

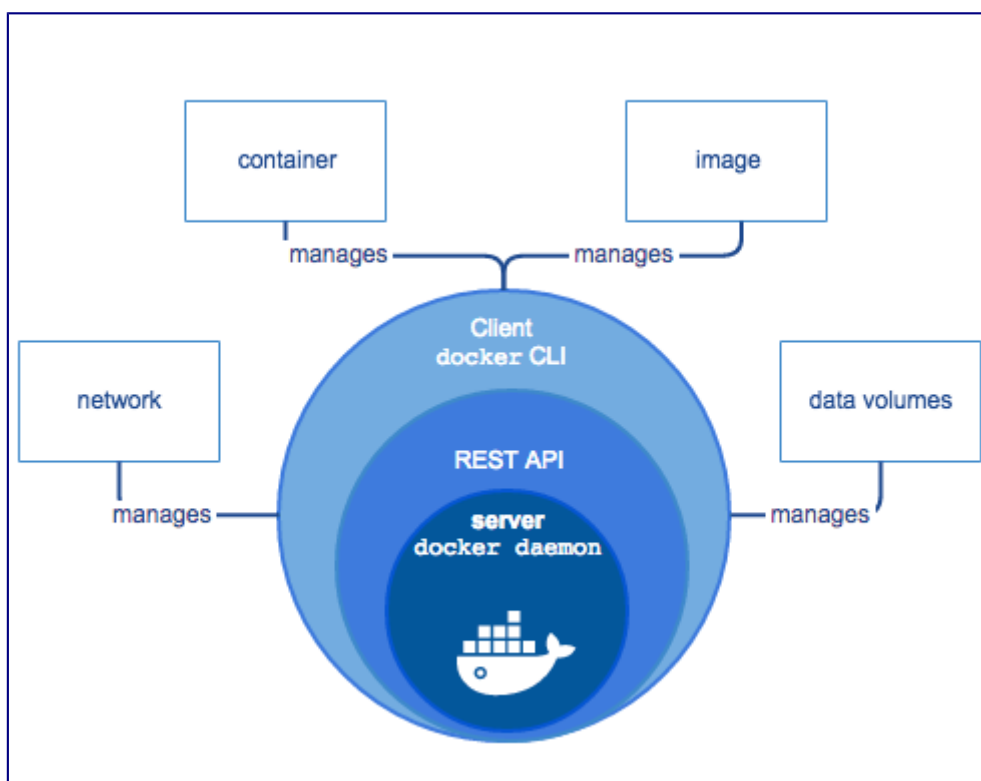
# Учебник Docker

В этом учебнике объединяются статьи, позволяющие освоить Docker в кратчайшие сроки фактически с самого нуля. Крайне желательно, чтобы вы знакомились с материалами раздела **"Обязательные для изучения материалы"** в том порядке, в котором они для вас подготовлены. Это существенно облегчит вам знакомство с этой увлекательной темой. Тем не менее, вы всегда можете выбрать именно тот раздел, который вам нравится больше всего прямо сейчас! Happy Dockering!

## Что такое Docker Engine?

*Docker Engine* — это клиент-серверное приложение, содержащее следующие основные компоненты:

- Сервер, по сути являющийся долго работающим приложением, называемым демоном
- REST API, определяющее интерфейсы для взаимодействия, которые могут использовать другие программы.
- Консольный (CLI) клиент



Консольный клиент использует Docker REST API для управления или взаимодействия с демоном Docker при помощи скриптов или непосредственных консольных команд.

Демон создает и управляет *объектами* Docker, такими как образы (images), контейнеры, сети и тома данных (data volumes)

На заметку: Docker распространяется под открытой Apache 2.0 лицензией.

## Для чего я могу использовать Docker?

*Быстрая, последовательная доставка ваших приложений*

Docker может упростить жизненный цикл разработки, позволяя разработчикам работать в стандартизованных окружениях, используя локальные контейнеры, в которых может работать ваше приложение или сервисы. Вы также можете интегрировать Docker в ваши процессы непрерывной интеграции (continuous integration) и непрерывной доставки (continuous deployment).

Представьте следующий примерный сценарий. Ваши разработчики пишут код локально и делятся своей работой с коллегами при помощи контейнеров Docker. Они могут использовать Docker для помещения своих приложений в тестовые среды и запуска автоматических и ручных тестов. Когда разработчики находят проблему, они могут исправить ее в разработческом окружении и повторно поместить контейнеры в тестовое окружение. Когда же тестирование завершено, доставить исправление в ПО конечному заказчику становится также просто как положить новые контейнеры в продуктивное окружение.

### *Адаптивная доставка и масштабируемость*

Используя контейнеры платформа Docker позволяет хорошо переносить нагрузки. Контейнеры Docker могут работать на локальной машине разработчика, на физическом или виртуальном сервере в датацентре, в Облаке, или в смешанном окружении.

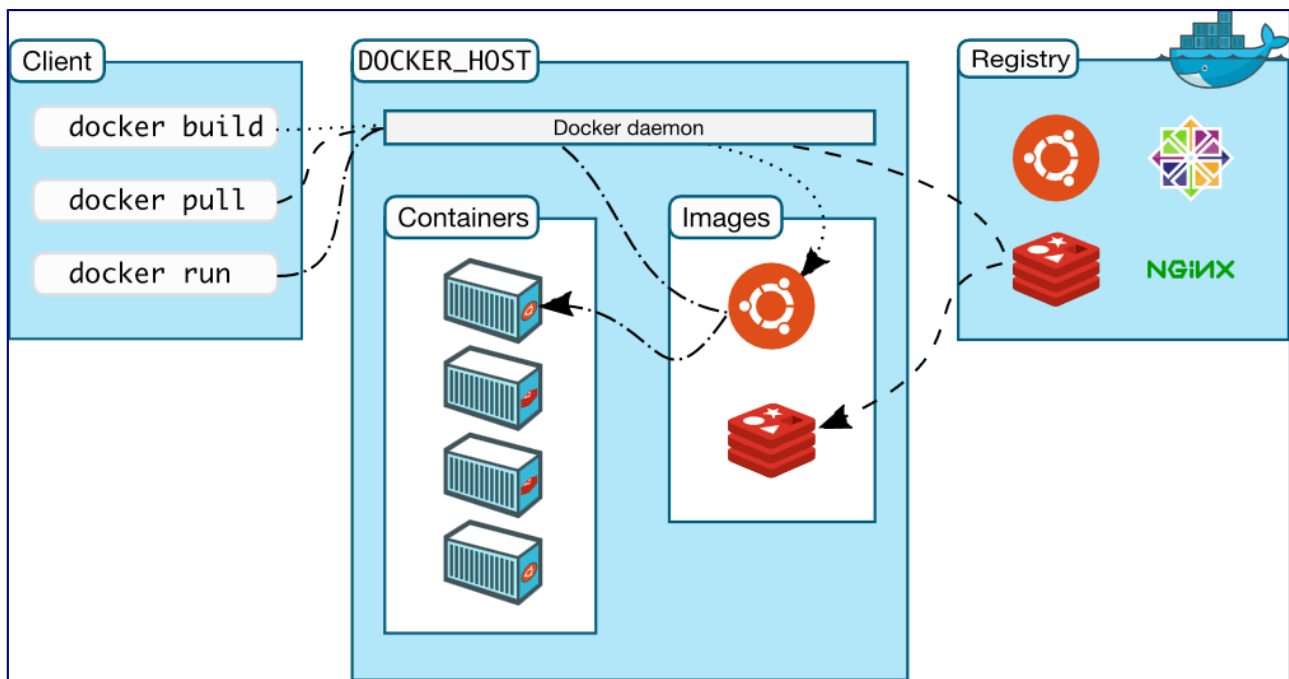
Переносимость и легковесная природа Docker также позволяют просто управлять нагрузкой, масштабировать приложения и сервисы вверх или вниз настолько быстро, насколько этого требует бизнес, почти в реальном времени.

### *Обслуживание больших нагрузок на том же самом железе*

Docker легковесен и очень быстр. Он предоставляет жизнеспособную и экономически эффективную альтернативу виртуальным машинам, запускаемым в гипервизорах, позволяя вам использовать больше ваших вычислительных мощностей для достижения ваших бизнес целей. Это особенно важно при использовании его в очень уплотненных окружениях при доставке приложений малых или средних размеров там, где необходимо сделать больше меньшими усилиями.

## **Архитектура Docker**

Docker использует клиент-серверную архитектуру. Клиент Docker общается с Docker демоном, который обслуживает тяжеловесную сборку, работу, а также размещение ваших контейнеров. Демон и клиент Docker могут работать на одной системе или же клиент Docker может подключаться к удаленному Docker демону. Клиент и демон Docker взаимодействуют при помощи REST API, через UNIX сокеты или при помощи сетевого интерфейса.



## Демон Docker

Демон Docker работает на хостовом сервере. Пользователь использует клиент Docker для взаимодействия с демоном.

## Клиент Docker

Клиент Docker в формате исполняемого файла `docker` — это основной пользовательский интерфейс для Docker. Он принимает команды и конфигурационные флаги от пользователя и взаимодействует с демоном Docker. Один клиент может взаимодействовать с множеством несвязанных демонов.

## Внутри Docker

Для понимания внутренностей Docker вам необходимо знать про образы (*images*), реестры (*registries*) и контейнеры (*containers*).

## Образы Docker

Образ Docker — это шаблон в формате «только для чтения» с инструкциями для создания контейнера Docker. Например, образ может содержать в себе ОС Ubuntu с web-сервером Apache и вашим установленным внутри web-приложением. Вы можете собрать или обновить образ с нуля, или загрузить и использовать образы, созданные другими людьми. Образ может быть основан или расширять один или более других образов. Образ Docker описан в текстовом файле `Dockerfile`, который имеет простой и вполне определенный синтаксис. Для получения большей информации о образах, смотрите «[Как работают образы Docker?](#)».

Образы Docker — это основной **строительный** компонент Docker.

## Контейнеры Docker

Контейнер Docker — это запускаемый экземпляр Docker образа. Вы можете запустить, остановить, переместить или удалить контейнер, используя Docker API или консольный клиент. Когда вы запускаете контейнер, вы можете предоставить конфигурационные метаданные такие как сетевая информация или переменные окружения. Каждый контейнер — это изолированная и безопасная платформа для приложений, однако, ему может быть предоставлен доступ к ресурсам, работающим на другом хосте

или контейнере, таким как постоянный сторадж (persistent storage) или база данных. Для получения большей информации о контейнерах, смотрите «[Как работает контейнер?](#)»

Контейнеры Docker — это **рабочий** компонент Docker.

## Реестры Docker

Реестр Docker — это библиотека образов. Реестр может быть публичным, приватным, а также может быть установлен на том же сервере, что и демон или клиент Docker или же на совсем частном сервере. Для получения большей информации о реестре, смотрите [Как работает реестр Docker?](#)

Реестры Docker — это компонент **распространения** Docker.

## Сервис Docker

*Сервис* Docker позволяет целому *рою* (swarm) Docker узлов работать одновременно, обслуживая определенное количество экземпляров реплицированной задачи (replica task), представляющей собой образ Docker. Вы можете определить количество одновременных запускаемых реплицированных задач и кластерный менеджер убедится, что нагрузка распределена. You can specify the number of concurrent replica tasks to run, and the swarm manager ensures that the load is spread равномерно по рабочим узлам. Для конечного потребителя сервис Docker представляется как одно приложение. Docker Engine поддерживает режим работы swarm mode с Docker 1.12 и выше.

Сервисы Docker — это компоненты **масштабирования** Docker.

## Как работают образы Docker?

Образы Docker — это шаблоны в формате «только чтение» из которых запускаются Docker контейнеры. Каждый образ состоит из последовательности слоев. Docker использует [union file systems](#) для объединения этих слоев в единый образ. Union FS позволяет файлам и директориям отдельных файловых систем известным как ветки (branches) быть прозрачно наложенными друг на друга, чтобы образовать единую связную файловую систему.

Эти слои одна из причин, почему Docker настолько легковесный. Когда вы изменяете образ Docker, например, в процессе обновления приложения на новую версию, собирается только новый слой, заменяет только тот слой, который обновляется. Остальные слои остаются нетронутыми. Для распространения обновления все, что вам нужно, это передать обновленный слой. Механизм работы со слоями увеличивает скорость распространения образов Docker. Docker сам определяет, какой слой должен быть обновлен в рантайме.

Образ определяется в Dockerfile. Каждый образ начинается с основного образа (base image), такого как ubuntu (основного образа ОС Ubuntu) или fedora (основного образа ОС Fedora). Вы можете также самостоятельно использовать доступные образы в качестве основы для новых, например, вам доступен базовый образ web-сервера Apache, и вы можете использовать его для всех образов ваших web-приложений. Базовый образ определяется при помощи ключевого слова FROM в Dockerfile.

**Заметка:** [Docker Store](#) — это публичный реестр и хранилище образов Docker

Образ Docker собирается на основе базового образа при помощи простого

The docker image is built from the base image using a simple, наглядного набора шагов, которые называются *инструкциями*, которые последовательно записаны в Dockerfile. Каждая инструкция создает новый слой внутри образа. Некоторые примеры инструкций Dockerfile:

- Определить базовый образ (FROM)
- Определить ответственного за поддержку (MAINTAINER)
- Выполнить команду (RUN)
- Добавить файл или директорию (ADD)
- Создать переменную окружения (ENV)
- Какой процесс запустить, когда будет запускаться контейнер из этого образа (CMD)

Docker читает `Dockerfile` когда вы запускаете сборку (build) образа, выполняет инструкции и возвращает образ.

## Как работает реестр Docker?

Реестр Docker хранит Docker образы. После того как вы собрали Docker образ, вы можете отправить (push) его в публичный реестр, такой как [Docker Store](#) или приватный реестр, закрытый вашим межсетевым экраном. Вы можете также искать уже существующие образы и скачивать (pull) их из реестра на хост.

[Docker Store](#) позволяет вам не только хранить, но и покупать и продавать Docker образы. Например, вы можете купить образ Docker, содержащий приложение или сервис от какого-либо вендора программного обеспечения и использовать его в процессе разворачивания вашего приложения в тестовое, промежуточное или продуктивное окружение, а также обновлять его и ваше приложение просто скачивая (pull) новую версию доступных образов и переразворачивая ваши контейнеры.

## Как работает контейнер?

В процессе своей работы контейнер использует Linux ядро хостовой машины, состоит из какого-то количества дополнительных файлов, которые были добавлены к нему в процессе его создания, а также метаданных, связываемых с контейнером в процессе его создания или когда контейнер уже запущен. Каждый контейнер собирается из образа. Образ определяет содержимое контейнера, какой процесс запустить, когда контейнер начинает работать, а также другие всевозможные подробности конфигурации. Образ Docker не изменяем. Когда Docker запускает контейнер из образа, он добавляет дополнительный слой поверх этого образа, который доступен для записи (при помощи UnionFS, как мы видели ранее).

## Что происходит, когда вы запускаете контейнер?

Когда вы используете `docker run` консольную команду или эквивалентный API-вызов, клиент Docker Engine отдает задачу Docker демону на запуск контейнера. В этом примере клиент сообщает Docker демону инструкцию по запуску контейнера из `ubuntu` образа Docker, (`-i`) используется, чтобы оставаться войти в контейнер в интерактивном режиме и запустить команду `/bin/bash`.

```
$ docker run -i -t ubuntu /bin/bash
```

Когда вы запускаете эту команду, Docker Engine выполняет следующие действия:

1. **Скачивает (pull) образ ubuntu:** Docker Engine проверяет наличие образа `ubuntu` в кеше на хосте, на котором работает Docker Engine. Если образ уже существует локально, Docker Engine использует его для запуска нового контейнера. В противном случае Docker Engine скачивает этот образ с [Docker Store](#).
2. **Создает новый контейнер:** Docker использует указанный образ для создания контейнера.

3. **Выделяет место на файловой системе хоста и монтирует *слой* доступный для записи:** Контейнер создан на файловой системе и доступный для записи слой добавлен к его образу.
4. **Выделяет сетевой / *bridge* интерфейс:** Создает сетевой интерфейс, который позволяет Docker контейнеру взаимодействовать с локальным хостом, на котором установлен Docker Engine.
5. **Назначает IP адрес:** Находит и присоединяет доступный IP-адрес из пула.
6. **Выполняет процесс, который вы определили:** Выполняет команду `/bin/bash`.
7. **Захватывает и предоставляет вывод приложения:** Подключается и логирует стандартный поток ввода, вывода и ошибок, чтобы показать вам, как выполняется ваше приложение, т.к. вы работаете в интерактивном режиме.

С текущего момента контейнер запущен. Вы можете управлять и взаимодействовать с ним, использовать сервисы и приложения, которые он предоставляет, а также при необходимости остановить и удалить его.

## Базовые технологии

Docker написан на [Go](#) и использует несколько возможностей Linux ядра для реализации собственной функциональности.

### Пространства имен (Namespaces)

Docker использует технологию, называемую пространством имен (namespaces) для предоставления изолированного рабочего пространства, называемого *контейнер*. Когда вы запускаете контейнер, Docker создает набор *пространств имен* для этого контейнера.

Эти пространства имен организуют слой изоляции. Каждый аспект контейнера запускается в отдельном пространстве имен и его доступ ограничен этим пространством имен.

Docker Engine использует следующие пространства имен в Linux:

- **Пространство имен *pid*:** Изоляция процесса (PID: ID процесса).
- **Пространство имен *net*:** Управление сетевыми интерфейсами (NET: Сеть).
- **Пространство имен *ipc*:** Управление доступом к IPC ресурсам (IPC: Межпроцессное взаимодействие).
- **Пространство имен *mnt*:** Управление точками монтирования файловой системы (MNT: Монтирование).
- **Пространство имен *uts*:** Изолирование идентификаторов ядра и версий. (UTS: Unix система разделения времени).

### Группы управления (Control groups)

Docker Engine в Linux также основывается на другой технологии, называемой *группами управления* (cgroups). каждая cgroup лимитирует приложение в доступе к определенному набору ресурсов. Группы управления позволяют Docker Engine разделять доступные аппаратные ресурсы между контейнерами и при необходимости применять лимиты и ограничения. Например, вы можете ограничить определенному контейнеру доступную оперативную память.

### Union FS

Файловая система Union или UnionFS — это файловая система, которая работает путем создания слоев, делая их очень легковесными и быстрыми. Docker Engine использует UnionFS как «строительные

блоки» для контейнеров. Docker Engine может использовать множество различных реализаций UnionFS, т.к. AUFS, btrfs, vfs, а также DeviceMapper.

## Формат контейнеров

Docker Engine объединяет пространства имен, группы управления и UnionFS в обертку (wrapper), называемую контейнерным форматом. Основным форматом контейнеров — это `libcontainer`. В будущем Docker возможно будет поддерживать другие форматы контейнеров, интегрируясь с такими технологиями как BSD Jails или Solaris Zones.

# Docker: управление вашими данными

- [Андрей Максимов](#) 0 комментариев

В этой статье будет рассмотрен вопрос управления данными внутри контейнеров Docker, совместное использование данных разными контейнерами, а также резервное копирование этих данных.

Согласно последней документации, в Docker существует два способа организации работы с данными:

- Docker тома (Docker volumes)
- Контейнеры Docker томов (Docker volume containers)

Но прежде чем рассказать о них чуть подробнее, необходимо еще раз обратить ваше внимание на то, как работает файловая система в контейнерах Docker. Образы Docker, из которых запускаются контейнеры, представляют собой последовательность неизменяемых (только чтение) слоев. Набор этих слоев, можно посмотреть при помощи команды `docker history`, например, для контейнера nginx версии latest это делается так:

```
$ docker pull nginx
```

```
$ docker history nginx
```

```
IMAGE CREATED BY SIZE COMMENT
a39777a1a4a6 20 hours ago /bin/sh -c #(nop) CMD ["nginx" "-g" "daemon 0 B
<missing> 20 hours ago /bin/sh -c #(nop) EXPOSE 443/tcp 80/tcp 0 B
<missing> 20 hours ago /bin/sh -c ln -sf /dev/stdout /var/log/nginx/ 22 B
<missing> 20 hours ago /bin/sh -c apt-key adv --keyserver hkp://pgp. 58.58
MB
<missing> 20 hours ago /bin/sh -c #(nop) ENV NGINX_VERSION=1.11.8-1 0 B
<missing> 20 hours ago /bin/sh -c #(nop) MAINTAINER NGINX Docker Ma 0 B
<missing> 42 hours ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0 B
<missing> 42 hours ago /bin/sh -c #(nop) ADD file:89ecb642d662ee7edb 123
MB
```

Полный вывод истории образа можно получить, добавив к последней команде ключ `--no-trunc`. А еще детализацию любого образа можно посмотреть воспользовавшись проектом <https://imagelayers.io>.

При запуске из образа нового контейнера к этому набору неизменяемых слоев присоединяется слой, который можно изменять. С этого момента, если в контейнере попытаться изменить какой-нибудь существующий файл, то этот файл будет скопирован из одного из нижележащих неизменяемых слоев, в котором он был создан, в изменяемый слой, куда и запишутся все его изменения. Версия изменяемого слоя скрывает нижележащий файл, но не удаляет его. Когда вы удаляете запущенный контейнер и



перезапускаете его из того же образа, все ваши изменения будут потеряны. Так работает [UnionFS](#), которую использует Docker, для организации своей работы с данными.

Для того, чтобы иметь возможность сохранять ваши данные, а также использовать эти данные несколькими контейнерами, Docker предоставляем вам возможность использовать так называемые тома (Docker volumes). Если совсем просто, то тома — это обычные директории (или файлы) хостовой файловой системы сервера, на котором работает Docker, монтируемые в любую директорию контейнера.

Существует несколько способов инициализировать тома, у которых есть несколько важных отличий, которые необходимо понимать. Самый прямой способ — это определить использование тома контейнером при его запуске при помощи флага `-v`:

```
$ docker run -it --name nginx_example -h nginx-container -v /opt/static
nginx /bin/bash
```

```
root@nginx-container:/# ls /opt/static/
```

Данная команда запустит контейнер с именем *nginx\_example* именем хоста контейнера (`-h`) *nginx-container* из образа *nginx* в интерактивном режиме работы (`-it`), передаст управление интерпретатору `/bin/bash`, предварительно создав безымянный том (`-v`) и смонтировав его в `/opt/static`. При этом в директория `/opt/static` будет создана автоматически, она будет жить вне UnionFS и будет доступна для записи внутри контейнера, т.е. все файлы, которые вы сохраните в этой директории будут помещены внутрь тома. Вы можете определить расположение тома на файловой системе хостового сервера, на котором работает Docker, при помощи команды `docker inspect`:

```
$ docker inspect -f "{{json .Mounts}}" nginx_example | jq .
[
  {
    "Name":
    "e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d",
    "Source":
    "/var/lib/docker/volumes/e01cb937734c02790c94c14339bc563159a88c025e266d298
    d48398352ea477d/_data",
    "Destination": "/opt/static",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

В этом примере используется фильтр (`-f`) всего вывода `docker inspect nginx_example`, показывающий только часть содержимого, которое относится к блоку `Mounts`. При этом вывод форматируется утилитой [jq](#).

После того, как мы знаем имя тома (ключ `Name`), мы можем получить ту же информацию при помощи команды

`docker volume inspect`:

```
$ docker volume inspect
e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d
[
```



```
{
  "Name":
  "e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d",
  "Driver": "local",
  "Mountpoint":
  "/var/lib/docker/volumes/e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d/_data",
  "Labels": null,
  "Scope": "local"
}
```

В обоих случаях вывод показывает вам, что Docker смонтировал `/opt/static` внутри контейнера как директорию

`/var/lib/docker/volumes/e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d/_data`. Чтобы убедиться в этом, создайте, например, файл `test` на хостовом сервере, на котором запущен Docker, внутри этой директории.

```
$ sudo touch
/var/lib/docker/volumes/e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d/_data/test
```

А затем посмотрите на содержимое `/opt/static` внутри контейнера.

```
root@nginx-container:/# ls /opt/static
test
```

Если вы отключились от интерактивного режима работы с контейнером, выполните команду `docker start` для запуска контейнера с интерактивным (`-ai`) к нему подключением:

```
$ docker start -ai nginx_example
```

Вы увидели этот результат только потому что `/opt/static` — это всего лишь директория на хостовом сервере, которая была смонтирована внутрь контейнера.

Тот же самый эффект будет достигнут при использовании инструкции `VOLUME` в `Dockerfile`:

```
FROM nginx
VOLUME /opt/static
```

Еще один способ создавать тома — это использовать команду `docker volume create`:

```
$ docker volume create --name nginx_static_data
nginx_static_data
```

Который в последствие может быть подключен к вашему контейнеру. Обратите внимание, том может быть подключен **только** к новому контейнеру, поэтому старый контейнер предварительно придется удалить:

```
$ docker rm nginx_example
nginx_example
```

```
$ docker run -it --name nginx_example -h nginx-container -v
nginx_static_data:/opt/static nginx /bin/bash
```

```
root@nginx-container:/# ls /opt/static
```

Есть еще один часто используемый способ создания тома, реализуемый только при помощи параметра `-v`, монтирующий указанную директорию хостового сервера внутрь контейнера. Для этого создайте на хостовом сервере директорию `folder_test`, внутри этой директории файл `test_file`, а затем пересоздайте контейнер, указав полный путь до директории `folder_test`:

```
$ mkdir folder_test
$ touch folder_test/test_file
$ docker rm nginx_example
nginx_example

$ docker run -it --name nginx_example -h nginx-container -v
/home/docker/folder_test:/opt/static nginx /bin/bash
root@nginx-container:/# ls /opt/static/
test_file
```

Как видите, содержимое директории `folder_test` стало доступно внутри запущенного контейнера. Этот способ работы с томами очень удобно использовать, например, в процессе разработке приложений. В целях сохранения свойства переносимости контейнеров, директория, используемая для тома на хостовом сервере, не может быть указана в `Dockerfile` (ведь эта директория совершенно не обязательно должна быть доступна во всех контейнерах). В случае использования этой формы работы с томами (монтирование тома при помощи флага `-v` в процессе создания контейнера) файлы, возможно содержащиеся внутри директории образа, не копируются внутрь тома (в данном случае в директорию на хостовом сервере). Docker не управляет такого рода томами как показано в прошлых примерах, и поэтому такие тома не появятся в выводе команды `docker volume ls` и никогда не будут удалены демоном Docker.

## Совместное использование данных контейнерами

Для того чтобы дать доступ одному контейнеру к томам другого, вы должны указать аргумент `--volumes-from` для команды `docker run`. Например:

```
$ docker run -it --name nginx_example1 -h nginx-container1 --volumes-
from=nginx_example nginx /bin/bash

root@nginx-container1:/# ls /opt/static/
test_file
```

Это работает вне зависимости от того, запущен контейнер `nginx_example` или нет. Причем, том не будет удален до тех пор, пока хотя бы один контейнер к нему подключен. Вы можете также монтировать том указывая в качестве значения аргумента `-v` имя тома, например:

```
$ docker rm nginx_example1
nginx_example1

$ docker run -it --name nginx_example1 -h nginx-container1 -v
e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d:/opt/stat
ic nginx /bin/bash

root@nginx-container1:/# ls /opt/static/
test_file
```

## Контейнеры Docker томов (Docker volume containers)

До момента появления в Docker команд `docker volume` широко использовались «контейнеры томов» для хранения и организации совместной работы с данными. Этот подход означает, что контейнер по своей сути становился «пространством имен» (namespace) для данных. Тем не менее в современных версиях Docker этот подход никогда не нужно использовать. Вместо этого просто создайте том с нужным вам именем при помощи команды `docker volume create --name`.

## Права доступа

Часто вам может потребоваться выставить привилегии и владельца содержимого Docker тома или поместить внутрь тома данные по-умолчанию или конфигурационные файлы. Ключевой важный момент в процессе создания образа вашего контейнера это то, что после инструкции **VOLUME** в Dockerfile Docker не сможет выполнить никаких изменений внутри тома, например:

```
FROM nginx:latest
RUN useradd foo
VOLUME /opt/static
RUN touch /opt/static/some_file
RUN chown -R foo:foo /opt/static
```

Не будет работать, как ожидается. Нам необходимо выполнить команду `touch` на файловой системе будущего образа, но она будет исполнена в томе временного слоя, использующегося при сборке. Работать будет немного по-другому:

```
FROM nginx:latest
RUN useradd foo
RUN mkdir /opt/static && touch /opt/static/static_file
RUN chown -R foo:foo /opt/static
VOLUME /opt/static
```

Docker достаточно интеллектуален, чтобы скопировать любые файлы, существующие в образе в нижележащих слоях в точку монтирования тома и выставить правильные привилегии. Этого не произойдет, если вы смонтируете директорию на хостовом сервере в качестве тома.

## Удаление тома

Если вы всегда использовали команду `docker rm` для удаления ваших контейнеров, велики шансы, что у вас есть большое количество не удаленных томов, занимающих место.

Тома автоматически удаляются только в том случае, если родительский контейнер был удален при помощи команды `docker rm -v` (использование флага `-v` очень важно) или же если флаг `-rm` был использован во время выполнения команды `docker run`. И даже после этого том будет удален только тогда, когда ни один другой контейнер не ссылается на него. Тома в виде директорий, смонтированные внутрь контейнера из хостового сервера, никогда не удаляются.

Чтобы посмотреть на все тома, существующие на хостовом сервере, выполните команду `docker volume ls`:

```
$ docker volume ls
DRIVER VOLUME NAME
```

```
local e01cb937734c02790c94c14339bc563159a88c025e266d298d48398352ea477d
local nginx_static_data
```

Для удаления всех не использующихся томов, выполните команду:

```
$ docker volume rm $(docker volume ls -q)
```

## Резервное копирование, восстановление и миграция томов

Еще одна функция, которую можно выполнять с томами — это их резервное копирование, восстановление и использование этих операций для осуществления миграции данных. Резервное копирование выполняется при помощи использования `—volumes-from` аргумента в процессе создания нового контейнера, который будет архивировать данные уже существующего:

```
$ docker run --rm --volumes-from nginx_example -v $(pwd):/backup ubuntu tar
cvf /backup/backup.tar /opt/static
```

Эта команда запустит новый контейнер из образа *ubuntu* и запустит внутри него команду архивирования директории `/opt/static` (команда для исполнения внутри контейнера `tar cvf /backup/backup.tar /opt/static`), предварительно смонтировав текущую директорию `$(pwd)` в директорию `/backup` контейнера.

Восстановить данные тома в тот же самый контейнер или в другой контейнер в любом другом месте (`backup.tar` легко передается при помощи `SCP`) можно, например, командой:

```
$ docker volume create --name new_volume_for_static_data
$ docker run -v new_volume_for_static_data:/opt/static --name
nginx_restored nginx /bin/bash
$ docker run --rm --volumes-from nginx_restored -v $(pwd):/backup ubuntu
bash -c "cd /opt/static && tar xvf /backup/backup.tar --strip 1"
```

Эти команды могут быть использованы вами для автоматизации резервного копирования, восстановления или миграции данных томов ваших контейнеров.

## Резервное копирование, восстановление и миграция контейнеров

Если же в процессе работы контейнера изменялись данные не тома, а непосредственно контейнера, то эти изменения тоже можно сохранить, например в образ при помощи команды `docker commit`:

```
$ docker commit nginx_example nginx_example_backup:19.01.2017
sha256:0dcad379614806b0485f2cad48fcd5208e413363e7a04819ca7edf81b92cf280
```

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx_example_backup	19.01.2017	0dcad3796148	4 seconds ago	181.5 MB

Далее вы можете сохранить этот образ в публичный или частный Docker реестр при помощи команды `docker push`, или же вы можете сохранить образ в `tar`-файл для передачи на другой удаленный сервер:

```
$ docker save -o nginx_example_backup_19.01.2017.tar
nginx_example_backup:19.01.2017
```

Чтобы впоследствии создать из него образ, но уже на другом удаленном сервере, где установлен Docker:

```
$ docker load -i nginx_example_backup_19.01.2017.tar
```

По материалам:

- [Manage data in containers](#)
- [Understanding Docker Volumes](#)
- [Docker backup – Easy steps to backup and restore your containers](#)

....

# Docker Machine

- [Андрей Максимов](#) 0 комментариев

Вне зависимости от того, в какой операционной системе вы работаете, начинать работу с Docker необходимо с того, чтобы понимать, с каким Docker Engine (см. [Docker: Введение](#)) вы сейчас работаете и в каком состоянии он находится. Для решения этой задачи и создана Docker Machine.

Docker Machine используется для:

- Установки и запуска Docker на Mac или Windows
- Установка, настройка и управление множества удаленных Docker хостов
- Установка, настройка и управление Swarm кластерами

Docker Machine — это утилита, которая нужна вам для того, чтобы устанавливать Docker Engine на удаленные физические или виртуальные сервера, а также управлять ими при помощи команды `docker-machine`. Вы можете использовать Docker Machine для создания Docker хостов на вашем Mac или Windows ПК, в корпоративной сети вашей компании, в вашем датацентре или в различных Облаках, т.к. например, AWS или [Digital Ocean](#).

Используя команды `docker-machine`, вы можете запустить, проверить, остановить, перезапустить управляемый ей хост, обновить клиент и демон Docker, а также настроить клиент Docker на работу с удаленным хостом.

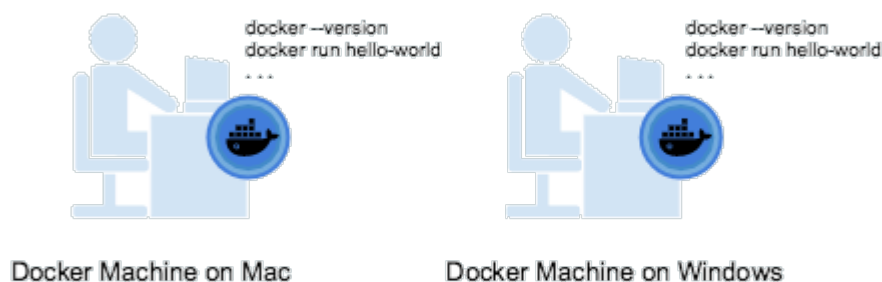
## Почему я должен использовать Docker Machine?

Docker Machine позволяет вам легко устанавливать, настраивать и управлять множеством Docker хостов, каждый из которых может работать на различном Linux дистрибутиве.

Более того, Docker Machine позволяет вам запускать Docker на старых Mac или Windows системах.

У Docker Machine есть два основных сценария использования.

- У меня есть устаревший ПК, и я хочу иметь возможность запустить Docker на Mac



или Windows

Если вы работаете на устаревшем Mac или Windows ноутбуке или рабочей станции, которые не удовлетворяют последним требованиям [Docker для Mac](#) или [Docker для Windows](#), значит, вам необходима Docker Machine, чтобы все таки “запустить Docker” (имеется в виду, Docker Engine) у себя локально. Во время такой установки Docker Machine на ваш Mac или Windows при помощи [Docker Toolbox](#), на самом деле у вас на ПК устанавливается локальная виртуальная машина, внутри которой и будет работать Docker Engine. Docker Machine в данном случае даст вам возможность настроить подключение клиента Docker (команда `docker`) к Docker Engine в этой виртуальной машине.

- Я хочу установить Docker на удаленные виртуальные машины или физические серверы



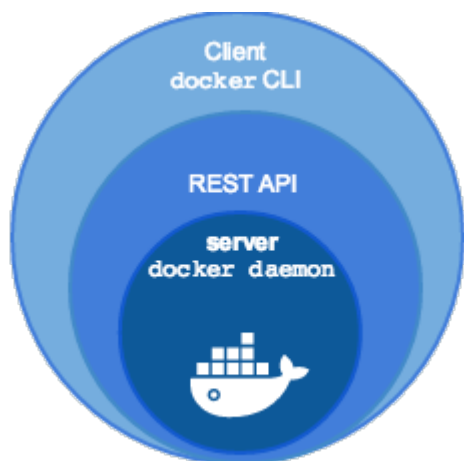
Docker Engine — это нативное приложение для Linux систем. Если у вас Linux в качестве основной системы и вам нужно иметь возможность запускать команду `docker`, все что вам нужно сделать — это скачать и установить Docker Engine. Тем не менее, если вам нужен эффективный способ работать с множеством Docker хостов в сети, в Облаке или же локально, вам нужна Docker Machine.

Вне зависимости от того, используете ли вы Mac, Windows или Linux, вы можете установить Docker Machine и использовать команду `docker-machine` для подготовки и управления большим количеством Docker хостов. Она автоматически устанавливает на хост Docker, т.е. устанавливает Docker Engine, затем помогает настроить и клиент `docker`.

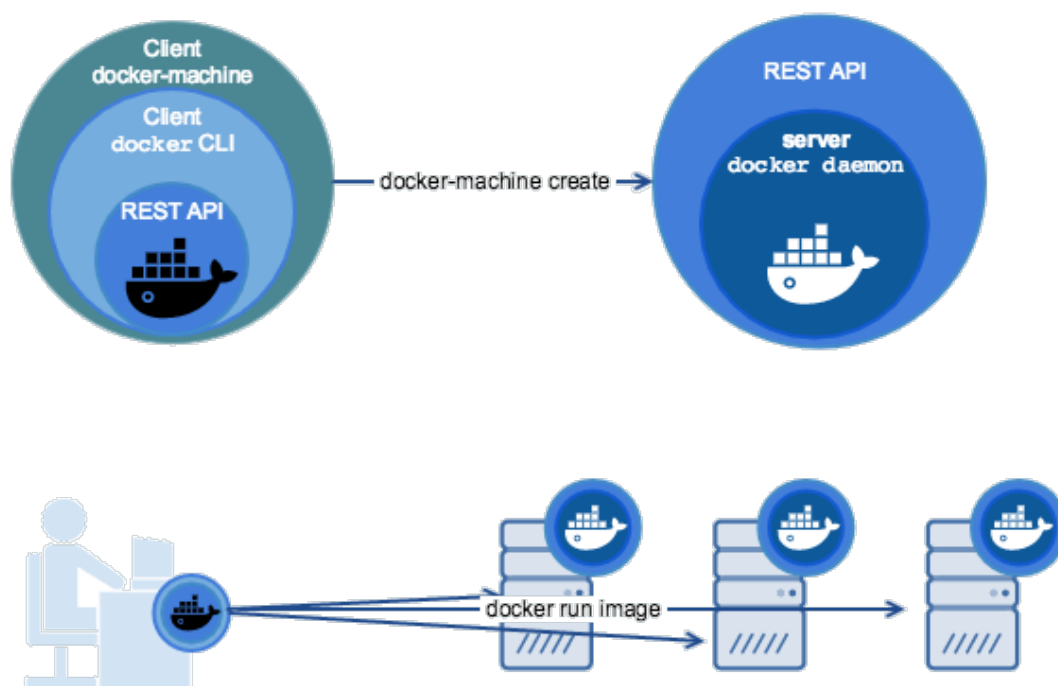
## Какова разница между Docker Engine и Docker Machine?

Когда говорят “Docker”, обычно имеют в виду **Docker Engine**, т.е. клиент-серверное программное обеспечение, состоящее из демона Docker, REST API, который определяет интерфейсы взаимодействия с демоном и клиентский консольный командный интерфейс, т.е. клиента `docker`, который общается с демоном при помощи обертки над REST API. Docker Engine принимает команды `docker`, такие

как `docker run <image>` для запуска контейнера из образа, `docker ps` для отображения запущенных контейнеров, `docker images` для отображения доступных образов и т.д.



**Docker Machine** — это утилита для подготовки и управления вашими докеризированными хостами (имеются в виду хосты с установленным на них Docker Engine). Обычно Docker Machine устанавливается на вашу локальную систему. У Docker Machine есть свой консольный клиент `docker-machine` так же, как клиент для Docker Engine — `docker`. Вы можете использовать Docker Machine для установки Docker Engine на один или более виртуальных серверов. Эти виртуальные серверы могут быть локальными или удаленными. Докеризированные хосты при этом называются управляемыми машинами (“*machines*”).



**Замечание:** для того, чтобы Docker Machine могла подготовить Docker хост, этот Docker хост должен иметь подключение к интернет.

## Использование Docker Machine

После установки Docker Machine посмотреть на список управляемых ей Docker хостов можно командой:



```
$ docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
default - virtualbox Stopped Unknown
```

Я использую Mac и в качестве Docker хоста у меня используется виртуальная машина, управляемая VirtualBox. Как видно из примера, эта виртуальная машина называется **default** и в настоящий момент она остановлена. Чтобы запустить эту виртуальную машину и иметь возможность использовать ее в качестве Docker хоста, нужно выполнить команду:

```
$ docker-machine start default
Starting "default"...
(default) Check network to re-create if needed...
(default) Waiting for an IP...
Machine "default" was started.
Waiting for SSH to be available...
Detecting the provisioner...
Started machines may have new IP addresses. You may need to re-run the
`docker-machine env` command.
```

Docker хост успешно запущен. Проверим этот факт:

```
$ docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
default * virtualbox Running tcp://192.168.99.100:2376 v1.12.4
```

Звездочка (\*) показывает, что данный Docker хост в настоящий момент активен, т.е. при запуске утилиты **docker**, все ее команды будут выполняться на этом хосте. Если звездочка в выводе отсутствует, то необходимо настроить переменные окружения для корректной работы клиента Docker. Посмотреть переменные окружения для работы с Docker хостом **default** можно командой:

```
$ docker-machine env default
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/amaksimov/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
# eval $(docker-machine env default)
```

Эта команда сама подсказывает, как применить эти переменные в вашем окружении, чтобы настроить клиента:

```
$ eval $(docker-machine env default)
```

После выполнения данной команды, клиент **docker** будет настроен на работу с Docker хостом **default**.

Помимо подготовки и настройки Docker хоста Docker Machine может осуществлять к нему ssh подключение:

```
$ docker-machine ssh default
```

```
Boot2Docker version 1.12.4, build HEAD : d0b8fd8 - Tue Dec 13 18:21:26 UTC
2016
```

```
Docker version 1.12.4, build 1564f02
```

docker@default:~\$

При помощи плагинов Docker Machine при помощи команды `docker-machine create` может сразу запускать виртуальные серверы в Облаке, например, AWS или [DigitalOcean](#). Если же у вас есть просто Linux сервер, которым необходимо управлять удаленно, то предлагаю вам ознакомиться со статьей [Использование Docker в Облаке КРОК](#), где этот вопрос освещается подробнее.

....

# Docker Compose: Введение

- [Андрей Максимов](#) 0 комментариев

Docker Compose — это утилита для определения и запуска многоконтейнерных Docker приложений. При работе с Docker Compose вы используете `docker-compose.yml` файл для настройки сервисов вашего приложения. Далее, используя всего лишь одну команду, вы создаете и запускаете сервисы из вашей конфигурации.

Docker Compose отлично подходит для разработки, тестирования, создания промежуточных окружений, а также процессов непрерывной интеграции (CI).

Использование Docker Compose обычно представляет собой трех этапный процесс.

1. Определить окружение вашего приложения при помощи `Dockerfile` так, чтобы оно могло быть воспроизведено где угодно.
2. Определить сервисы, которые составляют ваше приложение в `docker-compose.yml` так, чтобы они могли быть запущены вместе в изолированном окружении.
3. Наконец, запустить `docker-compose up` и Docker Compose запустит определенное вами приложение.

Файл `docker-compose.yml` выглядит следующим образом:

```
version: '2'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

У Docker Compose есть несколько команд, предназначенных для управления всем жизненным циклом вашего приложения:

- Запуск, остановка и пересборка сервисов
- Просмотр и получение статуса запущенных сервисов
- Отображение логов запущенных служб
- Выполнение одной команды за раз на сервис

# Упаковка простого Flask приложения в Docker контейнер

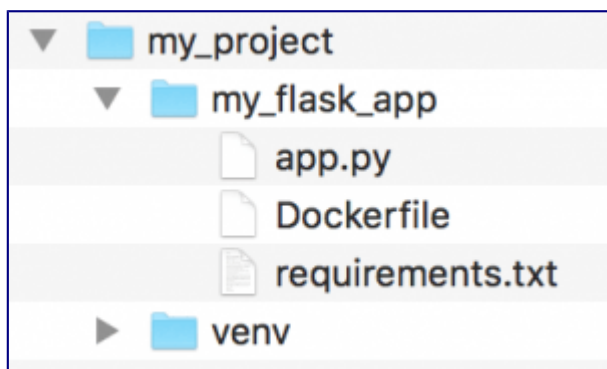
- [Андрей Максимов](#) 0 комментариев

Из этой статьи вы узнаете, как создать и упаковать простое Flask приложение в Docker контейнер.

## Требования

- Docker
- python 2.7 или выше

К окончанию статьи у вас должна получиться следующая структура директорий и файлов



## Создание простого Flask приложения

Создайте пустой каталог, в котором будут содержаться файлы проекта, а внутри него virtualenv окружение

```
$ mkdir my_project
$ cd my_project
$ virtualenv venv
```

Активируйте окружение и установите в него Flask

```
$ source venv/bin/activate
$ pip install flask
```

Создайте отдельную директорию, в которой будет находиться Flask приложение

```
$ mkdir my_flask_app
```

Сохраните список установленных в окружении пакетов в requirements.txt в каталоге my\_flask\_app

```
$ pip freeze > my_flask_app/requirements.txt
```

Содержимое my\_flask\_app/requirements.txt будет следующим

```
$ cat my_flask_app/requirements.txt
click==6.7 Flask==0.12
itsdangerous==0.24
Jinja2==2.9.4
MarkupSafe==0.23
Werkzeug==0.11.15
wheel==0.24.0
```

Создайте файл `app.py` в каталоге `my_flask_app`, который будет содержать наше простое python Flask Web-приложение

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from flask import Flask
app = Flask(__name__)

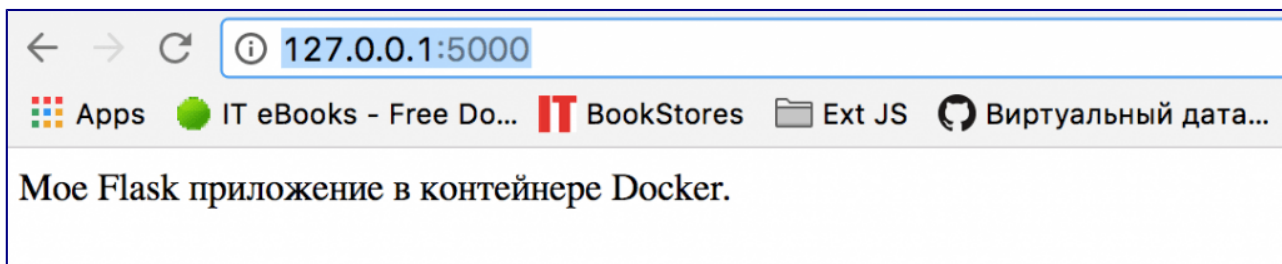
@app.route('/')
def hello_world():
    return 'Moe Flask приложение в контейнере Docker.'
```

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Проверьте работоспособность вашего приложения, запустив его командой

```
$ python my_flask_app/app.py
* Running on (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 160-707-431
```

После чего в браузере на странице `http://127.0.0.1:5000/` можно будет увидеть строку «Moe Flask приложение в контейнере Docker.»



## Упаковка Flask приложения в контейнер

### Dockerfile

Для сборки Docker образа используется специальный [Dockerfile](#), который необходимо создать в директории `my_flask_app`. Содержимое Dockerfile файла в нашем случае будет:

```
$ cat my_flask_app/Dockerfile
FROM ubuntu:latest
MAINTAINER Andrey Maksimov 'maksimov.andrei@gmail.com'
RUN apt-get update -y
RUN apt-get install -y python-pip python-dev build-essential
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ['python']
CMD ['app.py']
```

## Описание инструкций Dockerfile

Инструкция	Описание
FROM ubuntu:latest	В качестве базового образа будет использоваться latest версия образа Ubuntu.
MAINTAINER Andrey Maksimov «maksimov.andrei@gmail.com»	Справочная информация. Не обязательный параметр. Укажите контакты создателя образа.
RUN apt-get update -y	Обновить информацию о репозиториях внутри контейнера
RUN apt-get install -y python-pip python-dev build-essential	Установить внутрь контейнера пакеты: python-pip, python-dev, build-essential
COPY ./app	Скопировать содержимое текущей директории «.» в директорию /app внутри образа. Внимание: текущей директорией в процессе сборки будет считаться директория, содержащая Dockerfile, т.е. в нашем случае my_flask_app/
WORKDIR /app	Сменить рабочую директорию внутри контейнера. Все команды далее будут запускаться внутри директории /app внутри контейнера
RUN pip install -r requirements.txt	Установить зависимости, сохраненные вами в requirements.txt. Данная команда установит Flask и все, что необходимо для его запуска внутри контейнера.
ENTRYPOINT [«python»]	Запускать интерпретатор python по умолчанию
CMD [«app.py»]	В сочетании с использованием ENTRYPOINT интерпретатору python будет передан дополнительный аргумент app.py. Другими словами, во время запуска контейнера последней инструкцией будет выполнена команда python app.py из директории /app

## Создание образа контейнера

Для создания собственного контейнера, содержащего ваше простое Flask приложение, находясь внутри директории my\_project необходимо выполнить команду:

```
$ docker build -t my_flask_app:v0.1 my_flask_app/
```

Ключ **-t** предназначен для того, чтобы присвоить вашему образу метку (label) «my\_flask\_app» и его версию «v0.1». Метка и версия могут быть произвольными. Если в качестве версии использовать «latest», то собранный вами образ будет помечен как имеющий самую свежую версию.

В процессе выполнения команды как описано в вашем Dockerfile будет произведена загрузка последнего образа ubuntu, внутри него установлены все необходимые зависимости, создана директория /app, в которую будет помещено содержимое директории my\_flask\_app/, установлены все зависимости из файла requirements.txt, а сам образ настроен на запуск вашего Flask приложения из директории /app, находящейся внутри образа.

После успешного выполнения команды в списке ваших образов появится только что созданный образ my\_flask\_app:v0.1

```
$ docker images
```

```
REPOSITORY TAG IMAGE ID CREATED SIZE
```

```
my_flask_app v0.1 afe8322ae297 8 minutes ago 440.9 MB
```

## Запуск контейнера из собранного образа

Для запуска Docker контейнера из подготовленного вами образа необходимо выполнить команду

```
$ docker run -d -p 5000:5000 my_flask_app:v0.1
```

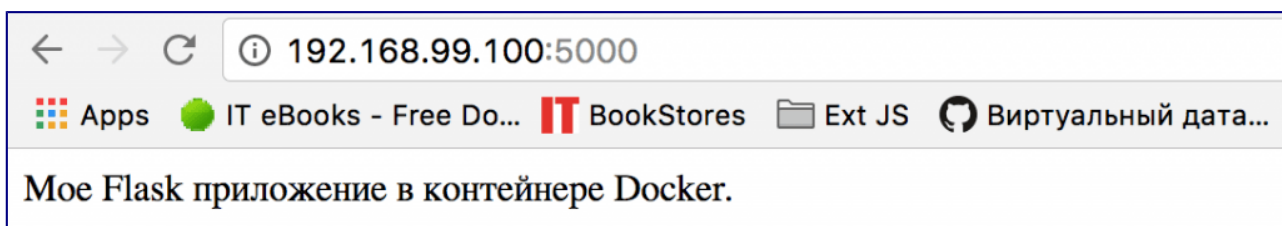
Ключ **-d** предназначен для запуска вашего контейнера в фоновом режиме (необходим для того, чтобы вернуть управление терминалу, в котором вы работаете).

Ключ **-p** заставит Docker Machine пробрасывать подключения, приходящие на порт 5000 внешнего адреса Docker Machine на порт 5000 контейнера, на котором будет слушать подключения ваше Flask приложение. Использование `-p [порт на Docker Machine:порт контейнера]`.

Проверить, что контейнер успешно запущен можно командой

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5a166878f263 my_flask_app:v0.1 'python app.py' About a minute ago Up About a minute 0.0.0.0:5000->5000/tcp stoic_allen
```

После чего в браузере по адресу `http://192.168.99.100:5000/` будет выведена строка «Мое Flask приложение в контейнере Docker.» так же, как это было сделано при тестовом запуске Flask приложения локально.



Адрес 192.168.99.100 обычно используется Docker Machine для публикации портов ваших контейнеров. Этот адрес обычно сообщается в выводе настроек вашей виртуальной машины в переменной `DOCKER_HOST`.

Посмотреть список всех созданных вами Docker машин, выполните команду

```
$ docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
default * virtualbox Running
tcp://192.168.99.100:2376 v1.12.4
```

Посмотреть настройки Docker машины с именем default выполните команду

```
$ docker-machine env default
export DOCKER_TLS_VERIFY='1'
export DOCKER_HOST='tcp://192.168.99.100:2376'
export DOCKER_CERT_PATH='/Users/amaksimov/.docker/machine/machines/default'
export DOCKER_MACHINE_NAME='default'
# Run this command to configure your shell:
# eval $(docker-machine env default)
```

## Остановка и удаление запущенного контейнера

Для того, чтобы выполнить какие-либо операции с тем или иным контейнером сначала необходимо узнать идентификатор.

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5a166878f263 my_flask_app:v0.1 'python app.py' 3 hours ago Up 3 hours 0.0.0.0:5000->5000/tcp stoic_allen
```

В моем случае — это **5a166878f263**.

Остановка запущенного контейнера выполняется командой

```
$ docker stop 5a166878f263
```

А для того, чтобы удалить запущенный контейнер, необходимо выполнить команду

```
$ docker rm 5a166878f263
```

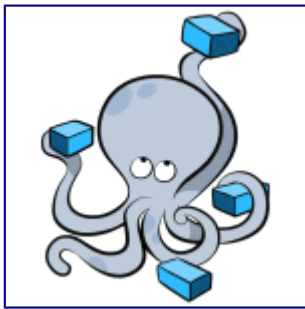
## Заключение

Упаковывать любые приложения в контейнеры крайне просто, а их запуск после занимает считанные мгновения. В следующей статье вы узнаете как использовать Docker Compose.

Итоговый Docker образ, созданный в статье можно найти на [Docker Hub](#), а сходный код приложения на [GitHub](#).

# Работа с контейнерами в Docker Compose

- [Андрей Максимов](#) 0 комментариев



В статье «[Упаковка простого Flask приложения в Docker контейнер](#)» вы создали простое Flask приложение и заставили его запускаться внутри контейнера. В этой статье мы продолжим начатый вами проект, добавив к нему поддержку [Docker Compose](#), предназначенного для автоматизации процесса запуска большого количества связанных контейнеров в сложных приложениях.

## Файл docker-compose.yml

В директории нашего проекта my\_project создайте файл docker-compose.yml, в который необходимо добавить следующее содержимое

```
$ cat docker-compose.yml
web:
  build: ./my_flask_app
  ports:
    - "5000:5000"
  volumes:
    - ./my_flask_app:/app
```

Описание содержания docker-compose.yml

Строка	Описание
web:	Название компонента приложения, управляемого Docker Compose. Каждый компонент запускается в отдельном контейнере или их группе
build: ./my_flask_app	Обозначение того, что данный компонент будет запускаться из образа, который необходимо собрать. Инструкции по сборке будут искаяться в



Строка	Описание
	Dockerfile в директории ./my_flask_app относительно docker-compose.yml файла
ports: - "5000:5000"	Обозначение того, что необходимо выполнить публикацию порта 5000 данного компонента в порт 5000, доступный по внешнему IP-адресу хоста (или виртуальной машины), на котором будет запущено приложение.
volumes: - ./my_flask_app:/app	Обозначение того, что внутри контейнера, отвечающего за данный компонент приложения, необходимо подключить Docker диск ( <a href="#">Docker Volume</a> ). В данном случае в роли Docker диска будет выступать директория ./my_flask_app, которая будет смонтирована в директорию /app внутри контейнера. Эта операция выполнена для того, чтобы можно было легко менять Flask приложение и наблюдать изменения обновляя окно браузера, а не выполняя пересборку всего контейнера целиком. Попробуйте изменить вывод функции hello_world в файле ./my_flask_app/app.py, чтобы в этом убедиться

Если вы еще не остановили запущенный в прошлой статье контейнер, сделайте это, чтобы не возникло конфликта с использованием порта 5000 в процессе запуска нового контейнера, который будет управляться Docker Compose.

## Работа с контейнерами в Docker Compose

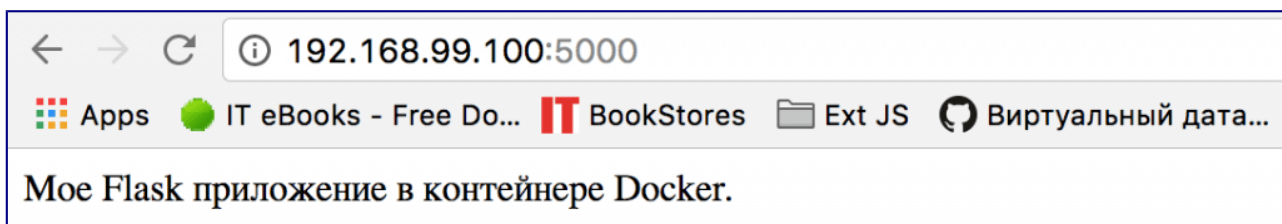
### Запуск контейнера

Сборка и запуск нового контейнера выполняется командой

```
$ docker-compose up -d
```

Ключ **-d** необходим, чтобы запустить ваше приложение, состоящее из одного контейнера, и вернуть управление терминалу, в котором вы вызывали docker-compose.

После успешной сборки и запуска контейнера, вы сможете снова увидеть вывод «Мое Flask приложение в контейнере Docker.» в браузере по адресу <http://192.168.99.100:5000/>.



Для того, чтобы пересобрать ваше приложение после сделанных вами изменений выполните команду

```
$ docker-compose build
```

А перезапуск контейнера вместе с предварительной пересборкой, выполняется командой

```
$ docker-compose up -d --build
```

Просмотр запущенных docker-compose контейнеров, а также информации по опубликованным ими портам осуществляется командой

```
$ docker-compose ps
```

```

Name Command State Ports
-----
myproject_web_1 python app.py Up 0.0.0.0:5000->5000/tcp

```

## Остановка и удаление приложения

Остановка приложения выполняется командой

```
$ docker-compose stop  
Stopping myproject_web_1 ... done
```

Одновременная остановка и удаление выполняются командой

```
$ docker-compose down  
Stopping myproject_web_1 ... done  
Removing myproject_web_1 ... done
```

## Заключение

Использование Docker Compose существенно облегчает сборку, запуск и остановку вашего приложения. Наибольший эффект от его использования достигается при управлении приложением, состоящим из нескольких связанных контейнеров, о чем я расскажу в следующей статье.

# Docker Compose: запуск Flask приложения с MongoDB

- [Андрей Максимов](#) 6 комментариев

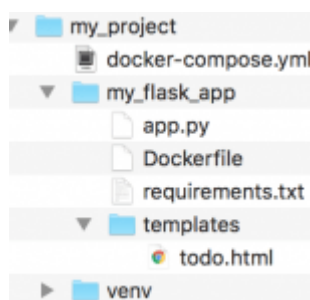
В прошлой статье «[Работа с контейнерами в Docker Compose](#)» вы узнали как использовать Docker Compose для автоматизации сборки и запуска [простого Flask приложения](#).

В этой статье вы узнаете как добавить к существующему приложению MongoDB, превратив его в прототип записной книжки дел.

## Требования

- Docker (1.11.0 или выше)
- Docker Compose (1.8.0 или выше)
- Python 2.7 или выше

## Структура файлов проекта



## Обновление Flask приложения

Обновим содержимое Flask приложения для того, чтобы оно могло использовать MongoDB в качестве СУБД, отредактировав файл my\_flask\_app/app.py.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import os
from flask import Flask, redirect, url_for, request, render_template
from pymongo import MongoClient

app = Flask(__name__)

client = MongoClient('mongodb', 27017)
db = client.tododb

@app.route('/')
def todo():

    _items = db.tododb.find()
    items = [item for item in _items]

    return render_template('todo.html', items=items)

@app.route('/new', methods=['POST'])
def new():

    item_doc = {
        'task': request.form['task'],
        'description': request.form['description']
    }
    db.tododb.insert_one(item_doc)

    return redirect(url_for('todo'))

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

Важно отметить, что в строке подключения в качестве имени хоста используется «**mongodb**». Это имя хоста получает контейнер, запускаемый Docker Compose по имени запускаемого сервиса (**mongodb**). Контейнер **web** знает о контейнере **mongodb**, т.к. в его описании присутствует параметр *links*, указывающий на этот сервис (см. Обновление *docker-compose.yml*).

Также в директории *my\_flask\_app* создадим директорию *templates*, в которой необходимо создать шаблон страницы *todo.html*, использующийся в вашем приложении.

```
<h1>Новая запись</h1>
<form action="/new" method="POST">
<table border=0>
<tr>
<td>Дело</td>
```

```

    <td><input type="text" name="task"></input></td>
<tr>
    <td>Описание</td>
    <td><input type="text" name="description"></input></td>
</tr>
<tr>
    <td></td><td><input type="submit" value="Сохранить"></input></td>
</tr>
</table>
</form>

```

```

<hr>

```

```

<h1>Список дел</h1>
<table border=1>
<thead>
<tr>
    <td><b>Дело</b></td>
    <td><b>Описание</b></td>
</tr>
</thead>
<tbody>
{% for item in items %}
<tr>
    <td>{{ item.task }}</td>
    <td>{{ item.description }}</td>
</tr>
{% endfor %}
</tbody>
</table>

```

Для работы с MongoDB нашему приложению необходима библиотека pymongo, которую необходимо добавить в качестве зависимостей следующим образом.

```

$ source venv/bin/activate
$ pip install pymongo
$ pip freeze > my_flask_app/requirements.txt

```

## Обновление docker-compose.yml

В уже существующий docker-compose.yml необходимо добавить новый сервис **mongodb**, запускающийся из образа mongo версии 3.2, а также обновить сервис **web**, добавив параметр *links*, указывающий на сервис **mongodb**, чтобы контейнер **web** знал о контейнере **mongodb** после запуска.

```

$ cat docker-compose.yml
web:
    build: ./my_flask_app
    ports:
        - "5000:5000"

```

volumes:

- ./my\_flask\_app:/app

links:

- mongodb

mongodb:

image: mongo:3.2

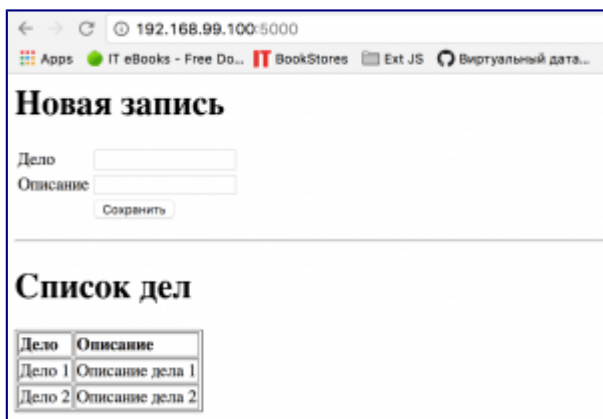
Параметр links — это инструкция для Docker, которая заставит его обновить файл /etc/hosts контейнера web, внося туда информацию об IP-адресе контейнера mongodb.

Сборка и запуск обновленного Flask приложения

Обновите ваше приложение командой

```
$ docker-compose up -d --build
```

После успешного выполнения этой команды в браузере по адресу <http://192.168.99.100:5000/> будет запущено ваше приложение, состоящее уже из двух контейнеров: **web** с простым Flask приложением и **mongodb** с СУБД.



## Заключение

Использование Docker Compose позволяет существенно сократить время, затрачиваемое на разработку, обновление и сопровождение ваших приложений. В следующих статьях я расскажу о том, как масштабировать ваше приложение.

# Использование Swarm secrets в Docker Compose

- [Андрей Максимов](#) 0 комментариев

Как уже упоминалось в статье [Docker Secrets — безопасное хранилище для чувствительных данных](#), у вас есть 2 способа создания «секретов» в Docker Swarm:

- На основе содержимого файла
- Со стандартного потока ввода-вывода

Давайте посмотрим, как использовать эту функциональность вместе с Docker Compose в `docker-compose.yml` файлах. Для иллюстрации примера создайте директорию `docker_secrets_with_compose`, в которой мы продолжим работу:

```
mkdir docker_secrets_with_compose
cd docker_secrets_with_compose/
```

## На основе содержимого файла

Создайте файл `my_db_password_1.txt` с произвольным паролем:

```
LC_ALL=C < /dev/urandom tr -dc '_A-Z-a-z-0-9' | head -c 32 > my_db_password1.txt
```

Далее создайте `docker-compose.yml`, который предназначен для создания сервиса `test` с «секретом» `my_db_password1`:

```
version: '3.1'
services:
  test:
    image: 'alpine'
    command: 'cat /run/secrets/my_db_password1'
    secrets:
      - my_db_password1

secrets:
  my_db_password1:
    file: ./my_db_password.txt
```

Запустите ваш сервис командой:

```
docker stack deploy -c docker-compose.yml my_secret_service
```

Проверьте содержимое лога вашего сервиса:

```
docker service logs my_secret_service_test
```

Пароль должен совпадать с содержанием файла `my_db_password.txt`

## Со стандартного потока ввода-вывода

Для иллюстрации этого подхода создайте отдельный «секрет» `my_db_password2`:

```
LC_ALL=C < /dev/urandom tr -dc '_A-Z-a-z-0-9' | head -c 32 | docker secret create my_db_password2
```

В качестве вывода команды вы получите сгенерированный и запомненный «секрет», например:

```
qvfvffi1nxv260e8p3v9zcnuy
```

Дополните содержимое файла `docker-compose.yml` следующим новым сервисом (`test2`), который будет использовать уже существующий «секрет» (все по аналогии с дисками):

```
version: '3.1'
services:
  test:
    image: 'alpine'
    command: 'cat /run/secrets/my_db_password1'
    secrets:
      - my_db_password1
  test2:
    image: 'alpine'
    command: 'cat /run/secrets/my_db_password2'
    secrets:
      - my_db_password2
```

```
secrets:
  my_db_password1:
    file: ./my_db_password.txt
  my_db_password2:
    external: true
```

И обновите запущенный вами сервис:

```
docker stack deploy -c docker-compose.yml my_secret_service
```

В логе `my_secret_service_test2` будет вывод вашего пароля, созданного из стандартного потока ввода-вывода:

```
docker service logs my_secret_service_test2
```

## Заключение

Использование «секретов» в Docker Swarm позволяет вам не только легко и просто управляться с приватными данными внутри ваших приложений, но и разделить зоны ответственности разработчиков и администраторов ваших приложений. Если у вас появились какие-либо вопросы, с удовольствием отвечу на них в комментариях внизу страницы. Приятной работы!

# Portainer: управление Docker и Swarm при помощи Web-UI

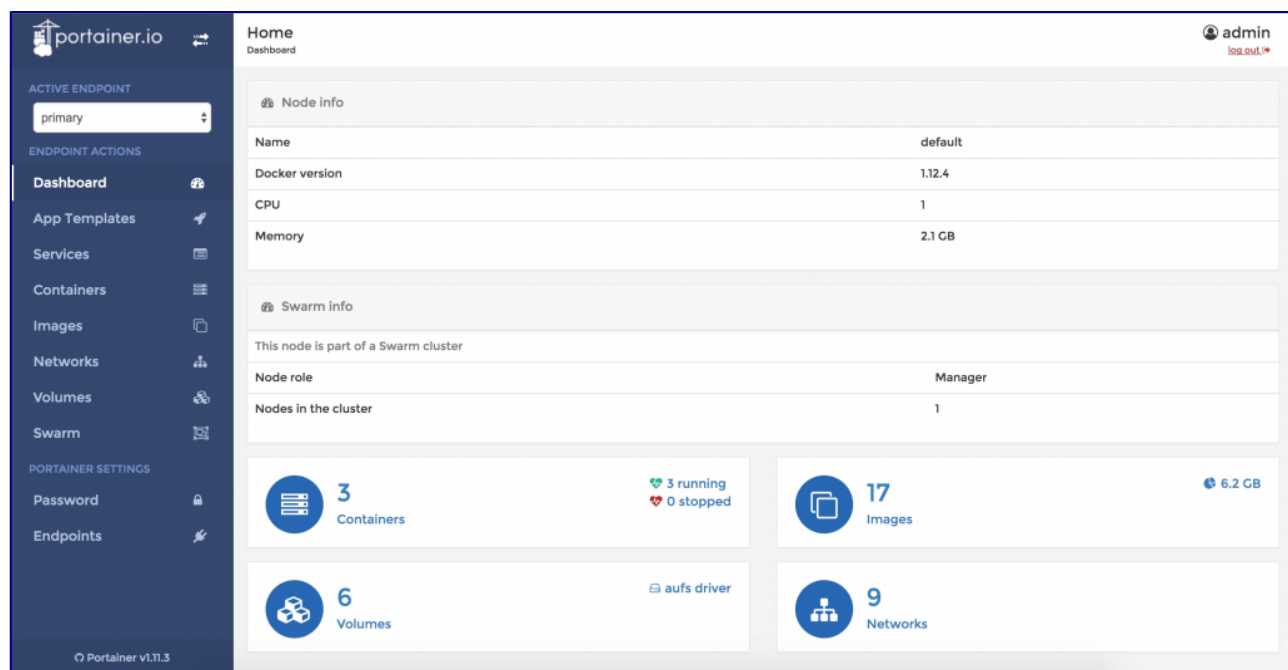
- [Андрей Максимов](#) [2 комментария](#)

Для тех из вас, кому хочется управлять вашими Docker хостами или Swarm кластерами не только из командной строки, но и получать эстетическое удовольствие используя красивый Web-UI интерфейс, рекомендую посмотреть на [Portainer](#). Portainer — это проект с открытым исходным кодом, предоставляющий вам минимальный по размеру (3 Мб) образ Web-интерфейса для управления Docker Engine или Swarm кластером.

## Основная панель

Как видно из экрана, Portainer — это отличное решение, которое позволяет не только очень наглядно предоставлять полную информацию по вашим Docker хостам и кластерам, но и эффективно управлять ими, предоставляя вам множество возможностей, вкратце о которых ниже.

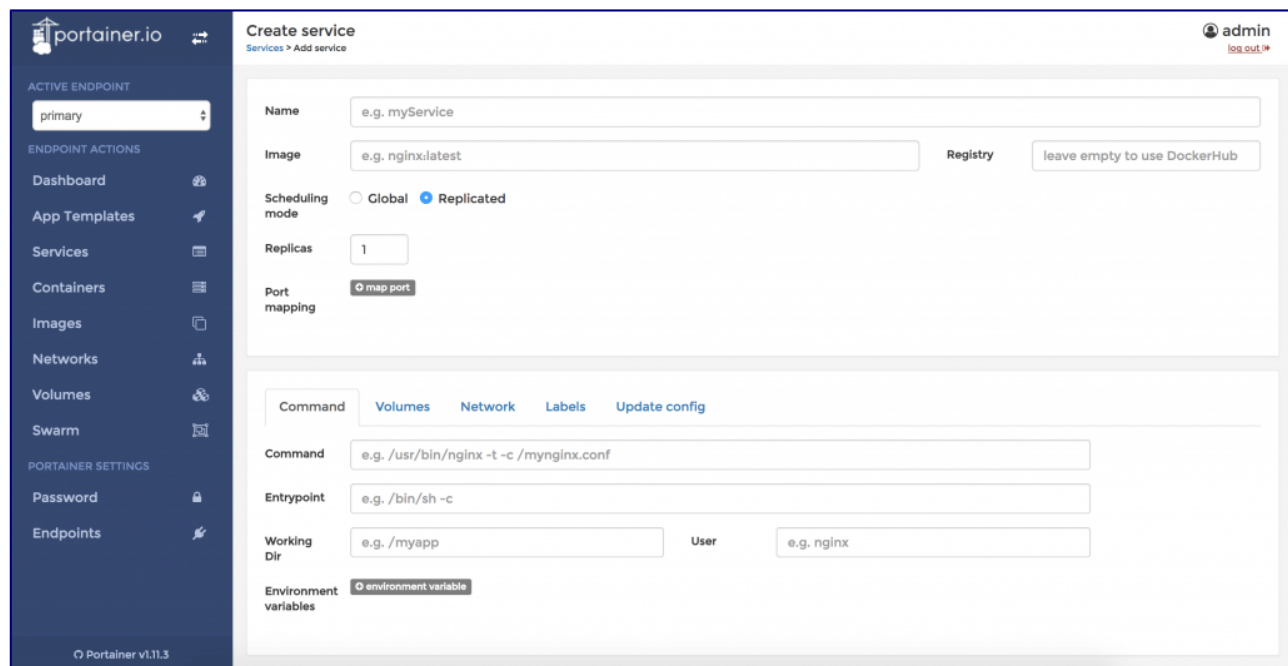




## Сервисы (Services)

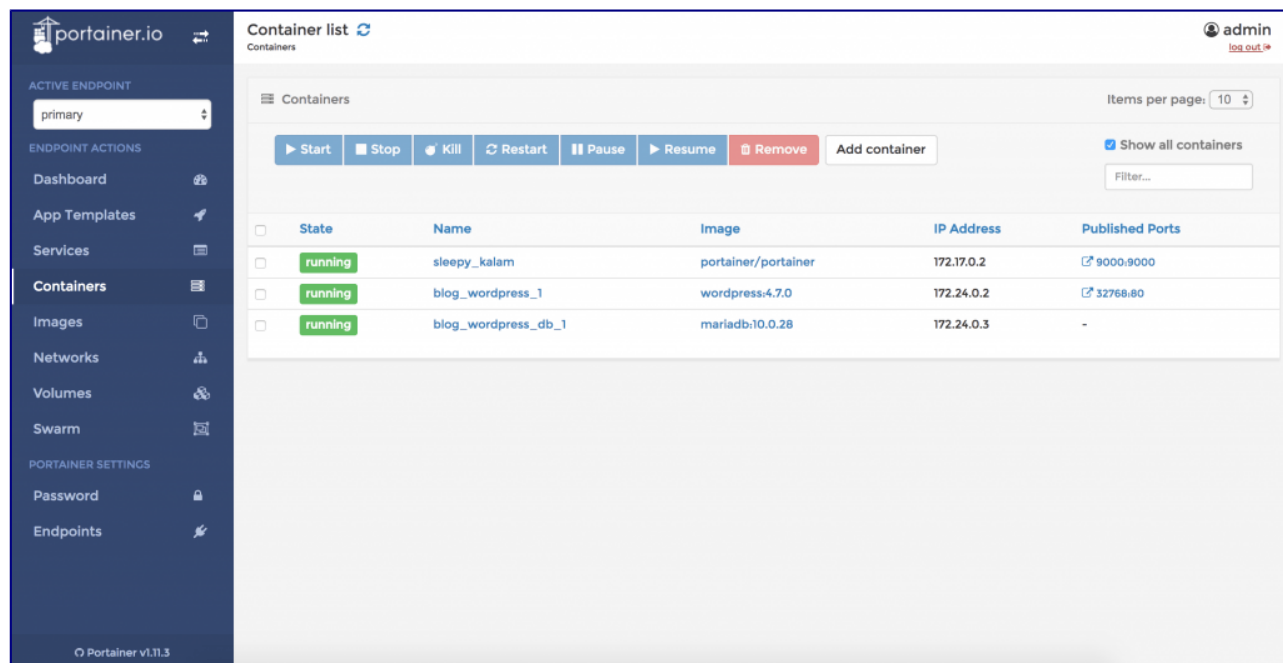
Если ваш Docker Engine запущен в Swarm mode, вам становится доступно меню управления сервисами Swarm кластера. Процесс создания сервиса продемонстрирован на экране ниже. Прямо из Web-UI вы можете задать все значимые параметры сервиса:

- Имя самого сервиса
- Имя образа, из которого будет запущен сервис
- Указать сторонний Docker реестр
- Режим работы планировщика
- Маппинг портов, дисков, сетей
- А также метки



# Контейнеры (containers)

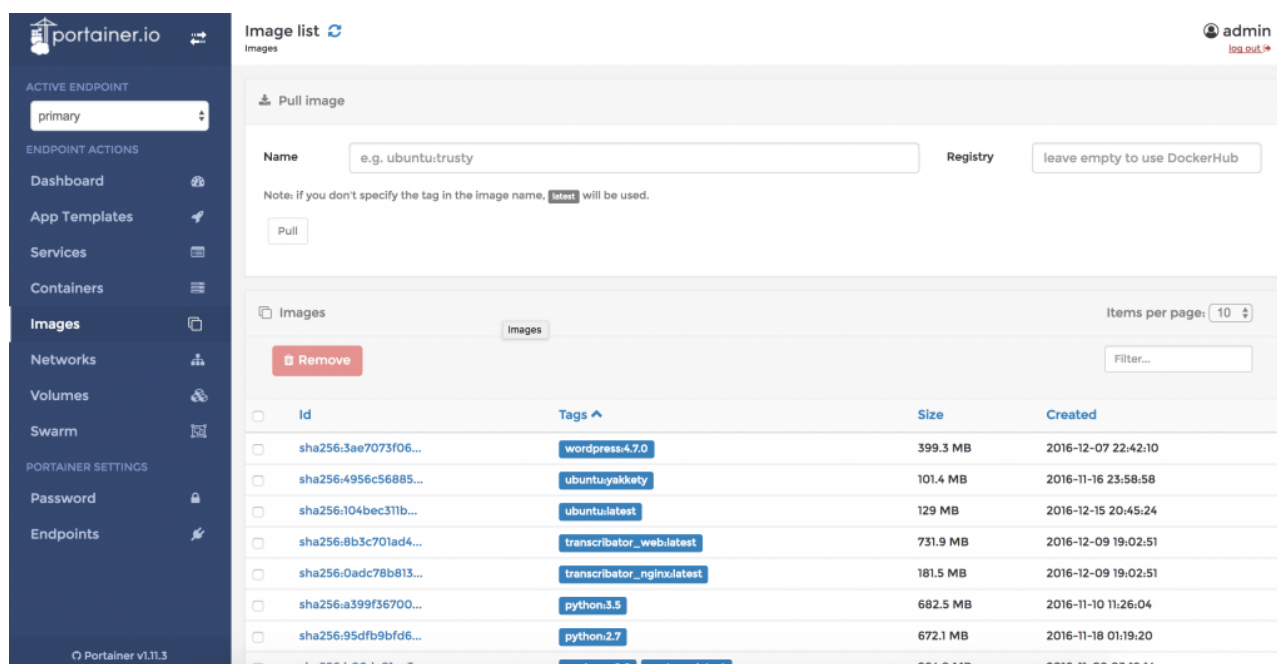
Помимо управления службами Portainer дает возможность управлять жизненным циклом контейнеров, запущенных на вашем хосте или кластере. Интерфейс создания контейнера ничем не отличается от интерфейса создания сервиса.



Если перейти в управление конкретным контейнером, то у вас появится возможность наблюдать за потреблением ресурсов конкретного контейнера, просмотра логов, а также подключению к интерактивной консоли (да, можно зайти внутрь работающего контейнера в терминальную сессию и выполнять нужные вам команды).

# Образы (Images)

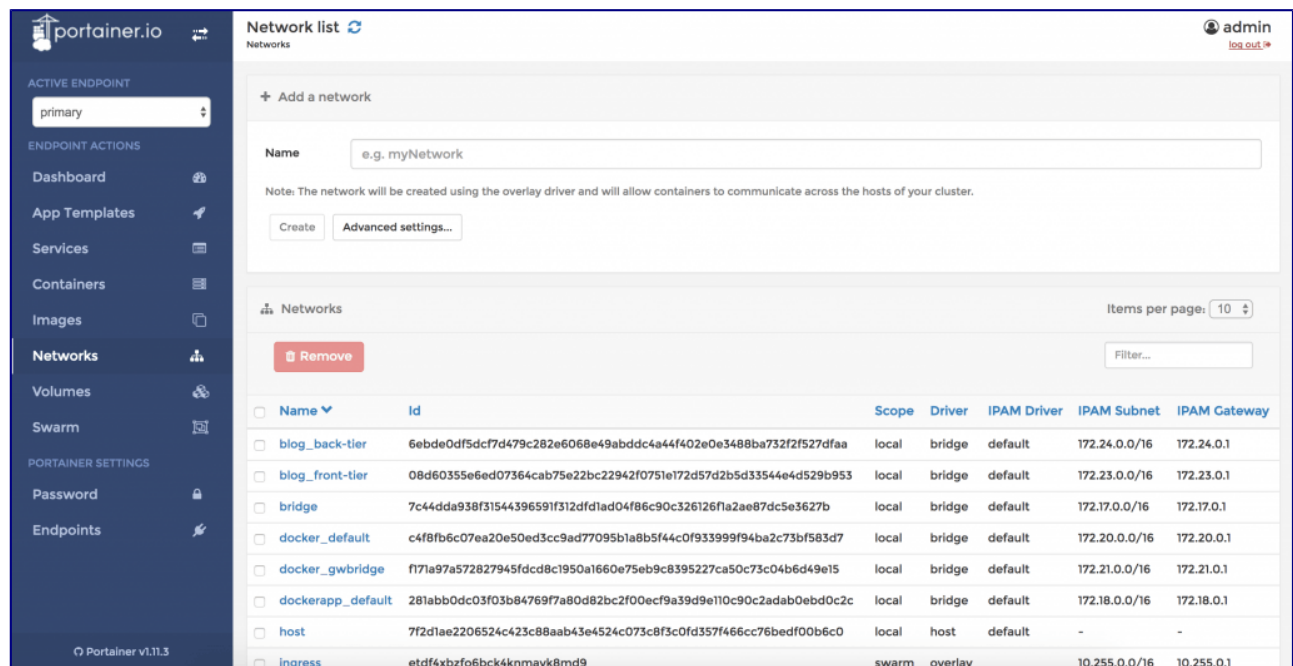
Не знаю как вам, а мне кажется очень удобной возможность выделить мышкой сразу 5-6 не используемых образов, а затем удалить их в один клик.



Более того, при клике на каждый конкретный образ можно изменить его тег, получить информацию о размере, дате создания, а также информацию из Dockerfile, такую как, например: CMD, ENTRYPOINT, EXPOSE, VOLUME и переменные окружения внутри контейнера из ENV.

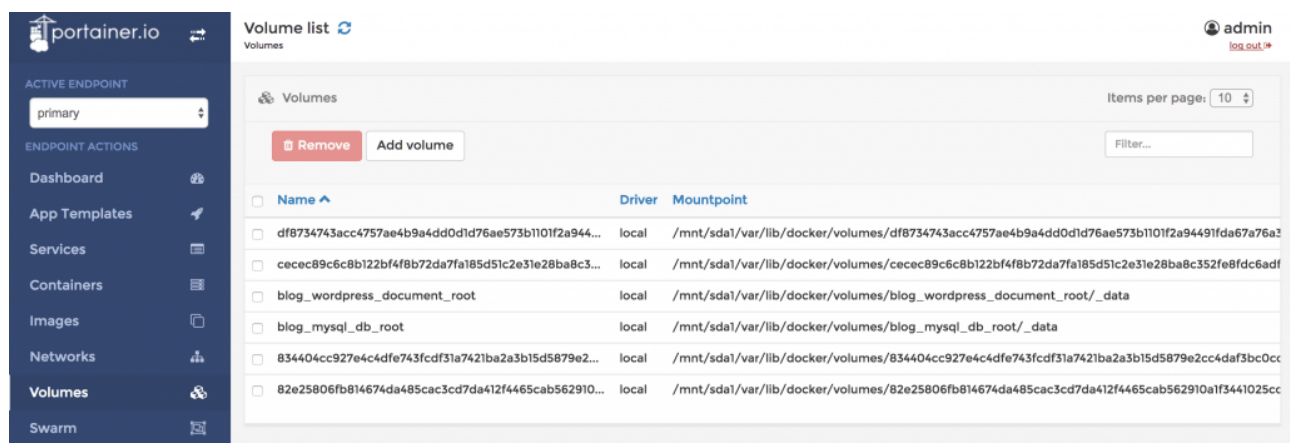
## Сети (Networks)

Возможностей работы с сетями в Portainer пока не очень много. В списке с сетями отображается лишь скудная информация о типе сети и ее адресации.



## Диски (Volumes)

В принципе все, что можно получить из `docker volume inspect` удобно и наглядно отображается в едином интерфейсе, дополняя остальную функциональность.



## Кластеры (Swarm)

Состав кластера, а также количество задействованных узлов и доступных в кластере ресурсов можно узнать в меню Swarm. При выборе каждого конкретного узла кластера можно очень удобно очистить его от виртуальных машин и остановить на нем аллокацию.

portainer.io Cluster overview Swarm admin log out

ACTIVE ENDPOINT: primary

ENDPOINT ACTIONS: Dashboard, App Templates, Services, Containers, Images, Networks, Volumes, Swarm

Cluster status

Nodes	1
Docker API version	1.24
Total CPU	1
Total memory	2.1 GB

Node status

Name	Role	CPU	Memory	Engine	Status
default	manager	1	2.1 GB	1.12.4	ready

## Подключения (Endpoints)

Portainer очень удобно использовать вместе со своей Docker Machine, чтобы иметь возможность управлять сразу всеми вашими Docker хостами и кластерами, определить очередное подключение не составит труда в соответствующем меню. Опция поддержки TLS также имеется.

portainer.io Endpoints Endpoint management admin log out

ACTIVE ENDPOINT: primary

ENDPOINT ACTIONS: Dashboard, App Templates, Services, Containers, Images, Networks, Volumes, Swarm

PORTAINER SETTINGS: Password, Endpoints

Endpoints

+ Add a new endpoint

Name: e.g. docker-prod01

Endpoint URL: e.g. 10.0.0.10:2375 or mydocker.mydomain.com:2375

TLS: ☐

Add endpoint

Endpoints

Name	URL	TLS
★ primary	192.168.99.100:2376	<input type="checkbox"/>

You cannot edit the active endpoint

## Запуск

В принципе, у Portainer очень достойная [документация](#), но зачем же в нее ходить, если можно получить



команду для запуска прямо не покидая страницу обзора?

```
$ docker run -d -p 9000:9000 -v ~/.docker/machine/certs:/certs portainer/portainer -H tcp://192.168.99.100:2376 --tlsverify
```

Данная команда запустит контейнер, содержащий Portainer на 9000 порту вашего Docker Engine. Я запускаю Portainer на своем ноутбуке. Флаг -v смонтирует сертификаты из моей домашней директории внутрь контейнера, чтобы предоставить мне возможность управлять любым Docker хостом или Swarm кластером, настроенным при помощи моей Docker Machine с использованием TLS аутентификации (очень удобно, когда вы администрируете множество хостов). Флаг -H указывает на IP адрес хоста,

которым Portainer будет управлять — это нужно исключительно для формирования **primary** подключения (см последний скриншот).

Приятной работы!

..

# Docker Secrets — безопасное хранилище для чувствительных данных

- [Андрей Максимов](#) 0 комментариев

Важнейшим элементом для построения безопасных приложений является наличие безопасного способа общения ваших контейнеров друг с другом, а также с другими приложениями и системами. А это в свою очередь первоначально требует безопасного способа управления учетными записями, токенами, паролями, ключами, а также другими видами чувствительной информации, как правило называемой в приложении просто «секретами». Начиная с версии 1.13 в Docker начал поддерживаться безопасный способ работы с такими чувствительными данными — Docker secrets. Этот безопасный способ работы реализуется в Docker кластере, работающем в так называемом режиме Swarm mode.

В данной статье будет рассмотрена работа чувствительными данными в Docker кластере такими как ключи, пароли и т.д.

Предполагается, что у вас уже есть Docker кластер, работающий в Swarm mode. А если еще нет, вы легко можете установить его, воспользовавшись инструкциями из статьи [«Установка отказоустойчивого Docker Swarm кластера в Облаке КРОК»](#), или же вы можете [обновить свой Docker Engine](#) до версии 1.13 и перевесть его в этот режим работы, чтобы посмотреть на работу технологии и не устанавливать отдельный кластер:

```
$ docker swarm init
```

## План

Мы запустим в контейнере простое Flask приложение (похожее на то, что описывалось в статье [«Упаковка простого Flask приложения в Docker контейнер»](#)), которое будет считывать переданный ему пароль и выводить его в ответ на http запросы. Если же «секрета» с паролем не будет, приложение будет выдаваться сообщение об ошибке.

Вы можете найти образ готового контейнера на [Docker Hub](#), а код приложения [GitHub](#).

## Создание приложения

Прежде чем переходить к описанию работы с «секретами», давайте создадим демонстрационное приложение. Для этого создайте директорию `docker_secrets_example`:

```
$ mkdir docker_secrets_example
```

```
$ cd docker_secrets_example
```

Создайте в нем файл `requirements.txt`, содержащий следующую информацию:

```
click==6.7
Flask==0.12
itsdangerous==0.24
```

```
Jinja2==2.9.5
MarkupSafe==0.23
Werkzeug==0.11.15
wheel==0.24.0
```

Создайте и активируйте виртуальное окружение, а затем установите зависимости

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

Создайте файл `app.py`, который будет содержать код нашего приложения

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from flask import Flask
from flask import render_template

app = Flask(__name__)

@app.route('/')
def index():
    try:
        db_password = None
        f = open('/run/secrets/db-password', 'r')
        db_password = f.readline()
    except IOError:
        pass
    finally:
        return render_template('index.html', password=db_password)

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

А также необходимо создать директорию `templates` и поместить внутрь файл `index.html` со следующим содержимым:

```
<!doctype html>
<title>Docker secrets demo</title>
{% if password %}
    <p>Your secret password: <strong>{{ password }}</strong></p>
{% else %}
    <strong>No password specified!</strong>
{% endif %}
```

Внутри рабочей директории создайте файл `Dockerfile` со следующим содержимым, чтобы иметь возможность собирать ваш Docker образ:

```
FROM python:2.7
MAINTAINER Andrey Maksimov 'maksimov.andrei@gmail.com'
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

Структура вашего приложения на текущий момент без учета виртуального окружения (`venv`) будет следующей:

```
$ tree -I venv docker_secrets_example
docker_secrets_example
```

```
├─ Dockerfile
├─ app.py
├─ requirements.txt
└─ templates
    └─ index.html
```

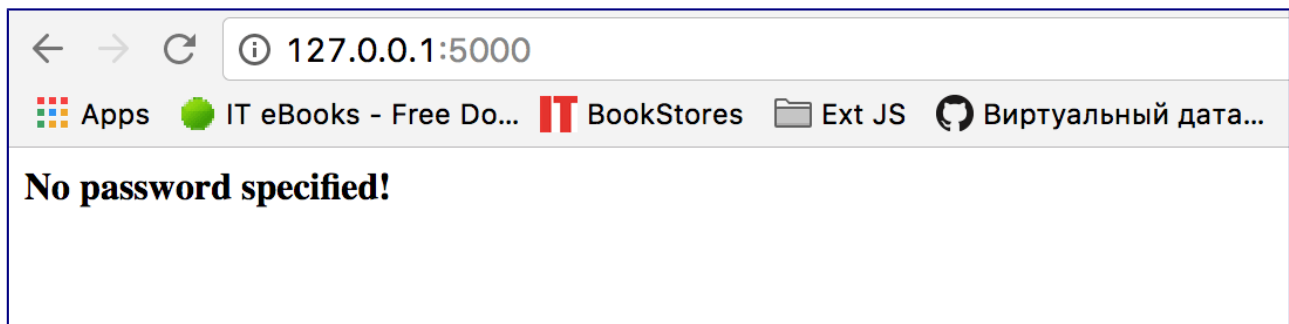
1 directory, 4 files

## Проверка работоспособности приложения

Если на текущий момент вы все сделали правильно, вы можете запустить демонстрационное приложение и проверить его работоспособность, открыв в браузере адрес :

```
$ python app.py
* Running on (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 237-318-490
127.0.0.1 - - [13/Feb/2017 19:02:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Feb/2017 19:02:47] "GET /favicon.ico HTTP/1.1" 404 -
```

При этом в браузере будет отображаться:



## Сборка образа

После того, как приложение заработало, его необходимо упаковать в контейнер. Для этого необходимо выполнить команду:

```
$ docker build -t amaksimov/docker_secrets_example:v0.1 .
```

Для корректной работы сервисов в Docker Swarm кластере, опубликуйте ваш собранный образ в вашем репозитории:

```
$ docker push amaksimov/docker_secrets_example:v0.1
```

**Важно:** если этого не сделать, в процессе запуска сервиса команда `docker service create` будет выдавать сложно интерпретируемую ошибку.

## Работа с «секретами» (Docker secrets)

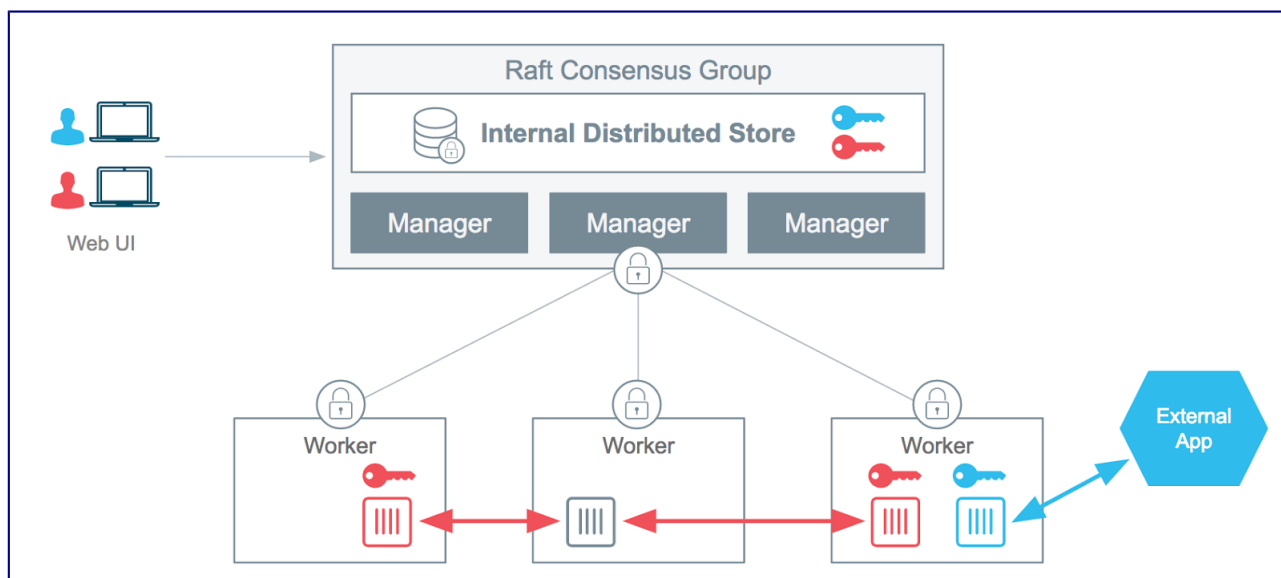
После того, как демонстрационный образ создан, можно переходить непосредственно к работе с «секретами».



## Как работают секреты

«Секреты» используются Docker сервисами посредством установления явной связи между сервисом и самими секретами. Сервисы, напомним, состоят из задач (отдельных контейнеров), которые могут выполняться на любом узле в пределах вашего кластера. Если ваш сервис состоит из нескольких задач (контейнеров), связанный с ним «секрет» будет доступен для всех задач этого сервиса.

Docker Swarm кластер использует [Raft Consensus Algorithm](#) для того, чтобы гарантировать, что узлы, участвующие в управлении кластером, договариваются о состоянии кластера. Часть этого процесса включает в себя репликацию состояния всех управляющих узлов кластера при помощи журнала.



Реализация «секретов» в кластере, работающем в Swarm mode режиме, использует алгоритм Raft. «Секреты» в процессе создания записываются в raft-журнал, что позволяет им реплицироваться на каждый из управляющих узлов кластера. До тех пор, пока кластер работает, журнал Raft хранится в памяти каждого управляющего узла и реплицируется в зашифрованном виде.

В процессе запуска нового сервиса в кластере, если в процессе создания ему предоставлена возможность доступа к тому или иному «секрету», происходит монтирование текстового содержимого «секретов» в директорию `/run/secrets` всем задачам (контейнерам) этого сервиса.

Команда `docker secret` позволяет нам задать «секреты», которые могут быть использованы сервисами, запущенными в Docker Swarm кластере. Есть два способа создания «секрета».

### Создание «секрета» из файла

Создайте в рабочей директории файл `password.txt` который будет содержать секретную информацию, например, пароль к базе данных `this-is-my-super-secret-password`. Чтобы дать возможность вашим приложениям использовать этот секретный пароль, выполните команду:

```
$ docker secret create db-password password.txt
```

Эта команда создаст «секрет» с именем `db-password`, используя содержимое файла `password.txt`

### Создание «секрета» со стандартных потоков ввода\вывода

Также вы можете создавать «секреты», используя информацию со стандартного потока ввода\вывода:

```
$ LC_ALL=C &lt;< /dev/urandom tr -dc '_A-Z-a-z-0-9' | head -c 32 |  
docker secret create db-password -
```

Эта команда также создаст «секрет» `db-password`, но уже используя сгенерированную информацию из `/dev/urandom`.

## Просмотр «секретов» кластера

Все созданные в кластере секреты могут быть просмотрены при помощи команды:

```
$ docker secret ls
```

ID	NAME	CREATED	
UPDATED			
ioqghuhj7ix9a57joiarivv8n	db-password	3 hours ago	3
hours ago			

## Удаление «секретов» из кластера

Удаление «секретов» также осуществляется достаточно просто:

```
$ docker secret rm db-password
```

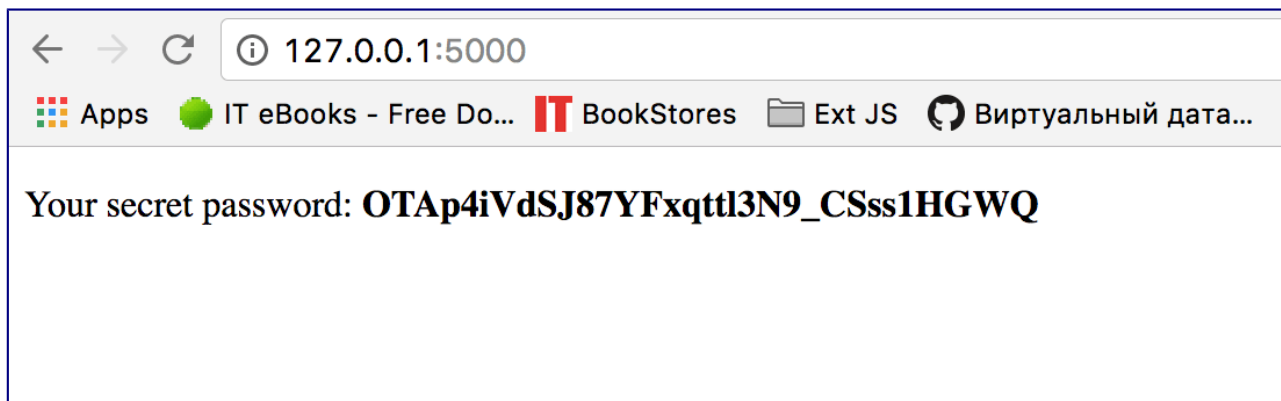
## Управление сервисом с «секретом»

### Запуск

Для того, чтобы запустить ваш сервис и передать ему информацию о «секрете», необходимо выполнить команду:

```
$ docker service create --name docker_secrets_example -p 5000:5000  
--secret db-password amaksimov/docker_secrets_example:v0.1
```

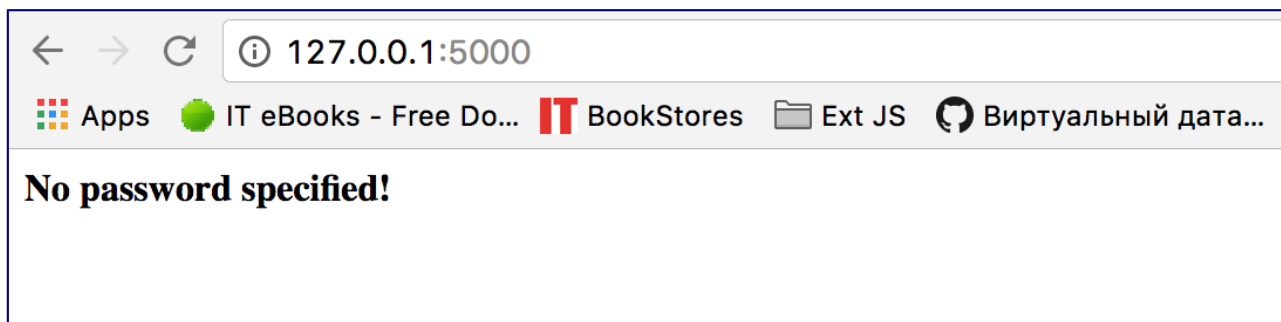
После чего можно будет убедиться в том, что ваше приложение прочитало файл с секретом:



Если же запустить ваш сервис без указания ключа `—secret`, например, командой:

```
$ docker service create --name docker_secrets_example -p 5000:5000  
amaksimov/docker_secrets_example:v0.1
```

То, как и ожидается, ваше приложение об этом сразу же сообщит:



## Добавление «секрета» к уже работающему сервису

К сожалению, для того, чтобы добавить секрет к сервису, последний придется пересоздать, предварительно выполнив команду:

```
$ docker service rm docker_secrets_example
```

## Обновление «секретов» для сервиса

Также к сожалению, для того, чтобы обновить секрет у сервиса, его придется пересоздать.

## Заключение

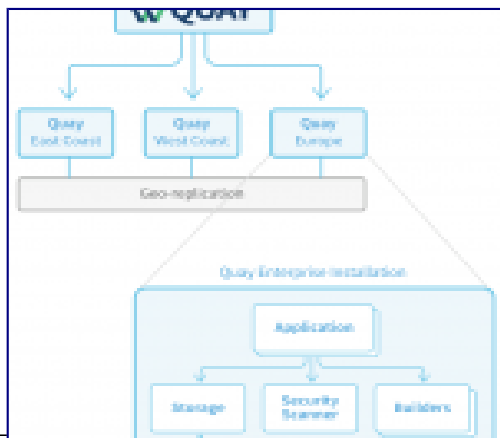
В этой статье мы рассмотрели еще одну возможность Docker, предназначенную для безопасной работы с чувствительными данными ваших приложений. Если у вас возникли какие-либо вопросы, буду рад ответить на них в комментариях в конце статьи.

# Docker: решения для хранения образов

- [Андрей Максимов](#) 0 комментариев

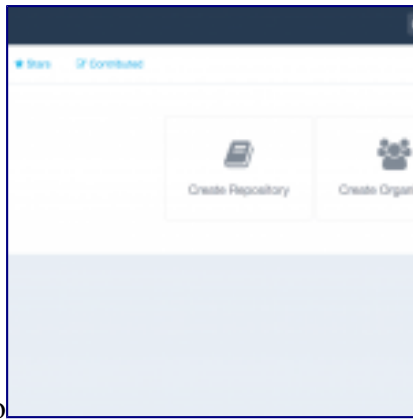
На сегодняшний день ни одно решение, построенное на основе или просто использующее Docker, не обходится без хранилища Docker образов (images). Такие хранилища делятся на публично доступные и приватные.

## Публично доступные хранилища



[Quay.io](#) — это хранилище образов Docker от [CoreOS](#), который продвигает себя как имеющие «продвинутые триггеры для сборки», «расширенные возможности по управлению привилегиями для команды» и «безопасное хранилище». В настоящий момент из публично доступных решений Quay является наиболее дружелюбным для использования Enterprise компаниями. Quay поддерживает любой сервер git, а также позволяет легко создавать сложные рабочие процессы, например, отображение названий веток git в теги Docker. Таким образом, когда вы делаете коммит вашего кода, правильно протегированный Docker образ будет создан автоматически. Quay предоставляется как публичное решение или же как частная инсталляция. Публичное решение предлагает неограниченное количество бесплатных публичных репозиторий, за приватные репозитории вашей организации придется заплатить.

[Docker Hub](#) является собственным предложением компании Docker. Как и у Quay он предлагает неограниченное количество бесплатных публичных репозиторий и не взимает плату за хранение или

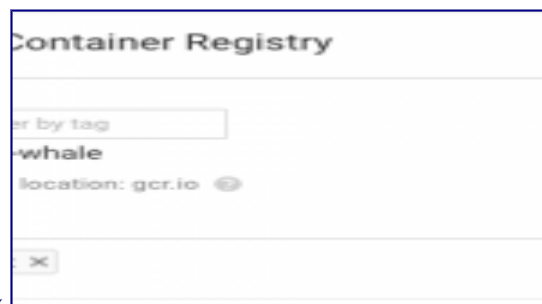


пропускную способность. Если вам нужны частные репозитории, они оплачиваются на основе нужного вам количества и количества параллельных сборок. Docker Hub не предлагает возможности гранулярного управления полномочиями, как это сделано у Quay, что может быть проблемой для большого количества Enterprise компаний. Docker Hub также имеет более ограниченную интеграцию git, требуя, чтобы ваш код жил на GitHub или Bitbucket, если вы хотите воспользоваться автоматизированной сборкой. Стоит отметить, что Docker Hub — это основной реестр используемый вами, когда вы выполняете команду `docker push` или `docker pull`.



**[AWS EC2 Container Registry \(ECR\)](#)** — это относительно новый сервис от Amazon AWS, использовать который имеет смысл только в том случае, если вы уже используете или собираетесь использовать какие-либо другие сервисы от Amazon AWS, например, [Elastic Container Service](#). ECR может быть использован как любое другое хранилище образов Docker, но максимальную пользу из него можно извлечь только используя его совместно с другими продуктами Amazon AWS. Как и большинство других сервисов от AWS, ECR также позволяет очень гибко регулировать уровень доступа и привилегии.

Одна из наиболее важных особенностей ECR в том, что его ценовая политика сильно отличается от Quay и Docker Hub. Вместо того, чтобы брать оплату за количество частных репозиторий, это решение оплачивается строго по объему занятого пространства и уровня утилизации сети.



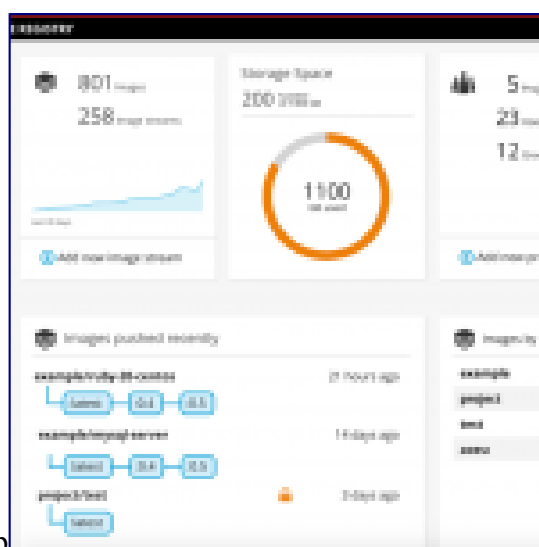
**[Google Container Registry](#)** — этот репозиторий очень похож на Amazon ECR, только он тесно интегрирован в инфраструктуру облачных сервисов Google. Точно так же его имеет смысл использовать только если часть вашей инфраструктуры будет использовать облачные услуги Google, например, Kubernetes.

Ценообразование также похоже на ECR, т.е. вы платите только за объемы хранимых Docker образов и

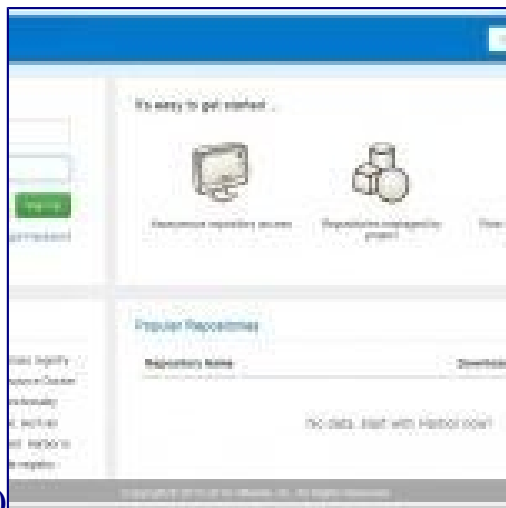
исходящий сетевой трафик. У данного решения как и у всего облака Google приоритет прежде всего на безопасность, поэтому все пока что не очень удобно, зато очень и очень безопасно. Если вам нужна возможность автоматизировать ваши процессы разработки, тестирования и релизов, скорее всего вам пока не сюда.

## Решения для организации частных частных репозиториев

**Docker Registry 2.0** — Открытая версия репозитория, который использует Docker для организации своего **Docker Hub**. Документация на него более, чем исчерпывающая. Возможно все, но это «все» надо настраивать: отказоустойчивость, аутентификацию из правильного источника и само собой управление пользователями. Никакого Web-интерфейса из коробки. Выбирать это решение стоит только если вы готовы потратить какое-то значительное количество времени на «допиливание» этого решения. В противном случае, у Docker можно купить цельное решение для построения контейнерной инфраструктуры **Docker Datacenter**.

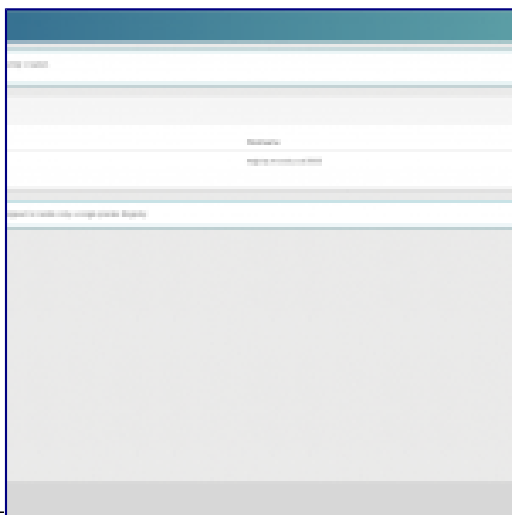


**Atomic Registry (RedHat)** — Открытая версия репозитория от RedHat, выполненная на базе одного из проектов OpenShift. Распространяется в виде образов виртуальных серверов, которые можно скачать и запустить в любом Облаке, среде виртуализации или физическом железе. Разработчики хвалят это решение за то, что оно изначально спроектировано для кластеризации и отказоустойчивой работы, красивый Web-интерфейс, а также поддержку SSO и RBAC, что позволяет встроить этот репозиторий в собственную инфраструктуру. Документация достаточная, но далеко не исчерпывающая. Запустив VM из образа с этим решением, очень легко понять насколько оно будет вам полезно. На основной странице проекта есть 2 видео, показывающие как поставить и как начать использовать это решение.



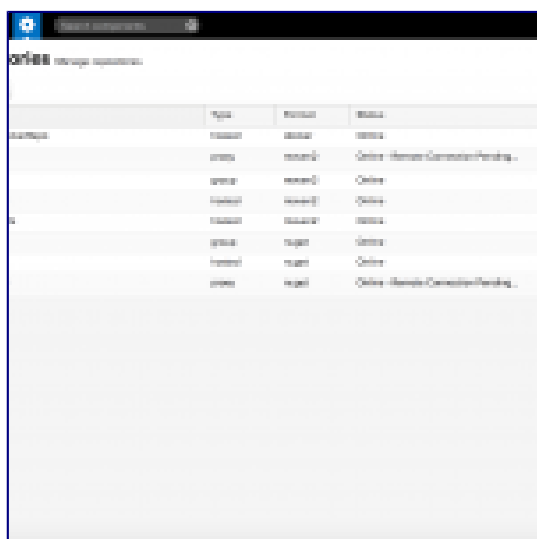
### **Harbor (VMware)**

— еще один открытый репозиторий, сделавший ставку на Enterprise компании, реализовав RBAC, репликацию образов между узлами кластера для лучшей балансировки нагрузки, поддержку AD\LDAP, аудитный журнал операций, красивый Web-интерфейс, переведенный на несколько языков, включая русский, REST API и удобный механизм развертывания (виртуальная машина для vSphere в формате OVA).



### **Portus (Suse)**

Открытая версия репозитория от Suse. Чуть менее выдающийся Web-интерфейс, чем у предшественников, все те же реализованные RBAC, интеграция с LDAP, аудит операций, тегирование репозитория, возможность включить пользователям возможность самостоятельной регистрации с последующей модерацией. Это решение очень простое и понятное.



вляю в этот список уже после того, как статья была опубликована. Это решение позволяет создавать не только репозитории для хранения образов Docker, но в добавок поддерживает и позволяет вам создавать репозитории bower, maven2, npm, nuget, pip и gubygems. Помимо всего прочего это решение может быть интегрировано с LDAP, позволяет очень гибко разграничивать права доступа пользователей к различным репозиториям, а всю настройку можно легко выполнить в удобном Web-интерфе. Решение абсолютно бесплатное.

**[Quay.io Enterprise](#)** — Частная инсталляция Quay для установки на площадке и оборудовании конечного заказчика характеризуется возможностью настройки отказоустойчивого режима работы всех компонентов, входящих в состав решения, возможностью организации гео-репликации образов Docker между датацентрами, возможностью строить процесс непрерывной интеграции, возможностью подключать сканер безопасности образов, например, [Clair](#), упоминавшийся в статье [Docker: проверка контейнеров на безопасность](#), а также возможность использования LDAP в качестве сторонней базы учетных записей пользователей. Полный список возможностей можно посмотреть [тут](#).

**[Artifactory](#)** — Наверное, самое странное и дорогое решение для хранения образов Docker. Не смотря на то, что это решение можно использовать как SaaS, большинство известных мне компаний успешно используют этот продукт все же в виде отдельной частной инсталляции. Вообще говоря, изначальное назначение этого программного продукта было далеко не хранение образов Docker, однако, с какого-то момента «теперь ваш бинарный репозиторий для хранения артефактов разработки еще и Docker образы хранить умеет».

## По материалам

- [An Overview of Docker Registries](#)
- [Comparing Four Hosted Docker Registries](#)
- [Awesome Docker GitHub repository](#)
- [Deploying a private, secured Docker Registry within 15 minutes](#)

# Установка приватного Docker репозитория

- [Андрей Максимов](#) 0 комментариев

В этой статье будет рассмотрено как быстро и легко установить собственный приватный Docker репозиторий. Приватный Docker репозиторий может понадобиться вам скорее всего только в одном случае: ограничение доступа к Docker образам периметром вашей организации. В остальных случаях можно пользоваться любыми публично доступными репозиториями.

В качестве решения для организации приватного репозитория мы будем использовать [Sonatype Nexus Repository Manager 3.0 OSS](#). А запускать необходимый контейнер из его образа мы будем при помощи Docker Compose ([кратко о Docker Compose](#)).

Для того, чтобы это сделать, создайте в месте, где вы обычно работаете над собственными проектами директорию, например, `nexus3`, а внутри нее директорию `data` и файл `docker-compose.yml`:

```
$ mkdir -p nexus3/data
$ touch nexus3/docker-compose.yml
$ tree nexus3/
nexus3/
├── data
└── docker-compose.yml
```

1 directory, 1 file

Содержимое `docker-compose.yml` будет следующим:

```
nexus:
  image: sonatype/nexus3:latest
  ports:
    - "8081:8081"
    - "8123:8123"
  volumes:
    - ./data:/nexus-data
```

В этой конфигурации мы пробрасываем 2 порта до контейнера:

- **8081** — для Web-UI интерфейса
- **8123** — API интерфейс для Docker реестра

Также мы монтируем директорию `data` текущей директории `nexus3` внутрь контейнера (в директорию `/nexus-data`) для сохранения состояния контейнера при его перезапусках. При использовании в продуктивной инсталляции вы можете использовать Docker диск (volume), расположенный на NAS или распределенной файловой системе, т.к. GlusterFS, например.

## Запуск Docker реестра (Nexus)

Для запуска реестра, как обычно, необходимо выполнить команду `docker-compose up` с ключем `-d` для того, чтобы терминал вернул вам управление:

```
$ docker-compose up -d
```

Первый запуск Nexus займет какое-то время. Понять, что реестр корректно запустился можно будет по выводу команды `docker-compose logs`:

```
$ docker-compose logs
```

Как только контейнер успешно запустится, вывод команды будет оканчиваться следующими строчками:

```
nexus_1 | -----
nexus_1 |
```



```
nexus_1 | Started Sonatype Nexus OSS 3.2.0-01
nexus_1 |
nexus_1 | -----
```

## Создание Docker репозитория

Как только контейнер с Nexus запустится, вы можете подключиться к Docker Engine на порт 8081. В моем случае (OS X, Docker for Mac) — это адрес .

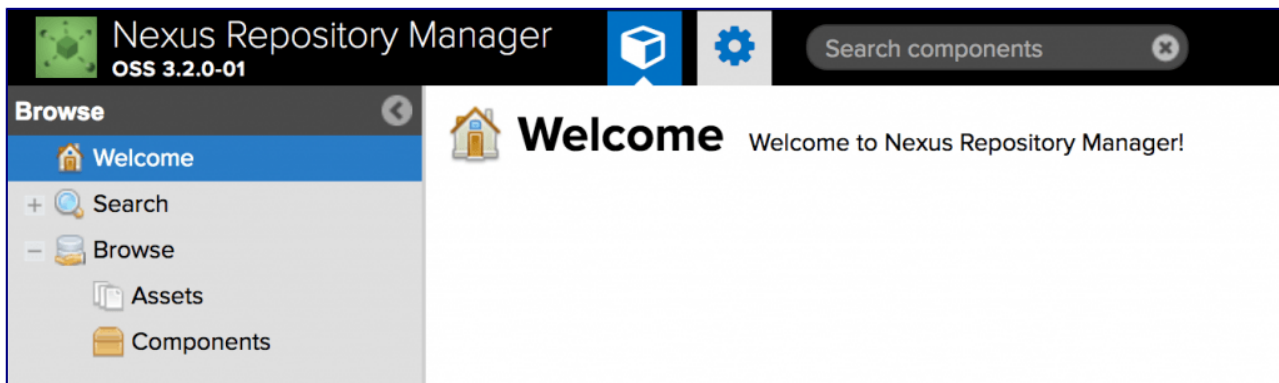
Для того, чтобы настроить репозиторий, необходимо зайти под учетной записью администратора (меню «**Sign In**» в верхнем правом углу экрана):



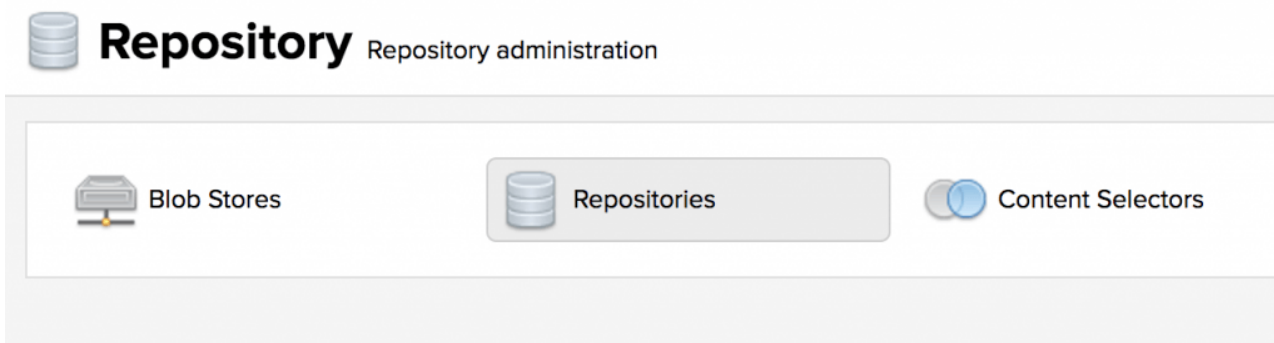
Учетная запись администратора:

- Логин: **admin**
- Пароль: **admin123**










После прохождения процедуры аутентификации, зайдите в настройки:











Выберите «**Repositories**»






Нажмите на кнопку «**Create repository**»:

<div>  <h1>Repositories</h1> <span>Manage repositories</span> </div>			
<div>  Create repository         </div>			
	Name ↑	Type	Format
	maven-central	proxy	maven2
	maven-public	group	maven2
	maven-releases	hosted	maven2
	maven-snapshots	hosted	maven2
	nuget-group	group	nuget
	nuget-hosted	hosted	nuget
	nuget.org-proxy	proxy	nuget

В списке доступных репозиториев выберите «**docker (hosted)**»:

<div>  <h1>Repositories</h1> </div> <div>  Select Recipe         </div>	
	Recipe ↑
	bower (group)
	bower (hosted)
	bower (proxy)
	docker (group)
	<b>docker (hosted)</b>
	docker (proxy)

Укажите имя репозитория (Name), выставьте порт **8123** (второй экспортированный порт в `docker-compose.yml`), а также поставьте галочку «*Allow clients to use the V1 API to interact with this Repository*», если вам нужна поддержка протокола репозитория V1:

 **Repositories** /  **Select Recipe** /  **Create Repository: docker (hosted)**

**Name:**

A unique identifier for this repository

MyPrivateDockerRepo

**Online:**

☒ If checked, the repository accepts incoming requests

**Repository Connectors**

Connectors allow Docker clients to connect directly to hosted registries, but are not always required. Consult our [documentation](#) for which connector is appropriate for your use case.

**HTTP:**

Create an HTTP connector at specified port. Normally used if the server is behind a secure proxy.

☒ 8123

**HTTPS:**

Create an HTTPS connector at specified port. Normally used if the server is configured for https.

☐

**Docker Registry API Support**

**Enable Docker V1 API:**

☒ Allow clients to use the V1 API to interact with this Repository.

Нажмите на кнопку «**Create repository**», чтобы сохранить изменения.

Поздравляю, вы только что установили ваш приватный репозиторий на Nexus!

## Работа с приватным репозиторием

### Подключение к приватному репозиторию

Для того, чтобы подключиться к вашему только что созданному репозиторию, выполните команду:

```
$ docker login -u admin -p admin123 127.0.0.1:8123
Login Succeeded
```

Напомню, что 127.0.0.1 — адрес моего Docker Engine, на котором запущен мой контейнер.

### Сохранение Docker образа в приватный репозиторий

Сохранение образа будет демонстрироваться на базе образа **nginx:latest**. Если такого образа в вашем репозитории нет, выполните команду:

```
$ docker pull nginx:latest
```

Для того, чтобы сохранить ваш образ в приватном репозитории, необходимо так же как и в публичном создать для него тег. Для этого выполните следующую команду:

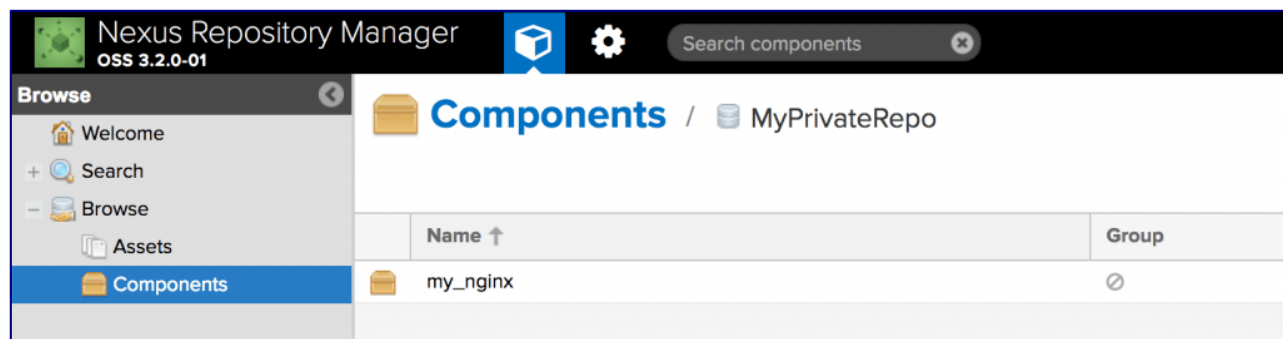
```
$ docker tag nginx:latest 127.0.0.1:8123/my_nginx:latest
```

После этого можно смело сохранять образ:

```
$ docker push 127.0.0.1:8123/my_nginx:latest
The push refers to a repository [127.0.0.1:8123/my_nginx]
7d530616ebc2: Pushed
```

```
db07381cb585: Pushed
a2ae92ffcd29: Pushed
latest: digest:
sha256:a5a0cadd0a8a9d2f42c0cd03a55cd0c610f6cc1466a6f9f9f96b4ae6513e9907
size: 948
```

Также сохраненный образ можно увидеть в Web-интерфейсе репозитория, перейдя в меню «Components» и выбрав ваш созданный репозиторий:



## Загрузка Docker образа из приватного репозитория

Для того, чтобы скачать необходимый образ из приватного репозитория необходимо пройти на нем аутентификацию и выполнить команду `docker pull`:

```
$ docker login -u admin -p admin123 127.0.0.1:8123
$ docker pull 127.0.0.1:8123/my_nginx:latest
```

На сегодня все. Успехов с Docker!

# Обновление Docker

- [Андрей Максимов](#) 0 комментариев

Одна из самых частых и важных операций в IT — это своевременное обновление программного обеспечения. Обновление Docker не исключение.

Узнать актуальную версию установленного на физическом или виртуальном сервере Docker можно при помощи команды

```
$ docker-machine ls
```

```
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
builder.avmaksimov.ru - generic Running tcp://81.23.10.76:2376 v1.12.3
default - virtualbox Running tcp://192.168.99.100:2376 v1.12.1
manager-1.avmaksimov.ru - generic Running tcp://185.12.28.58:2376 v1.12.3
manager-2.avmaksimov.ru - generic Running tcp://185.12.28.7:2376 v1.12.3
manager-3.avmaksimov.ru - generic Running tcp://185.12.28.39:2376 v1.12.3
node-1.avmaksimov.ru - generic Running tcp://81.23.10.217:2376 v1.12.3
node-2.avmaksimov.ru - generic Running tcp://185.12.28.80:2376 v1.12.3
node-3.avmaksimov.ru - generic Running tcp://185.12.29.107:2376 v1.12.3
vpn.avmaksimov.ru * digitalocean Running tcp://45.55.235.195:2376 v1.12.4
```

Как видно из вывода команды самая новая версия Docker установлена на моей виртуальной машине, находящейся в [Облаке DigitalOcean](#) (v1.12.4). В [Облаке КРОК](#) у меня развернут Swarm кластер на версии Docker v1.12.3. И самая старая версия Docker установлена на моем ноутбуке в VirtualBox-e (v1.12.1). Именно ее мы сейчас и обновим.

**Важно!** Помните о том, что при обновлении Docker происходит перезагрузка сервиса Docker управляющего всеми запущенными на вашем хосте контейнерами. При этом все контейнеры на обновляемом хосте будут остановлены и перезапущены автоматически только в том случае, если у них настроена политика автоматической перезагрузки. Проверить политику автоматической перезагрузки можно при помощи команды

```
$ docker inspect -f {{ .HostConfig.RestartPolicy.Name }} 'ID вашего контейнера';
```

Установить политику перезагрузки контейнера можно при помощи команды

```
$ docker update --restart=always 'ID вашего контейнера';
```

Опция `--restart` может принимать следующие значения:

- **no** — перезапуск контейнера производиться не будет
- **on-failure[:max-retries]** — перезапуск контейнера будет происходить max-retries раз в случае любого сбоя
- **always** — перезапуск контейнера будет происходить всегда
- **unless-stopped** — перезапуск контейнера будет происходить всегда до тех пор, пока вы явно не остановите контейнер

Процедура обновления сервера с Docker достаточно простая и выполняется всего одной командой

```
$ docker-machine upgrade default
```

```
Waiting for SSH to be available...
```

```
Detecting the provisioner...
```

```
Upgrading docker...
```

```
Stopping machine to do the upgrade...
```

```
Upgrading machine "default"...
```

```
Default Boot2Docker ISO is out-of-date, downloading the latest release...
```

```
Latest release for github.com/boot2docker/boot2docker is v1.12.4
```

```
Downloading /Users/amaksimov/.docker/machine/cache/boot2docker.iso from  
https://github.com/boot2docker/boot2docker/releases/download/v1.12.4/boot2  
docker.iso.
```

```
0%....10%....20%....30%....40%....50%....60%....70%....80%....90%....100%
```

```
Copying /Users/amaksimov/.docker/machine/cache/boot2docker.iso to  
/Users/amaksimov/.docker/machine/machines/default/boot2docker.iso...
```

```
Starting machine back up...
```

```
(default) Check network to re-create if needed...
```

```
(default) Waiting for an IP...
```

```
Restarting docker...
```

# Специализированные средства обеспечения безопасности Docker

Как только выросла популярность Docker, так сразу же и начали появляться инструменты для обеспечения безопасности контейнеров.

Одним из важных инструментов являются сканеры образов. На данный есть несколько проектов: [Clair от CoreOS](#) и [Docker Security Scanning](#), которые интегрированы в [Docker Hub](#) и могут проверить наличие уязвимостей в образах автоматически, что экономит массу времени. Они могут также отправляют уведомления по электронной почте, когда уязвимость обнаружена и самостоятельно ищут доступные исправления.

Есть еще один пакет для обеспечения безопасности контейнеров — [Twistlock](#). Он обеспечивает всесторонний подход к этому вопросу и охватывает множество областей безопасности. Он не только защищает контейнеры приложений, но и обеспечивает мониторинг, анализ и своевременное реагирование на угрозы. Twistlock является одним из немногих коммерчески доступных решений, которое сосредоточилось исключительно на обеспечении безопасности Docker.

## Усилия Docker сообщества по защите контейнеров

Docker выпустил ряд [обновлений для системы безопасности](#) в течение последних нескольких лет, что решило множество вопросов безопасности, которые не позволяли предприятиям решиться использовать его в 2013 году. Эти разработки позволили облегчить использование Docker среди Enterprise и обеспечили стабильный рост популярности этой технологии.

Основные вопросы в области безопасности, которые отталкивали от Docker в первые дни были namespaces и cgroups. Namespaces обеспечивают наиболее простую форму изоляции между контейнерами, предотвращая их взаимодействия друг с другом. Namespaces дают возможность использовать различные привилегии, которые могут быть назначены различным пользователям.

Cgroups же ограничивают количество ресурсов, которые разрешено использовать каждому отдельному контейнеру. Они дают ограничить доступность к ресурсам, таким как память и процессор для каждого контейнера. К контейнерам можно применить жесткие ограничения, чтобы быть уверенным в том, что процессы будут убиты, если они начинают требовать слишком много ресурсов. Эта мера обеспечивает защиту от эксплуатации от внешнего источника.

В сообществе Docker также разработали [Docker Bench](#), сценарий, который может проверить контейнеры и конфигурации безопасности своих хостов на соответствие с набором лучших практик, предоставленных [Center for Internet Security](#).

[Docker Swarm](#) по состоянию на версию Docker 1.12 встроенный в Docker, также предоставляет некоторые функции безопасности. Он обеспечивает узлы, на которых установлен Docker поддержкой сертификатов Transport Layer Security (TLS) и шифрует обмен данными между ними.

И, наконец, поддержка в Docker функций доверия к содержимому контейнеров — очень элегантное решение обеспечения возможности проверки достоверности содержимого контейнеров.

## Использование Docker от поставщиков услуг

Еще один способ увеличить безопасность Docker (особенно для организаций без большого опыта работы с Docker) — это использовать управляемые хостинг платформы для запуска Docker контейнеров.

Данный подход обеспечивает удобство упрощенного управления и некоторые встроенные функции безопасности (в зависимости от поставщика конечной услуги).

Наилучшим примером является [EC2](#) контейнерный сервис от [Amazon](#) (сокращенно [ECS](#)), который обеспечивает работу контейнеров Docker добавляя к вышеперечисленным функциям обеспечения безопасности дополнительные, такие как Управление идентификацией и доступом ([IAM](#)), а также запуск ваших контейнеров только на принадлежащих вам виртуальных машинах. [Azure](#) и [Google Cloud](#) также предлагают свои собственные услуги, управляющие Docker с интегрированными в услугу всеми передовыми практиками обеспечения безопасности.

Помимо этого, у самого Docker есть коммерческий продукт [Docker Datacenter](#) (или Docker cloud product), который по заявлениям разработчиков является лучшим способом для запуска контейнеров, принадлежащих предприятию, если есть необходимость обеспечить дополнительную защиту Docker контейнеров, разместив их в ЦОД организации и закрыв их брандмауэром, управляемым самой организацией.

## Выводы

В целом, глядя на то, как Docker улучшился за последние годы, можно смело сказать, что он значительно более безопасен сегодня по сравнению с тем, каким он был впервые увидя свет. Вообще говоря, опасения на тему безопасности Docker на ранних этапах его развития были разумными, однако, в настоящее время в Docker реализован целый ряд мер по обеспечению безопасности, которые делают его готовым для любого типа Enterprise предприятия.