть, насколько легко запустить веб-сервер.

Давайте начнем. Изображение, которое мы собираемся использовать, представляет собой одностраничный [веб-сайт](#), который я уже создал для этой демонстрации и разместил в [реестре](#) - `prakhar1989/static-site`. Мы можем загрузить и запустить изображение непосредственно за один раз, используя `docker run`. Как отмечалось выше, `--rm` флаг автоматически удаляет контейнер при выходе, а `-it` флаг указывает

☰ интерактивный терминал, который упрощает завершение работы контейнера с помощью `Ctrl + C` (в Windows).

```
$ docker run --rm -it prakhar1989/static-site
```

Поскольку образ не существует локально, клиент сначала извлечет образ из реестра, а затем запустит образ. Если все пройдет нормально, вы должны увидеть `Nginx is running...` сообщение в вашем терминале. Хорошо, теперь, когда сервер запущен, как просмотреть веб-сайт? На каком порту он запущен? И, что более важно, как нам получить доступ к контейнеру непосредственно с нашего хост-компьютера? Нажмите `Ctrl + C`, чтобы остановить контейнер.

Что ж, в этом случае клиент не предоставляет никаких портов, поэтому нам нужно повторно запустить `docker run` команду для публикации портов. Пока мы этим занимаемся, нам также следует найти способ, чтобы наш терминал не был подключен к запущенному контейнеру. Таким образом, вы можете спокойно закрыть свой терминал и продолжить работу контейнера. Это называется **отстраненный** режим.

```
$ docker run -d -P --name static-site prakhar1989/static-site  
e61d12292d69556eabe2a44c16cbd54486b2527e2ce4f95438e504afb7b02810
```

В приведенной выше команде, `-d` отсоединит наш терминал, `-P` опубликует все открытые порты на случайные порты и, наконец, `--name` соответствует имени, которое мы хотим дать. Теперь мы можем увидеть порты, выполнив команду `docker port [CONTAINER]`

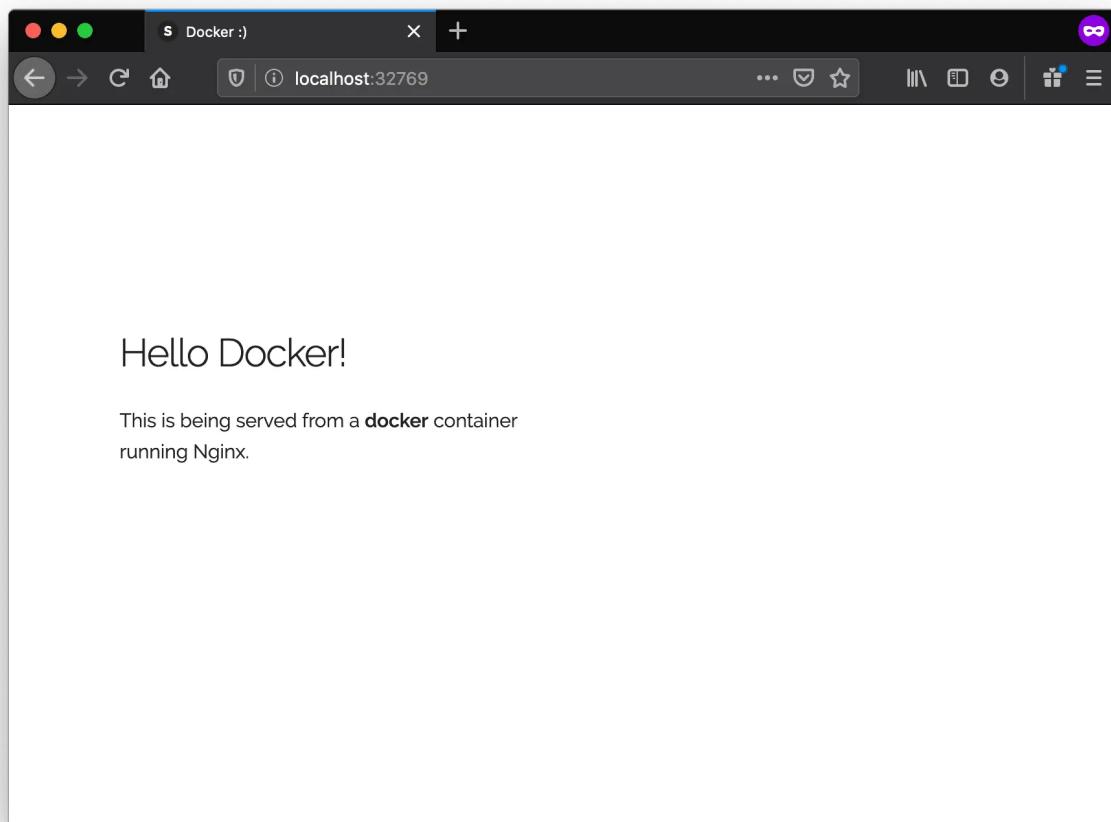
```
$ docker port static-site  
80/tcp -> 0.0.0.0:32769  
443/tcp -> 0.0.0.0:32768
```

Вы можете открыть <http://localhost:32769> в своем браузере.

Примечание: Если вы используете `docker-toolbox`, то вам может потребоваться использовать `docker-machine ip default` для получения IP-адреса.

Вы также можете указать пользовательский порт, на который клиент будет перенаправлять соединения с контейнером.

```
$ docker run -p 8888:80 prakhar1989/static-site  
Nginx is running...
```



Чтобы остановить отсоединенный контейнер, запустите его, `docker stop` указав идентификатор контейнера. В этом случае мы можем использовать имя, которое `static-site` мы использовали для запуска контейнера.

```
$ docker stop static-site  
static-site
```

Я уверен, вы согласитесь, что это было очень просто. Чтобы развернуть это на реальном сервере, вам просто нужно установить Docker и выполнить приведенную выше команду Docker. Теперь, когда вы увидели, как запустить веб-сервер внутри образа Docker, вам, должно быть, интересно - как мне создать свой собственный образ Docker? Этот вопрос мы рассмотрим в следующем разделе.

## Изображения Docker

Мы уже рассматривали изображения раньше, но в этом разделе мы углубимся в то, что такое изображения Docker, и создадим наш собственный образ! Наконец, мы также будем использовать этот образ для локального запуска нашего приложения и, наконец, для развертывания на AWS, чтобы поделиться им с друзьями! Рады? Отлично! Давайте начнем.

Изображения Docker являются основой контейнеров. В предыдущем примере мы **извлекли** образ *Busybox* из реестра и попросили клиент Docker запустить контейнер, **основанный** на этом образе. Чтобы просмотреть список изображений, доступных локально, используйте команду `docker images`.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATE
prakhar1989/catnip	latest	c7ffb5626a50	2 hour
prakhar1989/static-site	latest	b270625a1631	21 hours
python	3-onbuild	cf4002b2c383	5 days
martin/docker-cleanup-volumes	latest	b42990daaca2	7 weeks
ubuntu	latest	e9ae3c220b23	7 weeks
busybox	latest	c51f86c28340	9 weeks
hello-world	latest	0a6ba66e537a	11 weeks

Выше приведен список изображений, которые я извлек из реестра, а также тех, которые я создал сам (вскоре мы увидим, как это сделать). **TAG** относится к определенному снимку изображения, а **IMAGE ID** является соответствующим уникальным идентификатором для этого изображения.

Для простоты вы можете представить себе изображение, похожее на репозиторий git - изображения могут быть **зарегистрированы** с изменениями и иметь несколько версий. Если вы не укажете конкретный номер версии, по умолчанию в клиенте используется значение `latest`. Например, вы можете получить определенную версию `ubuntu` изображения

```
$ docker pull ubuntu:18.04
```

Чтобы получить новый образ Docker, вы можете либо получить его из реестра (например, Docker Hub), либо создать свой собственный. На [Docker Hub](#) доступны десятки тысяч изображений. Вы также можете выполнять поиск изображений непосредственно из командной строки с помощью `docker search`.

Важное различие, о котором следует помнить, когда речь заходит об изображениях, - это разница между базовыми и дочерними изображениями.

- **Базовые образы** - это образы, у которых нет родительского образа, обычно образы с такими ОС, как `ubuntu`, `busybox` или `debian`.
- **Дочерние изображения** - это изображения, которые основаны на базовых изображениях и добавляют дополнительную функциональность.

Затем есть официальные и пользовательские изображения, которые могут быть как базовыми, так и дочерними.

- **Официальные изображения** - это изображения, которые официально поддерживаются сотрудниками Docker. Обычно они состоят из одного слова. В приведенном выше списке изображений `python`, `ubuntu`, `busybox` и `hello-world` изображения являются официальными изображениями.
- **Пользовательские изображения** - это изображения, созданные и распространяемые такими пользователями, как вы и я. Они



основаны на базовых изображениях и добавляют дополнительную функциональность. Обычно они отформатированы как `user/image-name`.

## Наше первое изображение

Теперь, когда мы лучше разбираемся в изображениях, пришло время создать наши собственные. Нашей целью в этом разделе будет создание изображения, которое изолирует простое приложение `Flask`. Для целей этого семинара я уже создал забавное маленькое приложение `Flask`, которое отображает случайного кота `.gif` при каждой загрузке - потому что, знаете, кто не любит кошек? Если вы еще этого не сделали, пожалуйста, продолжайте и клонируйте репозиторий локально следующим образом -

```
$ git clone https://github.com/prakhar1989/docker-curriculum.git  
$ cd docker-curriculum/flask-app
```

*Это должно быть клонировано на компьютере, где вы запускаете команды `Docker`, а не внутри контейнера `Docker`.*

Теперь следующий шаг - создать изображение с помощью этого веб-приложения. Как упоминалось выше, все пользовательские изображения основаны на базовом изображении. Поскольку наше приложение написано на `Python`, базовым образом, который мы собираемся использовать, будет `Python 3`.

## Dockerfile

`Dockerfile` - это простой текстовый файл, содержащий список команд, которые клиент `Docker` вызывает при создании изображения. Это простой способ автоматизировать процесс

создания изображения. Самое приятное то, что [команды](#), которые вы пишете в `Dockerfile`, почти идентичны эквивалентным командам Linux. Это означает, что на самом деле вам не нужно изучать новый синтаксис, чтобы создавать свои собственные файлы `Dockerfile`.

Каталог приложения действительно содержит `Dockerfile`, но поскольку мы делаем это впервые, мы создадим его с нуля. Для начала создайте новый пустой файл в нашем любимом текстовом редакторе и сохраните его в **той же** папке, что и приложение `flask`, под названием `Dockerfile`.

Мы начинаем с указания нашего базового изображения. Для этого используйте `FROM` ключевое слово -

```
FROM python:3.8
```

Следующим шагом обычно является написание команд копирования файлов и установка зависимостей. Сначала мы устанавливаем рабочий каталог, а затем копируем все файлы для нашего приложения.

```
# set a directory for the app  
WORKDIR /usr/src/app  
  
# copy all the files to the container  
COPY . .
```

Теперь, когда у нас есть файлы, мы можем установить зависимости.

```
# install dependencies  
RUN pip install --no-cache-dir -r requirements.txt
```

Следующее, что нам нужно указать, - это номер порта, который необходимо предоставить. Поскольку наше приложение `flask` работает на порту `5000`, именно это мы и укажем.

```
EXPOSE 5000
```

Последний шаг - написать команду для запуска приложения, которая называется просто - `python ./app.py`. Для этого мы используем команду `CMD` -

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

Основная цель `CMD` - указать контейнеру, какую команду он должен выполнить при запуске. После этого наш `Dockerfile` теперь готов. Вот как это выглядит -

```
FROM python:3.8

# set a directory for the app
WORKDIR /usr/src/app

# copy all the files to the container
COPY . .

# install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# define the port number the container should expose
EXPOSE 5000

# run the command
CMD ["python", "./app.py"]
```

Теперь, когда у нас есть наш `Dockerfile`, мы можем создать наш образ. Команда `docker build` выполняет тяжелую работу по созданию образа Docker из `Dockerfile`.

В разделе ниже показан результат выполнения того же самого. Прежде чем запускать команду самостоятельно (не забудьте точку), обязательно замените мое имя пользователя на свое. Это имя пользователя должно быть тем же, которое вы ввели при регистрации на `Docker hub`. Если вы еще этого не сделали, пожалуйста, создайте учетную запись. `docker build` Команда довольно проста - она принимает необязательное имя тега с `-t` и расположение каталога, содержащего `Dockerfile`.

```
$ docker build -t yourusername/catnip .
Sending build context to Docker daemon 8.704 kB
```

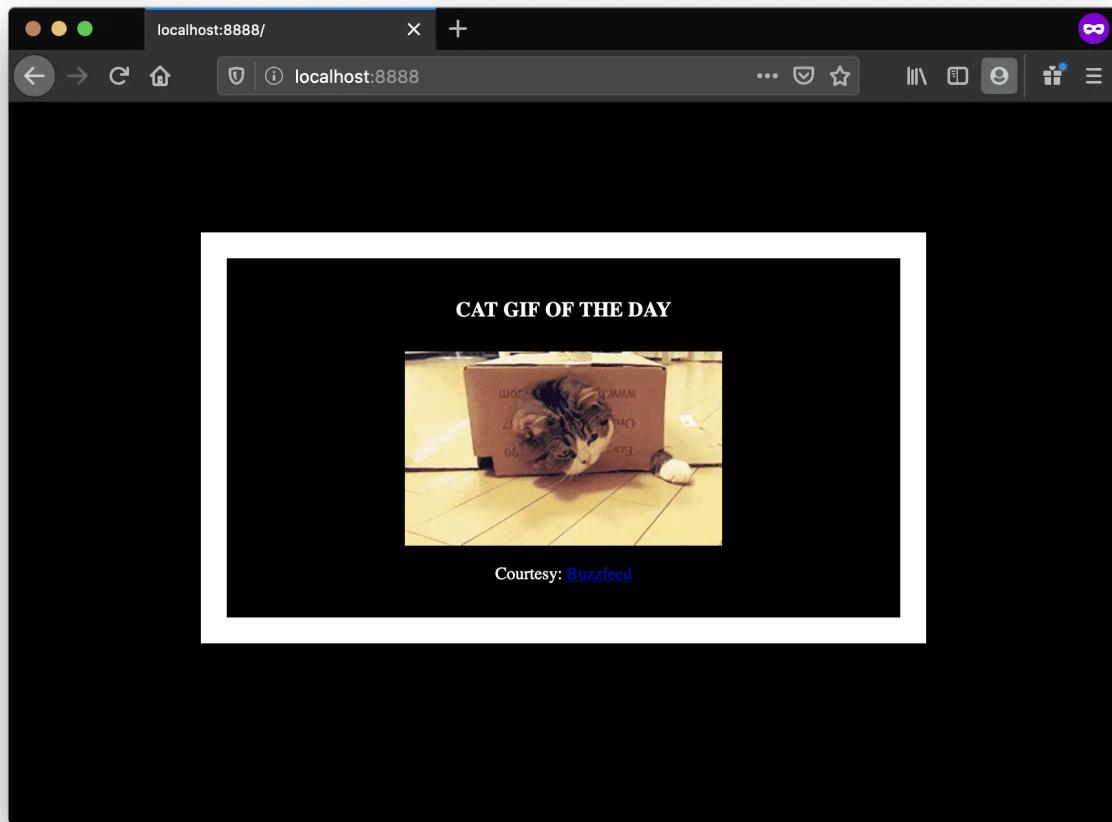
```
Step 1 : FROM python:3.8
# Executing 3 build triggers...
Step 1 : COPY requirements.txt /usr/src/app/
---> Using cache
Step 1 : RUN pip install --no-cache-dir -r requirements.txt
---> Using cache
Step 1 : COPY . /usr/src/app
---> 1d61f639ef9e
Removing intermediate container 4de6ddf5528c
Step 2 : EXPOSE 5000
---> Running in 12cf6d67ee
---> f423c2f179d1
Removing intermediate container 12cf6d67ee
Step 3 : CMD python ./app.py
---> Running in f01401a5ace9
---> 13e87ed1fbc2
Removing intermediate container f01401a5ace9
Successfully built 13e87ed1fbc2
```

If you don't have the `python:3.8` image, the client will first pull the image and then create your image. Hence, your output from running the command will look different from mine. If everything went well, your image should be ready! Run `docker images` and see if your image shows.

The last step in this section is to run the image and see if it actually works (replacing my username with yours).

```
$ docker run -p 8888:5000 yourusername/catnip
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The command we just ran used port 5000 for the server inside the container and exposed this externally on port 8888. Head over to the URL with port 8888, where your app should be live.



Поздравляем! Вы успешно создали свой первый образ Docker.

## Docker на AWS

Что хорошего в приложении, которым нельзя поделиться с друзьями, верно? Итак, в этом разделе мы собираемся посмотреть, как мы можем развернуть наше потрясающее приложение в облаке, чтобы мы могли поделиться им с нашими друзьями! Мы собираемся использовать AWS [Elastic Beanstalk](#), чтобы запустить наше приложение в несколько кликов. Мы также увидим, как легко сделать наше приложение масштабируемым и управляемым с помощью Beanstalk!

## Docker push

Первое, что нам нужно сделать перед развертыванием нашего приложения в AWS, - это опубликовать наше изображение в

реестре, доступ к которому доступен AWS. Вы можете использовать множество различных [реестров Docker](#) (вы даже можете разместить [свой собственный](#)). А пока давайте воспользуемся [Docker Hub](#) для публикации изображения.

Если вы впервые загружаете изображение, клиент попросит вас войти в систему. Укажите те же учетные данные, которые вы использовали для входа в Docker Hub.

```
$ docker login
Login in with your Docker ID to push and pull images from Docker Hub. If you d
Username: yourusername
Password:
WARNING! Your password will be stored unencrypted in /Users/yourusername/.dock
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/credential-store

Login Succeeded
```

Чтобы опубликовать, просто введите приведенную ниже команду, не забыв заменить название тега изображения выше на свое. Важно иметь формат [yourusername/image\\_name](#), чтобы клиент знал, где публиковать.

```
$ docker push yourusername/catnip
```

Как только это будет сделано, вы сможете просмотреть свое изображение в Docker Hub. Например, вот [веб-страница](#) моего изображения.

*Примечание: Прежде чем мы продолжим, я хотел бы прояснить одну вещь: не **обязательно** размещать ваш образ в общедоступном реестре (или любом другом реестре) для развертывания в AWS. Если вы пишете код для следующего стартапа [upicorg](#) стоимостью в миллион долларов, вы можете полностью пропустить этот шаг. Причина, по которой мы публикуем наши изображения, заключается в том, что это*

упрощает развертывание за счет пропуска нескольких промежуточных шагов настройки.

Теперь, когда ваше изображение подключено к Сети, любой, у кого установлен docker, может играть с вашим приложением, введя всего одну команду.

```
$ docker run -p 8888:5000 yourusername/catnip
```

Если вы в прошлом ломали голову над настройкой локальных сред разработки / общим доступом к конфигурации приложений, вы очень хорошо знаете, как потрясающе это звучит. Вот почему Docker такой классный!

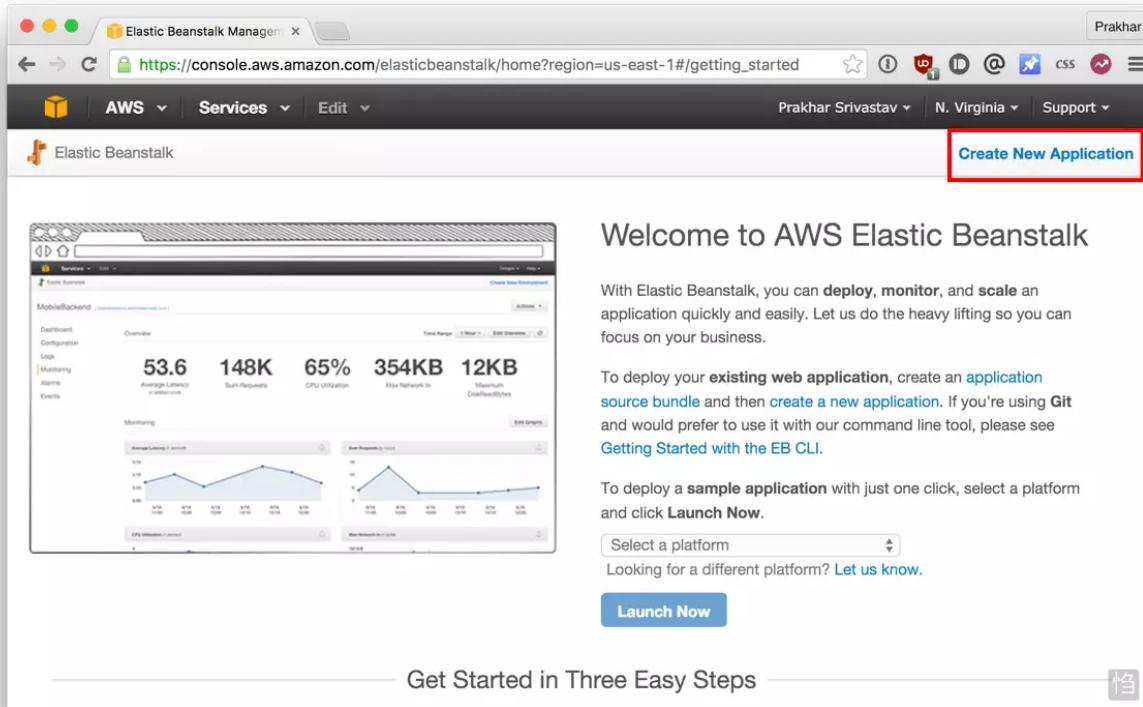
## Beanstalk

AWS Elastic Beanstalk (EB) - это PaaS (Платформа как услуга), предлагаемая AWS. Если вы использовали Heroku, Google App Engine и т.д. вы будете чувствовать себя как дома. Как разработчик, вы просто рассказываете EB, как запустить ваше приложение, а он берет на себя все остальное - включая масштабирование, мониторинг и даже обновления. В апреле 2014 года EB добавила поддержку для запуска одноконтейнерных развертываний Docker, которые мы будем использовать для развертывания нашего приложения. Хотя EB имеет очень интуитивно понятный **CLI**, он требует некоторой настройки, и для простоты мы будем использовать веб-интерфейс для запуска нашего приложения.

Для продолжения вам понадобится действующая учетная запись **AWS**. Если у вас ее еще нет, пожалуйста, сделайте это сейчас - вам нужно будет ввести данные своей кредитной карты. Но не волнуйтесь, это бесплатно, и все, что мы делаем в этом руководстве, также будет бесплатным! Давайте начнем.

Вот шаги:

- Войдите в свою **консоль AWS**.
- Нажмите на Elastic Beanstalk. Оно будет в разделе вычислений вверху слева. В качестве альтернативы вы можете получить доступ к **консоли Elastic Beanstalk**.



- Click on "Create New Application" in the top right
- Give your app a memorable (but unique) name and provide an (optional) description
- In the **New Environment** screen, create a new environment and choose the **Web Server Environment**.
- Fill in the environment information by choosing a domain. This URL is what you'll share with your friends so make sure it's easy to remember.
- Under base configuration section. Choose *Docker* from the *predefined platform*.

**Platform**

Platform

Platform branch

Platform version

**Application code**

Sample application  
Get started right away with sample code.

Upload your code  
Upload a source bundle from your computer or copy one from Amazon S3.

**Source code origin**

(Maximum size 512 MB)

Local file

Public S3 URL

File name : **Dockerrun.aws.json**

File successfully uploaded

Version label  
Unique name for this version of your application code.

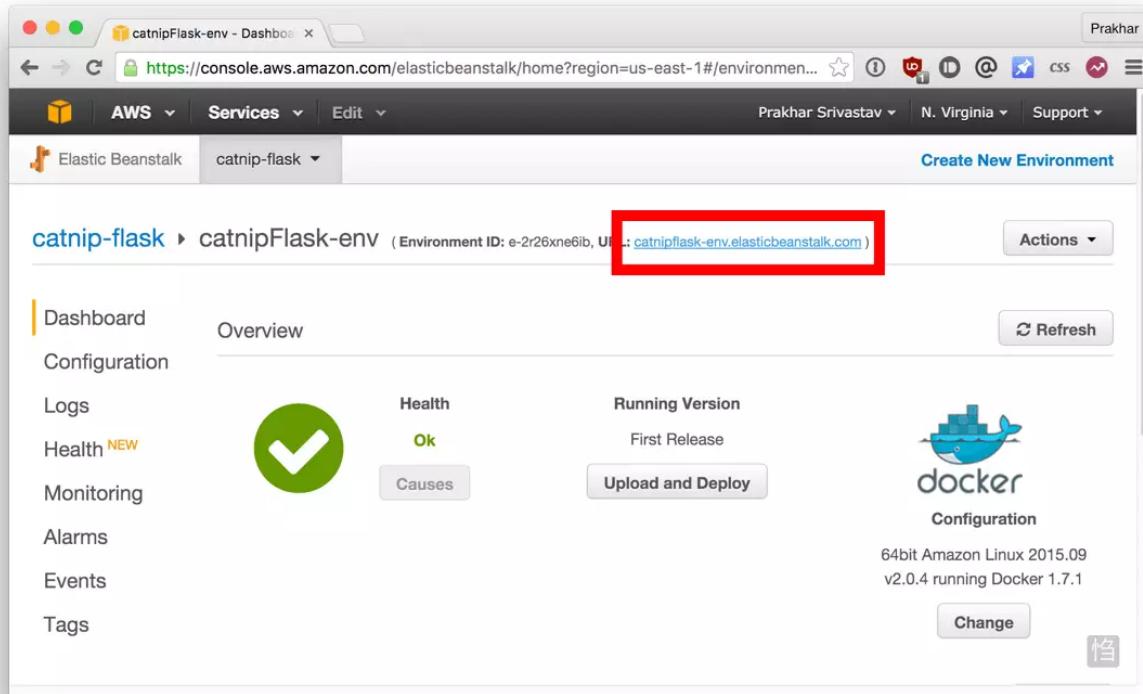
- Теперь нам нужно загрузить код нашего приложения. Но поскольку наше приложение упаковано в контейнер Docker, нам просто нужно сообщить ЕВ о нашем контейнере. Откройте **Dockerrun.aws.json** файл, расположенный в **flask-app** папке, и отредактируйте **Name** изображение, присвоив ему название вашего изображения. Не волнуйтесь, я скоро объясню содержимое файла. Когда вы закончите, нажмите переключатель для "Загрузить свой код", выберите этот файл и нажмите "Загрузить".
- Теперь нажмите "Создать среду". На последнем экране, который вы увидите, будет несколько кнопок, указывающих на то, что ваша среда настраивается. Обычно для первой настройки требуется около 5 минут.

Пока мы ждем, давайте быстро посмотрим, что содержит `Dockerrun.aws.json` файл. По сути, этот файл предназначен для AWS и содержит подробную информацию о нашем приложении и конфигурации docker.

```
{  
    "AWSEBDockerrunVersion": "1",  
    "Image": {  
        "Name": "prakhar1989/catnip",  
        "Update": "true"  
    },  
    "Ports": [  
        {  
            "ContainerPort": 5000,  
            "HostPort": 8000  
        }  
    ],  
    "Logging": "/var/log/nginx"  
}
```

Файл должен быть достаточно понятным, но вы всегда можете [обратиться](#) к официальной документации для получения дополнительной информации. Мы указываем имя образа, который должен использовать EB, а также порт, который должен открывать контейнер.

Надеюсь, к этому времени наш экземпляр должен быть готов. Перейдите на страницу EB, и вы увидите зеленую галочку, указывающую на то, что ваше приложение работает.



Продолжайте и откройте URL-адрес в своем браузере, и вы должны увидеть приложение во всей его красе. Не стесняйтесь отправлять по электронной почте / IM / snapchat эту ссылку своим друзьям и семье, чтобы они тоже могли насладиться несколькими GIF-изображениями кошек.

## Очистка

После того, как вы закончите наслаждаться великолепием своего приложения, не забудьте завершить работу среды, чтобы в конечном итоге с вас не взимали плату за дополнительные ресурсы.

Поздравляем! Вы развернули свое первое приложение Docker! Может показаться, что это много шагов, но с [инструментом командной строки для EB](#) вы можете практически имитировать функциональность Негоки несколькими нажатиями клавиш! Надеюсь, вы согласны с тем, что Docker снимает с себя большую часть хлопот по созданию и развертыванию приложений в облаке. Я бы посоветовал вам прочитать [документацию AWS](#) по одноконтейнерным средам Docker, чтобы получить представление о существующих функциях.

В следующей (и заключительной) части руководства мы немножко увеличим ставку и развернем приложение, более точно имитирующее реальный мир; приложение с постоянным уровнем внутреннего хранилища. Давайте сразу перейдем к делу!

## МНОГОКОНТЕЙНЕРНЫЕ СРЕДЫ

В последнем разделе мы увидели, как легко и увлекательно запускать приложения с помощью Docker. Мы начали с простого статического веб-сайта, а затем попробовали приложение Flask. Оба приложения мы могли запускать локально и в облаке с

помощью всего нескольких команд. Общим для обоих этих приложений было то, что они работали в **одном контейнере**.

Те из вас, у кого есть опыт запуска сервисов в рабочей среде, знают, что в настоящее время приложения не так просты. Почти всегда задействована база данных (или любой другой вид постоянного хранилища). Такие системы, как [Redis](#) и [Memcached](#), стали *обязательными* для большинства архитектур веб-приложений. Следовательно, в этом разделе мы собираемся потратить некоторое время на изучение того, как настраивать приложения, которые зависят от различных служб для запуска.

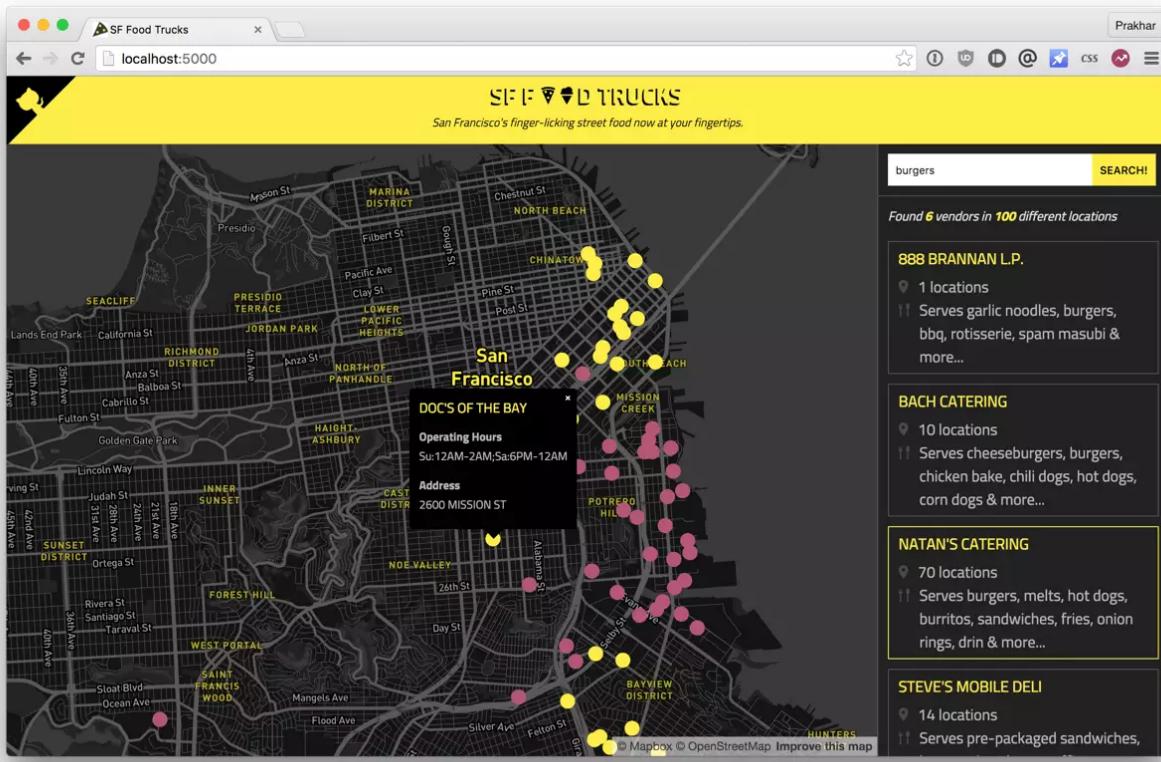
В частности, мы собираемся увидеть, как мы можем запускать **многоконтейнерные** среды docker и управлять ими. Почему многоконтейнерные, спросите вы? Что ж, одним из ключевых моментов Docker является способ, которым он обеспечивает изоляцию. Идея объединения процесса с его зависимостями в изолированной среде (называемой контейнерами) - вот что делает его таким мощным.

Точно так же, как это хорошая стратегия для разделения уровней ваших приложений, разумно хранить контейнеры для каждой из **служб** отдельно. У каждого уровня, вероятно, будут разные потребности в ресурсах, и эти потребности могут расти с разной скоростью. Разделив уровни на разные контейнеры, мы можем составить каждый уровень, используя наиболее подходящий тип экземпляра, исходя из различных потребностей в ресурсах. Это также очень хорошо сочетается со всем движением [микросервисов](#), что является одной из основных причин, по которой Docker (или любая другая контейнерная технология) находится на [переднем крае](#) современных архитектур микросервисов.

## SF Food Trucks

Приложение, которое мы собираемся докеризовать, называется SF Food Trucks. Моей целью при создании этого приложения было

создать что-то полезное (в том смысле, что оно напоминает реальное приложение), опирающееся хотя бы на один сервис, но не слишком сложное для целей этого руководства. Это то, к чему я пришел.



Серверная часть приложения написана на Python (Flask), а для поиска в нем используется [Elasticsearch](#). Как и все остальное в этом руководстве, весь исходный код доступен на [Github](#). Мы будем использовать это как наше приложение-кандидат для изучения того, как создавать, запускать и развертывать многоконтейнерную среду.

Для начала давайте клонируем репозиторий локально.

```
$ git clone https://github.com/prakhar1989/FoodTrucks
$ cd FoodTrucks
$ tree -L 2
.
├── Dockerfile
├── README.md
├── aws-compose.yml
├── docker-compose.yml
├── flask-app
│   └── app.py
```

```

├── package-lock.json
├── package.json
├── requirements.txt
├── static
├── templates
└── webpack.config.js
├── setup-aws-ecs.sh
├── setup-docker.sh
├── shot.png
└── utils
    ├── generate_geojson.py
    └── trucks.geojson

```

В `flask-app` папке находится приложение Python, а в `utils` папке есть некоторые утилиты для загрузки данных в Elasticsearch. Каталог также содержит несколько файлов YAML и Dockerfile, все из которых мы рассмотрим более подробно по мере прохождения этого руководства. Если вам интересно, не стесняйтесь взглянуть на файлы.

Теперь, когда вы взволнованы (надеюсь), давайте подумаем о том, как мы можем выполнить докеризацию приложения. Мы видим, что приложение состоит из внутреннего сервера Flask и службы Elasticsearch. Естественным способом разделить это приложение было бы иметь два контейнера - один, в котором выполняется процесс Flask, а другой, в котором выполняется процесс Elasticsearch (ES). Таким образом, если наше приложение станет популярным, мы сможем масштабировать его, добавляя больше контейнеров в зависимости от того, где находится узкое место.

Отлично, итак, нам нужны два контейнера. Это не должно быть сложно, верно? Мы уже создали наш собственный контейнер Flask в предыдущем разделе. Давайте посмотрим, сможем ли мы найти что-нибудь для Elasticsearch на хабе.

\$ docker search elasticsearch	
NAME	DESCRIPTION
elasticsearch	Elasticsearch is a powerful open source se..
itzg/elasticsearch	Provides an easily configurable Elasticsea..
tutum/elasticsearch	Elasticsearch image - listens in port 9200.
barnybug/elasticsearch	Latest Elasticsearch 1.7.2 and previous re..

Неудивительно, что для Elasticsearch существует официально поддерживаемый [образ](#). Чтобы запустить ES, мы можем просто использовать `docker run` и запустить одноузловой контейнер ES локально в кратчайшие сроки.

*Примечание: Elastic, компания, стоящая за Elasticsearch, ведет свой [собственный реестр](#) продуктов Elastic. Если вы планируете использовать Elasticsearch, рекомендуется использовать изображения из этого реестра.*

Давайте сначала извлекем изображение

```
$ docker pull docker.elastic.co/elasticsearch/elasticsearch:6.3.2
```

а затем запустите его в режиме разработки, указав порты и установив переменную окружения, которая настраивает кластер Elasticsearch для запуска как одноузловой.

```
$ docker run -d --name es -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node"
```



*Примечание: Если у вашего контейнера возникают проблемы с памятью, вам может потребоваться [настроить некоторые флаги JVM](#), чтобы ограничить потребление памяти.*

Как показано выше, мы используем `--name es`, чтобы присвоить нашему контейнеру имя, которое упрощает его использование в последующих командах. После запуска контейнера мы можем просмотреть журналы, выполнив команду `docker container logs` с именем контейнера (или ID) для проверки журналов. Вы должны

увидеть журналы, аналогичные приведенным ниже, если Elasticsearch был успешно запущен.

*Примечание: Запуск Elasticsearch занимает несколько секунд, поэтому вам, возможно, придется подождать, прежде чем вы увидите **initialized** в журналах.*

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND
277451c15ec1      docker.elastic.co/elasticsearch/elasticsearch:6.3.2   "/usr/bin/entrypoint.sh"
$ docker container logs es
[2018-07-29T05:49:09,304][INFO ][o.e.n.Node                ] [L1VMyzt] initializing ...
[2018-07-29T05:49:09,385][INFO ][o.e.e.NodeEnvironment ] [L1VMyzt] using [1]
[2018-07-29T05:49:09,385][INFO ][o.e.e.NodeEnvironment ] [L1VMyzt] heap size limit: 1024m
[2018-07-29T05:49:11,979][INFO ][o.e.p.PluginsService] [L1VMyzt] loaded n/a modules
[2018-07-29T05:49:11,980][INFO ][o.e.p.PluginsService] [L1VMyzt] loaded n/a modules
[2018-07-29T05:49:11,981][INFO ][o.e.p.PluginsService] [L1VMyzt] loaded n/a modules
[2018-07-29T05:49:11,981][INFO ][o.e.p.PluginsService] [L1VMyzt] loaded n/a modules
[2018-07-29T05:49:11,981][INFO ][o.e.p.PluginsService] [L1VMyzt] loaded n/a modules
[2018-07-29T05:49:17,659][INFO ][o.e.d.DiscoveryModule] [L1VMyzt] using discovery.type=single-node
[2018-07-29T05:49:18,962][INFO ][o.e.n.Node                ] [L1VMyzt] initialized
[2018-07-29T05:49:18,963][INFO ][o.e.n.Node                ] [L1VMyzt] starting
[2018-07-29T05:49:19,218][INFO ][o.e.t.TransportService] [L1VMyzt] publish_
[2018-07-29T05:49:19,302][INFO ][o.e.x.s.t.n.SecurityNetty4HttpServerTransport] [L1VMyzt] started
[2018-07-29T05:49:19,303][INFO ][o.e.n.Node                ] [L1VMyzt] started
[2018-07-29T05:49:19,439][WARN ][o.e.x.s.a.s.m.NativeRoleMappingStore] [L1VMyzt] failed to initialize role mapping store
[2018-07-29T05:49:19,542][INFO ][o.e.g.GatewayService] [L1VMyzt] recovered
[2018-07-29T05:49:20,000][INFO ][o.e.n.Node                ] [L1VMyzt] recovered
```

Теперь давайте попробуем проверить, можно ли отправить запрос в контейнер Elasticsearch. Мы используем **9200** порт для отправки **cURL** запроса в контейнер.

```
$ curl 0.0.0.0:9200
{
  "name" : "ijJDA0m",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "a_nSV3XmTCqpzYYzb-LhNw",
  "version" : {
    "number" : "6.3.2",
    "build_flavor" : "default",
    "build_type" : "tar",
```

```

    "build_hash" : "053779d",
    "build_date" : "2018-07-20T05:20:23.451332Z",
    "build_snapshot" : false,
    "lucene_version" : "7.3.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
},
"tagline" : "You Know, for Search"
}

```

Отлично! Выглядит неплохо! Пока мы этим занимаемся, давайте запустим и наш контейнер Flask. Но прежде чем мы перейдем к этому, нам понадобится **Dockerfile**. В последнем разделе мы использовали `python:3.8 image` в качестве базового изображения. Однако на этот раз, помимо установки зависимостей Python через `pip`, мы хотим, чтобы наше приложение также генерировало наш уменьшенный файл Javascript для производства. Для этого нам потребуется Nodejs. Поскольку нам нужен пользовательский этап сборки, мы начнем с `ubuntu` базового образа, чтобы создать наш **Dockerfile** с нуля.

*Примечание: если вы обнаружите, что существующее изображение не соответствует вашим потребностям, смело начинайте с другого базового изображения и настраивайте его самостоятельно. Для большинства изображений на Docker Hub вы должны быть в состоянии найти соответствующие Dockerfile на Github. Чтение существующих файлов Dockerfile - один из лучших способов научиться создавать свои собственные.*

Наш **Dockerfile** для приложения flask выглядит следующим образом

```

# start from base
FROM ubuntu:18.04

MAINTAINER Prakhar Srivastav <prakhar@prakhar.me>

# install system-wide deps for python and node

```

```
RUN apt-get -yqq update
RUN apt-get -yqq install python3-pip python3-dev curl gnupg
RUN curl -sL https://deb.nodesource.com/setup_10.x | bash
RUN apt-get install -yq nodejs

# 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 pip3 install -r requirements.txt

# expose port
EXPOSE 5000

# start app
CMD [ "python3", "./app.py" ]
```

Здесь довольно много нового, поэтому давайте быстро рассмотрим этот файл. Мы начинаем с базового образа **Ubuntu LTS** и используем менеджер пакетов **apt-get** для установки зависимостей, а именно - Python и Node. Флаг **yqq** используется для подавления вывода и предполагает "Да" для всех запросов.

Затем мы используем **ADD** команду, чтобы скопировать наше приложение в новый том в контейнере - **/opt/flask-app**. Именно там будет находиться наш код. Мы также установили это как наш рабочий каталог, чтобы следующие команды выполнялись в контексте этого каталога. Теперь, когда наши общесистемные зависимости установлены, мы приступаем к установке зависимостей для конкретного приложения. Сначала мы займемся Node, установив пакеты из npm и запустив команду сборки, как определено в нашем **package.json** файле. Мы завершаем работу с файлом, устанавливая пакеты Python, открывая порт и определяя **CMD** для запуска, как мы делали в предыдущем разделе.

Наконец, мы можем продолжить, создать образ и запустить контейнер (замените **yourusername** на ваше имя пользователя ниже).

```
$ docker build -t yourusername/foodtrucks-web .
```



При первом запуске это займет некоторое время, поскольку клиент Docker загрузит образ `ubuntu`, выполнит все команды и подготовит ваш образ. Повторный запуск `docker build` после любых последующих изменений, которые вы внесете в код приложения, будет практически мгновенным. Теперь давайте попробуем запустить наше приложение.

```
$ docker run -P --rm yourusername/foodtrucks-web  
Unable to connect to ES. Retrying in 5 secs...  
Unable to connect to ES. Retrying in 5 secs...  
Unable to connect to ES. Retrying in 5 secs...  
Out of retries. Bailing out...
```

Упс! Не удалось запустить наше приложение `flask`, поскольку не удалось подключиться к `Elasticsearch`. Как нам сообщить одному контейнеру о другом контейнере и заставить их общаться друг с другом? Ответ содержится в следующем разделе.

## Docker Network

Прежде чем мы поговорим о функциях, которые Docker предоставляет специально для работы с подобными сценариями, давайте посмотрим, сможем ли мы найти способ обойти проблему. Надеюсь, это должно дать вам представление о конкретной функции, которую мы собираемся изучить.

Хорошо, итак, давайте запустим `docker container ls` (это то же самое, что и `docker ps`) и посмотрим, что у нас есть.

```
$ docker container ls  
CONTAINER ID        IMAGE               COMMAND  
277451c15ec1        docker.elastic.co/elasticsearch/elasticsearch:6.3.2   "/usr/bin/es -Ees._source.lang.type=nested,_score.allow_extr...
```

Итак, у нас есть один контейнер ES, работающий на `0.0.0.0:9200` порту, к которому мы можем получить прямой доступ. Если мы можем указать нашему приложению Flask

ПОДКЛЮЧИТЬСЯ К ЭТОМУ URL, ОНО ДОЛЖНО ИМЕТЬ ВОЗМОЖНОСТЬ подключаться и разговаривать с ES, верно? Давайте углубимся в наш [код Python](#) и посмотрим, как определяются детали подключения.

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

Чтобы это заработало, нам нужно сообщить контейнеру Flask, что контейнер ES запущен на `0.0.0.0` хосте (по умолчанию используется порт `9200`), и это должно заставить его работать, не так ли? К сожалению, это неверно, поскольку IP `0.0.0.0` - это IP для доступа к контейнеру ES с **хост-компьютера**, т. е. с моего Mac. Другой контейнер не сможет получить к нему доступ по тому же IP-адресу. Хорошо, если не по этому IP, то по какому IP-адресу должен быть доступен ES-контейнер? Я рад, что вы задали этот вопрос.

Сейчас самое подходящее время начать наше изучение работы с сетями в Docker. Когда docker установлен, он автоматически создает три сети.

```
$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
c2c695315b3a    bridge    bridge      local
a875bec5d6fd   host      host       local
ead0e804a67b   none     null       local
```

Сеть **bridge** - это сеть, в которой по умолчанию запускаются контейнеры. Таким образом, это означает, что когда я запускал контейнер ES, он работал в этой bridge-сети. Чтобы убедиться в этом, давайте проверим сеть.

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "c2c695315b3aaf8fc30530bb3c6b8f6692cedd5cc7579663f0550dfdd21c9a26",
        "Created": "2018-07-28T20:32:39.405687265Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
```

```

    "IPAM": {
        "Driver": "default",
        "Options": null,
        "Config": [
            {
                "Subnet": "172.17.0.0/16",
                "Gateway": "172.17.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
        "277451c15ec183dd939e80298ea4bcf55050328a39b04124b387d668e3ed3943": {
            "Name": "es",
            "EndpointID": "5c417a2fc6b13d8ec97b76bb54aaaf3ee2d48f328c3f7279ee33517",
            "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"
    },
    "Labels": {}
}

```

1

Вы можете видеть, что наш контейнер `277451c15ec1` указан под **Containers** разделом в выходных данных. Мы также видим IP-адрес, выделенный этому контейнеру - `172.17.0.2`. Это тот IP-адрес, который мы ищем? Давайте выясним, запустив наш контейнер `flask` и попытавшись получить доступ к этому IP-адресу.

```
$ docker run -it --rm yourusername/foodtrucks-web bash
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
```

К настоящему моменту это должно быть довольно просто для вас. Мы запускаем контейнер в интерактивном режиме с помощью `bash` процесса. `--rm` это удобный флаг для запуска одноразовых команд, поскольку контейнер очищается по завершении работы. Мы пробуем `curl` но сначала нам нужно его установить. Как только мы это сделаем, мы увидим, что действительно можем общаться с ES на `172.17.0.2:9200`. Потрясающе!

Хотя мы выяснили, как заставить контейнеры взаимодействовать друг с другом, при таком подходе все еще существуют две проблемы -

1. Как нам сообщить контейнеру Flask, что `es` имя хоста означает `172.17.0.2` или какой-либо другой IP, поскольку IP может измениться?
2. Поскольку сеть *моста* по умолчанию является общей для всех контейнеров, этот метод **небезопасен**. Как нам изолировать нашу сеть?

Хорошая новость в том, что в Docker есть отличный ответ на наши вопросы. Он позволяет нам определять наши собственные сети, сохраняя их изолированными, с помощью команды `docker network`.

Давайте сначала продолжим и создадим нашу собственную сеть.

```
$ docker network create foodtrucks-net
0815b2a3bb7a6608e850d05553cc0bda98187c4528d94621438f31d97a6fea3c
```

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
c2c695315b3a	bridge	bridge	local
0815b2a3bb7a	foodtrucks-net	bridge	local
a875bec5d6fd	host	host	local
ead0e804a67b	none	null	local

**network create** Команда создает новую *мостовую сеть*, которая нам нужна в данный момент. С точки зрения Docker, мостовая сеть использует программный мост, который позволяет контейнерам, подключенным к одной и той же мостовой сети, обмениваться данными, обеспечивая при этом изоляцию от контейнеров, которые не подключены к этой мостовой сети. Драйвер Docker `bridge` автоматически устанавливает правила на хост-компьютере, чтобы контейнеры в разных сетях моста не могли напрямую взаимодействовать друг с другом. Существуют и другие типы сетей, которые вы можете создавать, и вам рекомендуется прочитать о них в официальных [документах](#).

Теперь, когда у нас есть сеть, мы можем запускать наши контейнеры внутри этой сети, используя флаг `--net`. Давайте сделаем это, но сначала, чтобы запустить новый контейнер с тем же именем, мы остановим и удалим наш ES-контейнер, запущенный в сети `bridge` (по умолчанию).

```
$ docker container stop es
es
```

```
$ docker container rm es
es
```

```
$ docker run -d --name es --net foodtrucks-net -p 9200:9200 -p 9300:9300 -e "d
13d6415f73c8d88bddb1f236f584b63dbaf2c3051f09863a3f1ba219edba3673
```

```
$ docker network inspect foodtrucks-net
[
  {
    "Name": "foodtrucks-net",
    "Id": "0815b2a3bb7a6608e850d05553cc0bda98187c4528d94621438f31d97a6fea3c",
    "Created": "2018-07-30T00:01:29.1500984Z",
```

```

    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": {},
        "Config": [
            {
                "Subnet": "172.18.0.0/16",
                "Gateway": "172.18.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
        "13d6415f73c8d88bddb1f236f584b63dbaf2c3051f09863a3f1ba219edba3673": {
            "Name": "es",
            "EndpointID": "29ba2d33f9713e57eb6b38db41d656e4ee2c53e4a2f7cf636bdca0e",
            "MacAddress": "02:42:ac:12:00:02",
            "IPv4Address": "172.18.0.2/16",
            "IPv6Address": ""
        }
    },
    "Options": {},
    "Labels": {}
}
1

```

Как вы можете видеть, наш `es` контейнер теперь запущен внутри `foodtrucks-net` сети `bridge`. Теперь давайте проверим, что происходит при запуске в нашей `foodtrucks-net` сети.

```
$ docker run -it --rm --net foodtrucks-net yourusername/foodtrucks-web bash
root@9d2722cf282c:/opt/flask-app# curl es:9200
{
  "name" : "wWALL9M",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "BA36Xu0iRPaghPNBLBHleQ",
  "version" : {
    "number" : "6.3.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_timestamp" : "2017-07-13T11:40:40Z",
    "build_snapshot" : true,
    "git_version" : "v6.3.2-11-ga0a3010"
  }
}
```

```

    "build_hash" : "053779d",
    "build_date" : "2018-07-20T05:20:23.451332Z",
    "build_snapshot" : false,
    "lucene_version" : "7.3.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
},
"tagline" : "You Know, for Search"
}
root@53af252b771a:/opt/flask-app# ls
app.py  node_modules  package.json  requirements.txt  static  templates  webpac
root@53af252b771a:/opt/flask-app# python3 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

```

Вау! Это работает! В пользовательских сетях, таких как foodtrucks-net, контейнеры могут не только обмениваться данными по IP-адресу, но и преобразовывать имя контейнера в IP-адрес. Эта возможность называется *автоматическое обнаружение служб*. Отлично! Давайте запустим наш контейнер Flask по-настоящему прямо сейчас -

```

$ docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web yourus
852fc74de2954bb72471b858dce64d764181dca0cf7693fed201d76da33df794

$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
852fc74de295        yourusername/foodtrucks-web   "pytl
13d6415f73c8        docker.elastic.co/elasticsearch/elasticsearch:6.3.2   "/us

$ 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

```

Перейдите на страницу <http://0.0.0.0:5000> и посмотрите на свое великолепное приложение вживую! Хотя это может показаться сложной работой, на самом деле мы просто ввели 4 команды,

Чтобы перейти от нуля к запуску. Я собрал команды в **bash-скрипте**.

```
#!/bin/bash

# build the flask container
docker build -t yourusername/foodtrucks-web .

# create the network
docker network create foodtrucks-net

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

# start the flask app container
docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web youruser
```

Теперь представьте, что вы распространяете свое приложение среди друзей или запускаете на сервере, на котором установлен docker. Вы можете запустить целое приложение всего одной командой!

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

И это все! Если вы спросите меня, я нахожу это чрезвычайно потрясающим и мощным способом совместного использования и запуска ваших приложений!

## Docker Compose

До сих пор мы тратили все свое время на изучение клиента Docker. Однако в экосистеме Docker есть множество других инструментов с открытым исходным кодом, которые очень хорошо сочетаются с Docker. Некоторые из них -

1. **Docker Machine** - Создайте узлы Docker на своем компьютере, у облачных провайдеров и в вашем собственном центре обработки данных

- ☰
  - 2. **Docker Compose** - инструмент для определения и запуска многоконтейнерных приложений Docker.
  - 3. **Docker Swarm** - собственное решение для кластеризации для Docker
  - 4. **Kubernetes** - Kubernetes - это система с открытым исходным кодом для автоматизации развертывания, масштабирования и управления контейнерными приложениями.

В этом разделе мы рассмотрим один из этих инструментов, Docker Compose, и посмотрим, как он может упростить работу с многоконтейнерными приложениями.

История создания Docker Compose довольно интересна. Примерно в январе 2014 года компания OrchardUp запустила инструмент под названием Fig. Идея Fig заключалась в том, чтобы заставить изолированные среды разработки работать с Docker. Проект был очень хорошо принят в [Hacker News](#) - я, как ни странно, помню, что читал о нем, но не совсем разобрался в нем.

[Первый комментарий](#) на форуме на самом деле хорошо объясняет, что такое Fig.

*Итак, на данный момент суть Docker именно в этом: запуске процессов. Теперь Docker предлагает довольно богатый API для запуска процессов: общие тома (каталоги) между контейнерами (т. Е. Запущенные образы), перенаправление порта с хоста на контейнер, отображение журналов и так далее. Но это все: Docker на данный момент остается на уровне процесса.*

*Хотя оно предоставляет опции для управления несколькими контейнерами для создания единого "приложения", в нем не рассматривается управление такой группой контейнеров как единым объектом. И вот тут на помощь приходят такие инструменты, как Fig: речь идет о группе контейнеров как о*

едином объекте. Подумайте "запустить приложение" (т. е. "запустить организованный кластер контейнеров") вместо "запустить контейнер".

Оказывается, что многие люди, использующие Docker, согласны с этим мнением. Медленно и неуклонно, по мере того как Fig становился популярным, Docker Inc. обратила на это внимание, [приобрела компанию](#) и переименовала Fig в Docker Compose.

Итак, для чего *используется Compose?* Compose - это инструмент, который используется для простого определения и запуска многоконтейнерных приложений Docker. Оно предоставляет файл конфигурации под названием `docker-compose.yml`, который можно использовать для запуска приложения и набора служб, от которых оно зависит, всего одной командой. Compose работает во всех средах: `production`, `staging`, разработке, тестировании, а также в рабочих процессах CI, хотя Compose идеально подходит для сред разработки и тестирования.

Давайте посмотрим, сможем ли мы создать `docker-compose.yml` файл для нашего приложения SF-Foodtrucks и оценим, соответствует ли Docker Compose своим обещаниям.

Однако первым шагом является установка Docker Compose. Если вы используете Windows или Mac, Docker Compose уже установлен в том виде, в каком он поставляется в Docker Toolbox.

Пользователи Linux могут легко освоить Docker Compose, следуя [инструкциям](#) в документах. Поскольку Compose написан на Python, вы также можете просто выполнить `pip install docker-compose`. Протестируйте свою установку с помощью -

```
$ docker-compose --version
docker-compose version 1.21.2, build a133471
```

Теперь, когда мы его установили, мы можем перейти к следующему шагу, то есть к файлу Docker Compose `docker-compose.yml`.

Синтаксис YAML довольно прост, и репозиторий уже содержит файл docker-compose, который мы будем использовать.

```
version: "3"
services:
  es:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
    container_name: es
    environment:
      - discovery.type=single-node
    ports:
      - 9200:9200
    volumes:
      - esdata1:/usr/share/elasticsearch/data
  web:
    image: yourusername/foodtrucks-web
    command: python3 app.py
    depends_on:
      - es
    ports:
      - 5000:5000
    volumes:
      - ./flask-app:/opt/flask-app
volumes:
  esdata1:
    driver: local
```

Позвольте мне объяснить, что означает приведенный выше файл. На родительском уровне мы определяем названия наших сервисов - `es` и `web`. Параметр `image` всегда является обязательным, и для каждой службы, которую мы хотим запустить в Docker, мы можем добавить дополнительные параметры. Для `es` мы просто ссылаемся на `elasticsearch` изображение, доступное в `Elastic registry`. Для нашего приложения `Flask` мы ссылаемся на изображение, созданное в начале этого раздела.

Другие параметры, такие как `command` и `ports`, предоставляют дополнительную информацию о контейнере. `volumes` Параметр указывает точку монтирования в нашем `web` контейнере, где будет находиться код. Это чисто необязательно и полезно, если вам нужен доступ к журналам и т.д. Позже мы увидим, как это может быть полезно во время разработки. Обратитесь к [онлайн-справочнику](#), чтобы узнать больше о параметрах, поддерживаемых

этим файлом. Мы также добавляем тома для `es` контейнера, чтобы загружаемые нами данные сохранялись между перезапусками. Мы также указываем, `depends_on` который сообщает docker о необходимости запуска `es` контейнера перед `web`. Подробнее об этом вы можете прочитать в [документах docker compose](#).

*Примечание: Вы должны находиться внутри каталога с `docker-compose.yml` файлом, чтобы выполнить большинство команд Compose.*

Отлично! Теперь файл готов, давайте посмотрим `docker-compose` в действии. Но прежде чем мы начнем, нам нужно убедиться, что порты и имена свободны. Итак, если у вас запущены контейнеры Flask и ES, давайте выключим их.

```
$ docker stop es foodtrucks-web  
es  
foodtrucks-web  
  
$ docker rm es foodtrucks-web  
es  
foodtrucks-web
```

Теперь мы можем запустить `docker-compose`. Перейдите в каталог `food trucks` и запустите `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] ve  
es_1  | [2016-01-11 03:43:50,307][INFO ][node] [Comet] ir  
es_1  | [2016-01-11 03:43:50,366][INFO ][plugins] [Comet] lc  
es_1  | [2016-01-11 03:43:50,421][INFO ][env] [Comet] us  
es_1  | [2016-01-11 03:43:52,626][INFO ][node] [Comet] ir  
es_1  | [2016-01-11 03:43:52,632][INFO ][node] [Comet] si  
es_1  | [2016-01-11 03:43:52,703][WARN ][common.network] [Comet] pu  
es_1  | [2016-01-11 03:43:52,704][INFO ][transport] [Comet] pu  
es_1  | [2016-01-11 03:43:52,721][INFO ][discovery] [Comet] e]  
es_1  | [2016-01-11 03:43:55,785][INFO ][cluster.service] [Comet] ne
```

```

es_1 | [2016-01-11 03:43:55,818][WARN ][common.network] [Comet] pr
es_1 | [2016-01-11 03:43:55,819][INFO ][http] [Comet] pr
es_1 | [2016-01-11 03:43:55,819][INFO ][node] [Comet] s1
es_1 | [2016-01-11 03:43:55,826][INFO ][gateway] [Comet] re
es_1 | [2016-01-11 03:44:01,825][INFO ][cluster.metadata] [Comet] [s
es_1 | [2016-01-11 03:44:02,373][INFO ][cluster.metadata] [Comet] [s
es_1 | [2016-01-11 03:44:02,510][INFO ][cluster.metadata] [Comet] [s
es_1 | [2016-01-11 03:44:02,593][INFO ][cluster.metadata] [Comet] [s
es_1 | [2016-01-11 03:44:02,708][INFO ][cluster.metadata] [Comet] [s
es_1 | [2016-01-11 03:44:03,047][INFO ][cluster.metadata] [Comet] [s
web_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

```

Перейдите на страницу IP, чтобы увидеть свое приложение вживую. Это было потрясающе, не правда ли? Всего несколько строк настройки, и у нас есть два контейнера Docker, успешно работающих в унисон. Давайте остановим службы и повторно запустим в отключенном режиме.

```

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
Creating es ... done
Creating foodtrucks_web_1 ... done

```

```

$ docker-compose ps
      Name           Command       State    Ports
----- 
es        /usr/local/bin/docker-entr ...   Up      0.0.0.0:9200->9200
foodtrucks_web_1  python3 app.py       Up      0.0.0.0:5000->5000

```

Неудивительно, что мы видим, что оба контейнера успешно запущены. Откуда взялись названия? Они были созданы автоматически с помощью Compose. Но Compose также автоматически создает сеть? Хороший вопрос! Давайте выясним.

Для начала остановим запуск служб. Мы всегда можем запустить их обратно всего одной командой. Объемы данных сохранятся, поэтому можно снова запустить кластер с теми же данными с помощью `docker-compose up`. Чтобы уничтожить кластер и тома данных, просто введите `docker-compose down -v`.

```
$ docker-compose down -v
Stopping foodtrucks_web_1 ... done
Stopping es ... done
Removing foodtrucks_web_1 ... done
Removing es ... done
Removing network foodtrucks_default
Removing volume foodtrucks_esdata1
```

Пока мы этим занимаемся, мы также удалим `foodtrucks` сеть, которую мы создали в прошлый раз.

```
$ docker network rm foodtrucks-net
$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
c2c695315b3a    bridge    bridge      local
a875bec5d6fd    host      host       local
ead0e804a67b    none      null       local
```

Отлично! Теперь, когда у нас все с чистого листа, давайте перезапустим наши сервисы и посмотрим, творит ли *Compose* свое волшебство.

```
$ docker-compose up -d
Recreating foodtrucks_es_1
Recreating foodtrucks_web_1

$ docker container ls
CONTAINER ID      IMAGE      COMMAND      CRE/
f50bb33a3242      yourusername/foodtrucks-web   "python3 app.py"      14 sec
e299ceeb4caa      elasticsearch      "/docker-entrypoint.s"  14 sec
◀          ▶
```

Пока все хорошо. Пора посмотреть, были ли созданы какие-либо сети.

```
$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
c2c695315b3a    bridge    bridge      local
f3b80f381ed3    foodtrucks_default  bridge      local
a875bec5d6fd    host      host       local
ead0e804a67b    none      null       local
```

Вы можете видеть, что *compose* пошла дальше и создала новую сеть под названием `foodtrucks_default` и подключила обе новые

службы в этой сети, чтобы каждую из них можно было обнаружить в другой. Каждый контейнер для службы подключается к сети по умолчанию и доступен как для других контейнеров в этой сети, так и для обнаружения ими по имени хоста, идентичному имени контейнера.

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
8c6bb7e818ec	docker.elastic.co/elasticsearch/elasticsearch:6.3.2	"/usr
7640cec7feb7	yourusername/foodtrucks-web	"pyt

```
$ docker network inspect foodtrucks_default
```

```
[
```

```
{
```

```
    "Name": "foodtrucks_default",
    "Id": "f3b80f381ed3e03b3d5e605e42c4a576e32d38ba24399e963d7dad848b3b4fe7",
    "Created": "2018-07-30T03:36:06.0384826Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": null,
        "Config": [
            {
                "Subnet": "172.19.0.0/16",
                "Gateway": "172.19.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": true,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
        "7640cec7feb7f5615eaac376271a93fb8bab2ce54c7257256bf16716e05c65a5": {
            "Name": "foodtrucks_web_1",
            "EndpointID": "b1aa3e735402abafea3edfbba605eb4617f81d94f1b5f8fcc566a87",
            "MacAddress": "02:42:ac:13:00:02",
            "IPv4Address": "172.19.0.2/16",
            "IPv6Address": ""
        },
        "8c6bb7e818ec1f88c37f375c18f00beb030b31f4b10aee5a0952aad753314b57": {
            "Name": "es",
            "EndpointID": "649b3567d38e5e6f03fa6c004a4302508c14a5f2ac086ee6dcf13dd"
        }
    }
}
```

```

        "MacAddress": "02:42:ac:13:00:03",
        "IPv4Address": "172.19.0.3/16",
        "IPv6Address": ""

    },
    "Options": {},
    "Labels": {
        "com.docker.compose.network": "default",
        "com.docker.compose.project": "foodtrucks",
        "com.docker.compose.version": "1.21.2"
    }
}
1

```

## Рабочий процесс разработки

Прежде чем мы перейдем к следующему разделу, я хотел бы рассказать еще кое-что о docker-compose. Как говорилось ранее, docker-compose действительно отлично подходит для разработки и тестирования. Итак, давайте посмотрим, как мы можем настроить compose, чтобы упростить нашу жизнь во время разработки.

На протяжении всего этого руководства мы работали с готовыми образами docker. Хотя мы создавали образы с нуля, мы еще не касались кода приложения и в основном ограничивались редактированием Dockerfiles и конфигураций YAML. Вам, должно быть, интересно, как выглядит рабочий процесс во время разработки? Предполагается ли продолжать создавать образы Docker для каждого изменения, затем публиковать их, а затем запускать, чтобы увидеть, работают ли изменения так, как ожидалось? Я уверен, что это звучит очень утомительно. Должен быть способ получше. В этом разделе мы собираемся изучить именно это.

Давайте посмотрим, как мы можем внести изменения в приложение Foodtrucks, которое мы только что запустили. Убедитесь, что приложение запущено.,

```
$ docker container ls
CONTAINER ID      IMAGE
```

COM

5450ebedd03c 05d408b25dfe	yourusername/foodtrucks-web docker elastic.co/elasticsearch/elasticsearch:6.3.2	"pyt "/lls
------------------------------	--	---------------

Теперь давайте посмотрим, можем ли мы изменить это приложение, чтобы оно отображало `Hello world!` сообщение при отправке запроса в `/hello` route . В настоящее время приложение отвечает 404.

```
$ curl -I 0.0.0.0:5000/hello
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 233
Server: Werkzeug/0.11.2 Python/2.7.15rc1
Date: Mon, 30 Jul 2018 15:34:38 GMT
```

Почему это происходит? Поскольку наше приложение является приложением Flask, мы можем посмотреть `app.py` ([ссылка](#)) для получения ответов. Во Flask маршруты определяются с помощью синтаксиса `@app.route` . В файле вы увидите, что у нас определены только три маршрута - `/`, `/debug` и `/search` . `/` Маршрут отображает основное приложение, `debug` маршрут используется для возврата некоторой отладочной информации и, наконец, `search` используется приложением для запроса `elasticsearch`.

```
$ curl 0.0.0.0:5000/debug
{
  "msg": "yellow open sfdata Ibkx7WYjSt-g8NZXOEtTMg 5 1 618 0 1.3mb 1.3mb\\n",
  "status": "success"
}
```

Учитывая этот контекст, как бы мы добавили новый маршрут для `hello`? Вы уже догадались! Давайте откроем `flask-app/app.py` в нашем любимом редакторе и внесем следующее изменение

```
@app.route('/')
def index():
    return render_template("index.html")

# add a new hello route
@app.route('/hello')
def hello():
    return "hello world!"
```

Теперь давайте попробуем сделать запрос еще раз

```
$ curl -I 0.0.0.0:5000/hello
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 233
Server: Werkzeug/0.11.2 Python/2.7.15rc1
Date: Mon, 30 Jul 2018 15:34:38 GMT
```

О нет! Это не сработало! Что мы сделали не так? Хотя мы и внесли изменения в `app.py`, файл находится на нашем компьютере (или хост-компьютере), но поскольку Docker запускает наши контейнеры на основе `yourusername/foodtrucks-web` образа, он не знает об этом изменении. Чтобы убедиться в этом, давайте попробуем следующее -

```
$ docker-compose run web bash
Starting es ... done
root@581e351c82b0:/opt/flask-app# ls
app.py      package-lock.json  requirements.txt  templates
node_modules  package.json      static           webpack.config.js
root@581e351c82b0:/opt/flask-app# grep hello app.py
root@581e351c82b0:/opt/flask-app# exit
```

Что мы пытаемся здесь сделать, так это проверить, что наши изменения не относятся к `app.py` который выполняется в контейнере. Мы делаем это, выполняя команду `docker-compose run`, которая похожа на свою двоюродную сестру `docker run`, но принимает дополнительные аргументы для службы (которые есть `web` в нашем `случае`). Как только мы запускаем `bash`, оболочка открывается в `/opt/flask-app`, как указано в нашем `Dockerfile`. Из команды `grep` мы можем видеть, что наших изменений нет в файле.

Давайте посмотрим, как мы можем это исправить. Прежде всего, нам нужно сказать Docker compose, чтобы он не использовал изображение, а вместо этого использовал файлы локально. Мы также установим режим отладки на `true`, чтобы Flask знал о необходимости перезагрузки сервера при `app.py` изменениях.

☰ Замените web часть docker-compose.yml файла следующим образом:

```
version: "3"
services:
  es:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
    container_name: es
    environment:
      - discovery.type=single-node
    ports:
      - 9200:9200
    volumes:
      - esdata1:/usr/share/elasticsearch/data
  web:
    build: . # replaced image with build
    command: python3 app.py
    environment:
      - DEBUG=True # set an env var for flask
    depends_on:
      - es
    ports:
      - "5000:5000"
    volumes:
      - ./flask-app:/opt/flask-app
volumes:
  esdata1:
    driver: local
```

С этим изменением ([diff](#)) давайте остановим и запустим контейнеры.

```
$ docker-compose down -v
Stopping foodtrucks_web_1 ... done
Stopping es ... done
Removing foodtrucks_web_1 ... done
Removing es ... done
Removing network foodtrucks_default
Removing volume foodtrucks_esdata1

$ docker-compose up -d
Creating network "foodtrucks_default" with the default driver
Creating volume "foodtrucks_esdata1" with local driver
Creating es ... done
Creating foodtrucks_web_1 ... done
```

В качестве последнего шага давайте внесем изменения в [app.py](#), добавив новый маршрут. Теперь мы попробуем свернуть

```
$ curl 0.0.0.0:5000/hello  
hello world
```

Вау! Мы получаем действительный ответ! Попробуйте поиграть, внеся больше изменений в приложение.

На этом завершается наше знакомство с Docker Compose. С помощью Docker Compose вы также можете приостановить работу своих служб, запустить одноразовую команду для контейнера и даже увеличить количество контейнеров. Я также рекомендую вам ознакомиться с несколькими другими [вариантами использования Docker compose](#). Надеюсь, я смог показать вам, насколько легко управлять многоконтейнерными средами с помощью Compose. В заключительном разделе мы собираемся развернуть наше приложение на AWS!

## AWS Elastic Container Service

В последнем разделе мы использовали [docker-compose](#) для локального запуска нашего приложения одну команду: [docker-compose up](#). Теперь, когда у нас есть работающее приложение, мы хотим поделиться им со всем миром, привлечь несколько пользователей, заработать кучу денег и купить большой дом в Майами. Выполнение последних трех пунктов выходит за рамки данного руководства, поэтому вместо этого мы потратим наше время на выяснение того, как мы можем развернуть наши многоконтейнерные приложения в облаке с помощью AWS.

Если вы дочитали до этого места, вы в значительной степени убеждены, что Docker - довольно крутая технология. И вы не одиноки. Видя стремительный рост Docker, почти все поставщики облачных технологий начали работать над добавлением поддержки для развертывания приложений Docker на своей платформе. На сегодняшний день вы можете развертывать контейнеры на [Google](#)

**Cloud Platform**, **AWS**, **Azure** и многих других. У нас уже есть руководство по развертыванию одноконтейнерных приложений с помощью **Elastic Beanstalk**, и в этом разделе мы рассмотрим **Elastic Container Service** (или **ECS**) от AWS.

AWS ECS - это масштабируемый и сверхгибкий сервис управления контейнерами, поддерживающий контейнеры Docker. Он позволяет управлять кластером Docker поверх инстансов EC2 с помощью простого в использовании API. Там, где Beanstalk поставляется с разумными настройками по умолчанию, ECS позволяет вам полностью настроить вашу среду в соответствии с вашими потребностями. Это делает ECS, на мой взгляд, довольно сложным для начала работы.

К счастью для нас, в ECS есть удобный инструмент **CLI**, который понимает файлы Docker Compose и автоматически настраивает кластер в ECS! Поскольку у нас уже есть действующий `docker-compose.yml` запуск AWS не должен потребовать больших усилий. Итак, давайте начнем!

Первым шагом является установка CLI. Инструкции по установке CLI как на Mac, так и на Linux очень четко описаны в [официальных документах](#). Продолжайте, установите CLI, а когда закончите, подтвердите установку, запустив

```
$ ecs-cli --version  
ecs-cli version 1.18.1 (7e9df84)
```

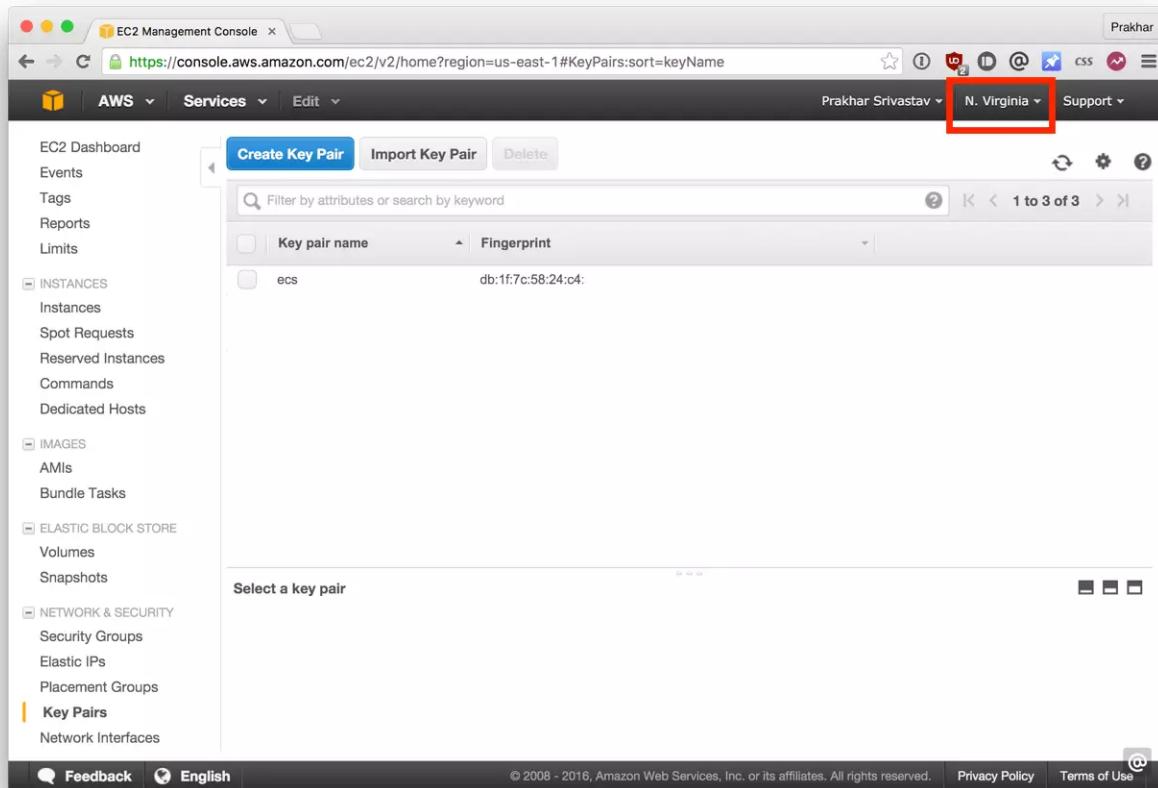
Далее мы будем работать над настройкой командной строки, чтобы иметь возможность взаимодействовать с ECS. Мы будем следовать инструкциям, подробно описанным в [официальном руководстве](#) по AWS ECS docs. В случае каких-либо недоразумений, пожалуйста, не стесняйтесь обращаться к этому руководству.

Первым шагом будет создание профиля, который мы будем использовать в остальной части руководства. Чтобы продолжить, вам понадобятся ваши `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY`. Чтобы получить их, выполните действия,

подробно описанные в разделе, озаглавленном *Ключ доступа и секретный ключ доступа на этой странице*.

```
$ ecs-cli configure profile --profile-name ecs-foodtrucks --access-key $AWS_ACCESS_KEY_ID
```

Далее нам нужно получить пару ключей, которые мы будем использовать для входа в инстансы. Перейдите в свою **консоль EC2** и создайте новую пару ключей. Загрузите пару ключей и сохраните ее в надежном месте. Еще одна вещь, на которую следует обратить внимание, прежде чем вы уйдете с этого экрана, - это название региона. В моем случае я назвал свой ключ - **ecs** и задал свой регион как **us-east-1**. Это то, что я буду предполагать для остальной части этого пошагового руководства.



Следующим шагом будет настройка CLI.

```
$ ecs-cli configure --region us-east-1 --cluster foodtrucks
INFO[0000] Saved ECS CLI configuration for cluster (foodtrucks)
```

Мы предоставляем команде `configure` название региона, в котором мы хотим разместить наш кластер, и имя кластера. Убедитесь, что вы указали **то же название региона**, которое вы использовали при создании пары ключей. Если вы еще не настраивали **AWS CLI** на своем компьютере, вы можете воспользоваться официальным **руководством**, в котором очень подробно объясняется, как все запустить.

Следующий шаг позволяет командной строке создать шаблон **CloudFormation**.

```
$ ecs-cli up --keypair ecs --capability-iam --size 1 --instance-type t2.medium
INFO[0000] Using recommended Amazon Linux 2 AMI with ECS Agent 1.39.0 and Docker
INFO[0000] Created cluster                                     cluster=foodtrucks
INFO[0001] Waiting for your cluster resources to be created
INFO[0001] Cloudformation stack status                         stackStatus=CREATE_IN_PROGRESS
INFO[0062] Cloudformation stack status                         stackStatus=CREATE_IN_PROGRESS
INFO[0122] Cloudformation stack status                         stackStatus=CREATE_IN_PROGRESS
INFO[0182] Cloudformation stack status                         stackStatus=CREATE_IN_PROGRESS
INFO[0242] Cloudformation stack status                         stackStatus=CREATE_IN_PROGRESS
VPC created: vpc-0bbbed8536930053a6
Security Group created: sg-0cf767fb4d01a3f99
Subnet created: subnet-05de1db2cb1a50ab8
Subnet created: subnet-01e1e8bc95d49d0fd
Cluster creation succeeded.
```

Здесь мы указываем имя пары ключей, которую мы загрузили изначально (`ecs` в моем случае), количество экземпляров, которые мы хотим использовать (`--size`), и тип экземпляров, на которых мы хотим запускать контейнеры. Флаг `--capability-iam` сообщает командной строке, что мы признаем, что эта команда может создавать ресурсы IAM.

На последнем шаге мы будем использовать наш `docker-compose.yml` файл. Нам нужно внести несколько незначительных изменений, поэтому вместо модификации оригинала давайте сделаем его копию. Содержимое **этого файла** (после внесения изменений) выглядит следующим образом (ниже) -

```
version: '2'
services:
```

```

es:
  image: docker.elastic.co/elasticsearch/elasticsearch:7.6.2
  cpu_shares: 100
  mem_limit: 3621440000
  environment:
    - discovery.type=single-node
    - bootstrap.memory_lock=true
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  logging:
    driver: awslogs
    options:
      awslogs-group: foodtrucks
      awslogs-region: us-east-1
      awslogs-stream-prefix: es
  web:
    image: yourusername/foodtrucks-web
    cpu_shares: 100
    mem_limit: 262144000
    ports:
      - "80:5000"
    links:
      - es
    logging:
      driver: awslogs
      options:
        awslogs-group: foodtrucks
        awslogs-region: us-east-1
        awslogs-stream-prefix: web

```

Единственные изменения, которые мы внесли по сравнению с оригиналом, `docker-compose.yml` касаются предоставления `mem_limit` (в байтах) и `cpu_shares` значений для каждого контейнера и добавления некоторой конфигурации ведения журнала. Это позволяет нам просматривать журналы, сгенерированные нашими контейнерами в [AWS CloudWatch](#). Перейдите в CloudWatch, чтобы [создать группу журналов](#) под названием `foodtrucks`. Обратите внимание, что, поскольку ElasticSearch обычно занимает больше памяти, мы ограничили объем памяти примерно 3,4 ГБ. Еще одна вещь, которую нам нужно сделать, прежде чем мы перейдем к следующему шагу, - это опубликовать наше изображение в Docker Hub.

```
$ docker push yourusername/foodtrucks-web
```

Отлично! Теперь давайте запустим последнюю команду, которая развернет наше приложение в ECS!

```
$ cd aws-ecs
$ ecs-cli compose up
INFO[0000] Using ECS task definition
INFO[0000] Starting container...
INFO[0000] Starting container...
INFO[0000] Describe ECS container status
INFO[0000] Describe ECS container status
INFO[0036] Describe ECS container status
INFO[0048] Describe ECS container status
INFO[0048] Describe ECS container status
INFO[0060] Started container...
INFO[0060] Started container...
TaskDefinition=ecscolors
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/web
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/es
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/web
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/es
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/web
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/es
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/web
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/es
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/web
container=845e2368-170d-44a7-bf9f-84c7fcd9ae29/es
```

Не случайно, что приведенный выше вызов похож на тот, который мы использовали с **Docker Compose**. Если все прошло хорошо, вы должны увидеть `desiredStatus=RUNNING lastStatus=RUNNING` в качестве последней строки.

Потрясающе! Наше приложение работает в режиме реального времени, но как мы можем получить к нему доступ?

```
ecs-cli ps
Name                           State    Ports
845e2368-170d-44a7-bf9f-84c7fcd9ae29/web  RUNNING  54.86.14.14:80->5000/tcp
845e2368-170d-44a7-bf9f-84c7fcd9ae29/es    RUNNING
```

Продолжайте и откройте <http://54.86.14.14> в своем браузере, и вы должны увидеть Food Trucks во всем его черно-желтом великолепии! Раз уж мы затронули эту тему, давайте посмотрим, как выглядит наша консоль **AWS ECS**.

## Clusters

An Amazon ECS cluster is a regional grouping of one or more container instances on which you can run task requests. Each account receives a default cluster the first time you use the Amazon ECS service. Clusters may contain more than one Amazon EC2 instance type.

The screenshot shows the 'foodtrucks' cluster details. It includes a 'Create Cluster' button, a cluster name 'foodtrucks' with an 'x' icon, and summary statistics: Registered Container Instances :2, Pending tasks :0, and Running tasks :1. To the right is an 'Additional Information' sidebar with links to Documentation, Support, Forums, and Contact Us.

### Cluster : foodtrucks

Get a detailed view of the resources on your cluster.

The screenshot shows the 'Tasks' tab selected in the ECS Tasks interface. It displays the following status: Status ACTIVE, Registered container instances 2, Pending tasks count 0, and Running tasks count 1. Below this, there are buttons for 'Run new Task', 'Stop', and 'Stop All'. A filter bar shows 'Desired task status: Running'. The table lists one task: 845e2368-170d-4... (ecscompose-food...), Container Instance cb83f963-3bbb-48..., Last status RUNNING, Desired status RUNNING, and Started By ecscompose-food... . The table has columns: Task, Task Definition, Container Instan..., Last status, Desired status, and Started By.

Мы можем видеть выше, что наш ECS-кластер под названием "foodtrucks" был создан и теперь выполняет 1 задачу с 2 экземплярами контейнера. Потратьте некоторое время на просмотр этой консоли, чтобы ознакомиться со всеми представленными здесь опциями.

## Очистка

После того, как вы поиграете с развернутым приложением, не забудьте отключить кластер -

```
$ ecs-cli down --force
INFO[0001] Waiting for your cluster resources to be deleted...
INFO[0001] Cloudformation stack status stackStatus=DELETE_IN
INFO[0062] Cloudformation stack status stackStatus=DELETE_IN
```

INFO[0124] Cloudformation stack status

stackStatus=DELETE\_IN



INFO[0155] Deleted cluster

cluster=foodtrucks

Итак, у вас все получилось. Всего с помощью нескольких команд мы смогли развернуть наше потрясающее приложение в облаке AWS!



## ЗАКЛЮЧЕНИЕ

И это все! После долгого, исчерпывающего, но увлекательного урока вы теперь готовы взять мир контейнеров штурмом! Если вы следовали ему до самого конца, то вам определенно стоит гордиться собой. Вы узнали, как настроить Docker, запускать собственные контейнеры, работать со статическими и динамическими веб-сайтами и, самое главное, получили практический опыт развертывания ваших приложений в облаке!

Я надеюсь, что завершение этого руководства сделает вас более уверенными в своих способностях работать с серверами. Когда у вас появится идея создания вашего следующего приложения, вы можете быть уверены, что сможете представить ее людям с минимальными усилиями.

## Следующие шаги

Ваше путешествие в мир контейнеров только началось! Мой целью с помощью этого руководства было разжечь ваш аппетит и показать вам возможности Docker. В море новых технологий может быть трудно ориентироваться в одиночку, и такие руководства, как это, могут протянуть руку помощи. Это руководство по Docker, которое я хотел бы иметь, когда начинал. Надеюсь, оно послужило своей цели - заинтересовать вас контейнерами, чтобы

вам больше не приходилось наблюдать за происходящим со стороны.

Ниже приведены несколько дополнительных ресурсов, которые будут полезны. Для вашего следующего проекта я настоятельно рекомендую вам использовать Docker. Имейте в виду - практика делает совершенным!

## Дополнительные ресурсы

- Потрясающий Docker
- Почему именно Docker
- Docker Weekly и архивы
- Блог Codeship

Вперед, юный падаван!

## Оставляйте отзывы

Теперь, когда учебное пособие закончено, моя очередь задавать вопросы. Как вам понравилось учебное пособие? Вам показалось, что в учебном пособии полная неразбериха, или вы повеселились и чему-то научились?

Присылайте свои мысли напрямую [мне](#) или просто [создайте проблему](#). Я тоже в [Твиттере](#), так что, если вас это устраивает, не стесняйтесь кричать там!

Я был бы очень рад услышать о вашем опыте работы с этим руководством. Дайте рекомендации, как улучшить это, или сообщите мне о моих ошибках. Я хочу, чтобы это руководство стало одним из лучших вводных руководств в Интернете, и я не смогу сделать это без вашей помощи.