



alexryabtsev /
docker-workshop



 Code  Issues 1  Pull requests  Actions  Projects  Security  In



Introduction to Docker tutorial

 MIT license

 134 stars  65 forks  8 watching  Activity

 Public repository

 мастер ▾



 Branches  Tags



alexryabtsev ...

23 марта 2019



[View code](#)

 README.md

Что такое Docker и как его использовать с Python (учебное пособие)



Это вводное руководство по контейнерам Docker. К концу этой статьи вы будете знать, как использовать Docker на вашем локальном компьютере. Наряду с Python мы собираемся запускать контейнеры Nginx и Redis. Эти примеры предполагают, что вы знакомы с основными концепциями этих технологий. Там будет много примеров оболочки, так что продолжайте и откройте терминал.

Содержание

- Что такое Docker?
- Чем это отличается от виртуализации?
- Зачем нам нужен Docker?
- Поддерживаемые платформы
- Установка
- Терминология
- Пример 1: привет, мир
- Пример 2: переменные среды и тома
- Пример 3: Написание вашего первого файла Dockerfile
- Рекомендации по созданию образов
- Alpine images
- Пример 4: Соединение между контейнерами
- Способ работы с Docker
- Заключение

Что такое Docker?

Docker - это инструмент с открытым исходным кодом, который автоматизирует развертывание приложения внутри программного контейнера. Самый простой способ понять идею, лежащую в основе Docker, - сравнить его с, ну ... стандартными транспортными контейнерами.

В свое время транспортные компании сталкивались со следующими проблемами:

- Как перевозить различные (несовместимые) типы товаров бок о бок (например, продукты питания и химикаты, стекло и кирпичи).
- Как обрабатывать пакеты разных размеров с помощью одного транспортного средства.

После внедрения контейнеров кирпичи можно было класть поверх стекла, а химикаты можно было хранить рядом с продуктами питания. Грузы различных размеров можно было помещать в стандартный контейнер и загружать / разгружать одним и тем же транспортным средством.

Давайте вернемся к **контейнерам в разработке программного обеспечения**.

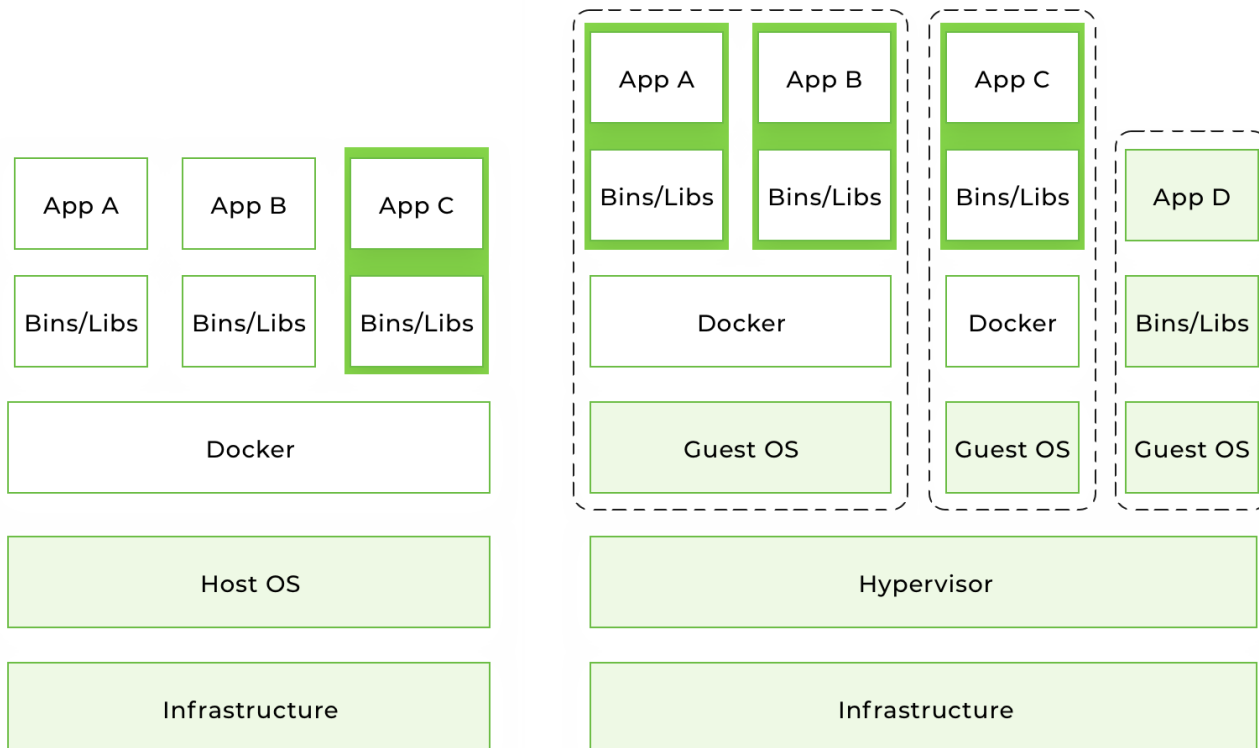
При разработке приложения вам необходимо предоставить свой код вместе со всеми возможными зависимостями, такими как библиотеки, веб-сервер, базы данных и т.д. Вы можете оказаться в ситуации, когда приложение работает на вашем компьютере, но даже не запускается на промежуточном сервере или на компьютере разработчика или QA.

Эту проблему можно решить, изолировав приложение, чтобы сделать его независимым от системы.

Чем это отличается от виртуализации?

Традиционно виртуальные машины использовались для предотвращения такого неожиданного поведения. Основная проблема с виртуальной машиной заключается в том, что "дополнительная ОС" поверх операционной системы хоста добавляет гигабайты пространства для проекта. Большую часть времени на вашем сервере будет размещаться несколько виртуальных машин, которые будут занимать еще больше места. И, кстати, на данный момент большинство поставщиков облачных серверов взимают с вас плату за это дополнительное пространство. Еще одним существенным недостатком виртуальной машины является медленная загрузка.

Docker устраняет все вышеперечисленное, просто распределяя ядро ОС по всем контейнерам, работающим как отдельные процессы основной операционной системы.



Имейте в виду, что Docker - не первая и не единственная платформа контейнеризации. Однако на данный момент Docker является крупнейшим и наиболее мощным игроком на рынке.

Зачем нам нужен Docker?

Краткий список преимуществ включает:

- Ускоренный процесс разработки
- Удобная инкапсуляция приложений
- То же поведение на локальном компьютере / серверах разработки / промежуточных / производственных
- Простой и понятный мониторинг
- Простота масштабирования

Ускоренный процесс разработки

Нет необходимости устанавливать в систему сторонние приложения, такие как PostgreSQL, Redis, Elasticsearch - вы можете запускать их в контейнерах. Docker также дает вам возможность запускать разные версии одного и того же приложения одновременно. Например, предположим, вам нужно выполнить перенос данных вручную из более старой версии Postgres в более новую версию. В микросервисной архитектуре может возникнуть такая ситуация, когда вы хотите создать новый микросервис с новой версией программного обеспечения третьего производителя.

Хранить две разные версии одного и того же приложения на одной хост-ОС может быть довольно сложно. В этом случае контейнеры Docker могут стать идеальным решением - вы получаете изолированные среды для своих приложений и сторонних разработчиков.

Удобная инкапсуляция приложений

Вы можете создать свое приложение как единое целое. Большинство языков программирования, фреймворков и всех операционных систем имеют свои собственные менеджеры упаковки. И даже если ваше приложение может быть упаковано с помощью встроенного менеджера пакетов, создать порт для другой системы может быть сложно.

Docker предоставляет вам унифицированный формат образа для распространения ваших приложений по различным хост-системам и облачным сервисам. Вы можете доставить свое приложение как единое целое со всеми необходимыми зависимостями (включенными в образ), готовыми к запуску.

То же поведение на локальном компьютере / серверах разработки / промежуточных / производственных

Docker не может гарантировать 100%-ный паритет между разработкой, промежуточной обработкой и производством, потому что всегда присутствует человеческий фактор. Но это сводит почти к нулю вероятность ошибки, вызванной различными версиями операционных систем, системными зависимостями и т.д.

При правильном подходе к созданию образов Docker ваше приложение будет использовать тот же базовый образ с той же версией ОС и требуемыми зависимостями.

Простой и понятный мониторинг

Из коробки у вас есть унифицированный способ чтения файлов журналов из всех запущенных контейнеров. Вам не нужно запоминать все конкретные пути, по которым ваше приложение и его зависимости хранят файлы журналов, и писать пользовательские перехваты для обработки этого.

Вы можете интегрировать [внешний драйвер ведения журнала](#) и отслеживать файлы журнала вашего приложения в одном месте.

Простота масштабирования

Правильно упакованное приложение будет охватывать большинство из [двенадцати факторов](#). По своей конструкции Docker заставляет вас следовать своим основным принципам, таким как настройка через переменные среды, обмен данными через порты TCP / UDP и т.д. И если вы все сделали правильно, ваше приложение будет готово к масштабированию не только в Docker.

Поддерживаемые платформы

Родной платформой Docker является Linux, поскольку она основана на функциях, предоставляемых ядром Linux. Однако вы все еще можете запускать ее на macOS и Windows. Разница лишь в том, что в macOS и Windows Docker инкапсулирован в крошечную виртуальную машину. На данный момент Docker для macOS и Windows достиг значительного уровня удобства использования и больше похож на родное приложение.

Установка

Вы можете ознакомиться с инструкциями по установке Docker [здесь](#).

Если вы используете Docker в Linux, вам необходимо выполнить все следующие команды от имени root или добавить своего пользователя в группу docker и повторно войти в систему:

```
sudo usermod -aG docker $(whoami)
```



Terminology

- **Container** -- a running instance that encapsulates required software. Containers are always created from images. A container can expose ports and volumes to interact with other containers or/and the outer world. Containers can be easily killed / removed and re-created again in a very short time. Containers don't keep state.
- **Image** -- the basic element for every container. When you create an image, every step is cached and can be reused ([Copy On Write model](#)). Depending on the image, it can take some time to build. Containers, on the other hand, can be started from images right away.
- **Port** -- a TCP/UDP port in its original meaning. To keep things simple, let's assume that ports can be exposed to the outer world (accessible from the host OS) or connected to other containers -- i.e., accessible only from those containers and invisible to the outer world.

- **Volume** -- can be described as a shared folder. Volumes are initialized when a container is created. Volumes are designed to persist data, independent of the container's lifecycle.
- **Registry** – the server that stores Docker images. It can be compared to Github – you can pull an image from the registry to deploy it locally, and push locally built images to the registry.
- **Docker hub** -- a registry with web interface provided by Docker Inc. It stores a lot of Docker images with different software. Docker Hub is a source of the "official" Docker images made by the Docker team or in cooperation with the original software manufacturer (it doesn't necessary mean that these "original" images are from official software manufacturers). Official images list their potential vulnerabilities. This information is available to any logged-in user. There are both free and paid accounts available. You can have one private image per account and an infinite amount of public images for free. **Docker Store** -- a service very similar to Docker Hub. It's a marketplace with ratings, reviews, etc. My personal opinion is that it's marketing stuff. I'm totally happy with Docker Hub.

OFFICIAL REPOSITORY

nginx ☆
Last pushed: 2 hours ago

Repo Info Tags

Scanned Images ?

1.14-alpine-perl Compressed size: 18 MB Scanned 15 hours ago	ⓘ This image has vulnerabilities <div><div></div></div>
stable-alpine-perl Compressed size: 18 MB Scanned 15 hours ago	ⓘ This image has vulnerabilities <div><div></div></div>
1.14.0-alpine-perl Compressed size: 18 MB Scanned 15 hours ago	ⓘ This image has vulnerabilities <div><div></div></div>
1.14-alpine Compressed size: 9 MB Scanned 15 hours ago	ⓘ This image has vulnerabilities <div><div></div></div>

Example 1: hello world

It's time to run your first container:

```
docker run ubuntu /bin/echo 'Hello world'
```



Console output:

```
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
6b98dfc16071: Pull complete
4001a1209541: Pull complete
6319fc68c576: Pull complete
b24603670dc3: Pull complete
97f170c87c6f: Pull complete
Digest:
sha256:5f4bd3467537cbb563e80db2c3ec95d548a9145d64453b06939c4592d67b6d
Status: Downloaded newer image for ubuntu:latest
Hello world
```



- **docker run** is a command to run a container.
- **ubuntu** is the image you run. For example, the Ubuntu operating system image. When you specify an image, Docker looks first for the image on your Docker host. If the image does not exist locally, then the image is pulled from the public image registry -- Docker Hub.
- **/bin/echo 'Hello world'** is the command that will run inside a new container. This container simply prints "Hello world" and stops the execution.

Let's try to create an interactive shell inside a Docker container:

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



- **-t** flag assigns a pseudo-tty or terminal inside the new container.
- **-i** flag allows you to make an interactive connection by grabbing the standard input (STDIN) of the container.
- **--rm** flag automatically removes the container when the process exits. By default, containers are not deleted. This container exists until we keep the shell session and terminates when we exit the session (like an SSH session with a remote server).

If you want to keep the container running after the end of the session, you need to daemonize it:

```
docker run --name daemon -d ubuntu /bin/sh -c "while true; do echo hello w
```



- **--name daemon** assigns daemon name to a new container. If you don't specify a name explicitly, Docker will generate and assign it automatically.
- **-d** flag runs the container in the background (i.e., daemonizes it).

Let's see what containers we have at the moment:


```
docker ps -a
```



Console output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
1fc8cee64ec2	ubuntu	"/bin/sh -c 'while..." seconds daemon	32 seconds ago	Up 30
c006f1a02edf	ubuntu	"/bin/echo 'Hello ..." About a minute ago gifted_nobel	About a minute ago	Exited (0)



- **docker ps** is a command to list containers.
- **-a** shows all containers (without -a flag ps will show only running containers).

The **ps** shows us that we have two containers:

- **gifted_nobel** (the name for this container was generated automatically – it will be different on your machine). It's the first container we created, the one that printed 'Hello world' once.
- **daemon** -- the third container we created, which runs as a daemon.

Note: there is no second container (the one with interactive shell) because we set the **--rm option**. As a result, this container is automatically deleted right after execution.

Let's check the logs and see what the daemon container is doing right now:

```
docker logs -f daemon
```



Console output:

```
...  
hello world  
hello world  
hello world
```



- **docker logs** fetch the logs of a container.
- **-f** flag to follow the log output (works actually like **tail -f**).

Now let's stop the daemon container:

```
docker stop daemon
```



Make sure the container has stopped.

```
docker ps -a
```



Console output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
1fc8cee64ec2	ubuntu	"/bin/sh -c 'while..." seconds ago	5 minutes ago	Exited (137) 5
c006f1a02edf	ubuntu	"/bin/echo 'Hello ..." minutes ago	6 minutes ago	Exited (0) 6



The container is stopped. We can start it again:

```
docker start daemon
```



Let's ensure that it's running:

```
docker ps -a
```



Console output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
1fc8cee64ec2	ubuntu	"/bin/sh -c 'while..." daemon	5 minutes ago	Up 3 seconds
c006f1a02edf	ubuntu	"/bin/echo 'Hello ..." minutes ago	6 minutes ago	Exited (0) 7



Now, stop it again and remove all the containers manually:

```
docker stop daemon  
docker rm <your first container name>  
docker rm daemon
```



To remove all containers, we can use the following command:

```
docker rm -f $(docker ps -aq)
```



- **docker rm** is the command to remove the container.

- **-f** flag (for rm) stops the container if it's running (i.e., force deletion).
- **-q** flag (for ps) is to print only container IDs.

Example 2: Environment variables and volumes

Starting from this example, you'll need several additional files you can find on my [GitHub repo](#). You can clone my repo or simply use the [following link](#) to download the sample files.

It's time to create and run more a meaningful container, like **Nginx**.

Change the directory to **examples/nginx**:

```
docker run -d --name "test-nginx" -p 8080:80 -v $(pwd):/usr/share/nginx/html
```

Warning: This command looks quite heavy, but it's just an example to explain volumes and env variables. In 99% of real-life cases, you won't start Docker containers manually -- you'll use orchestration services (we'll cover [docker-compose](#) in [example #4](#)) or write a custom script to do it.

Console output:

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
683abbb4ea60: Pull complete
a470862432e2: Pull complete
977375e58a31: Pull complete
Digest:
sha256:a65beb8c90a08b22a9ff6a219c2f363e16c477b6d610da28fe9cba37c2c3a2ac
Status: Downloaded newer image for nginx:latest
afa095a8b81960241ee92ecb9aa689f78d201cff2469895674cec2c2acdcc61c
```

- **-p** is a ports mapping **HOST PORT:CONTAINER PORT**.
- **-v** is a volume mounting **HOST DIRECTORY:CONTAINER DIRECTORY**.

Important: run command accepts only absolute paths. In our example, we've used **\$(pwd)** to set the current directory absolute path.

Now check this [url](#) in your web browser.

We can try to change **/example/nginx/index.html** (which is mounted as a volume to **/usr/share/nginx/html** directory inside the container) and refresh the page.

Let's get the information about **test-nginx** container:

```
docker inspect test-nginx
```



This command displays system-wide information about the Docker installation. This information includes the kernel version, number of containers and images, exposed ports, mounted volumes, etc.

Example 3: Writing your first Dockerfile

To build a Docker image, you need to create a Dockerfile. It is a plain text file with instructions and arguments. Here is the description of the instructions we're going to use in our next example:

- **FROM** -- set base image
- **RUN** -- execute command in container
- **ENV** -- set environment variable
- **WORKDIR** -- set working directory
- **VOLUME** -- create mount-point for a volume
- **CMD** -- set executable for container

You can check [Dockerfile reference](#) for more details.

Let's create an image that will get the contents of the website with **curl** and store it to the text file. We need to pass website url via environment variable **SITE_URL**. Resulting file will be placed in a directory mounted as a volume.

Place a file name **Dockerfile** in **examples/curl** directory with the following contents:

```
FROM ubuntu:latest
RUN apt-get update \
    && apt-get install --no-install-recommends --no-install-suggests -y curl \
    && rm -rf /var/lib/apt/lists/*
ENV SITE_URL http://example.com/
WORKDIR /data
VOLUME /data
CMD sh -c "curl -Lk $SITE_URL > /data/results"
```



Dockerfile is ready. It's time to build the actual image.

Go to **examples/curl** directory and execute the following command to build an image:

```
docker build . -t test-curl
```



Console output:

```

Sending build context to Docker daemon 3.584kB
Step 1/6 : FROM ubuntu:latest
--> 113a43faa138
Step 2/6 : RUN apt-get update      && apt-get install --no-install-recommends
--no-install-suggests -y curl    && rm -rf /var/lib/apt/lists/*
--> Running in ccc047efe3c7
Get:1 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]
Get:2 http://security.ubuntu.com/ubuntu bionic-security InRelease [83.2 kB]
...
Removing intermediate container ccc047efe3c7
--> 8d10d8dd4e2d
Step 3/6 : ENV SITE_URL http://example.com/
--> Running in 7688364ef33f
Removing intermediate container 7688364ef33f
--> c71f04bdf39d
Step 4/6 : WORKDIR /data
Removing intermediate container 96b1b6817779
--> 1ee38cca19a5
Step 5/6 : VOLUME /data
--> Running in ce2c3f68dbbb
Removing intermediate container ce2c3f68dbbb
--> f499e78756be
Step 6/6 : CMD sh -c "curl -Lk $SITE_URL > /data/results"
--> Running in 834589c1ac03
Removing intermediate container 834589c1ac03
--> 4b79e12b5c1d
Successfully built 4b79e12b5c1d
Successfully tagged test-curl:latest

```

- **docker build** command builds a new image locally.
- **-t** flag sets the name tag to an image.

Now we have the new image, and we can see it in the list of existing images:

```
docker images
```

Console output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
test-curl	latest	5ebb2a65d771	37 minutes ago	180 MB
nginx	latest	6b914bbcb89e	7 days ago	182 MB
ubuntu	latest	0ef2e08ed3fa	8 days ago	130 MB

We can create and run the container from the image. Let's try it with the default parameters:

```
docker run --rm -v $(pwd)/vol:/data/:rw test-curl
```



To see results saved to file run:

```
cat ./vol/results
```



Let's try with **facebook.com**:

```
docker run --rm -e SITE_URL=https://facebook.com/ -v $(pwd)/vol:/data/:rw -c
```



To see the results saved to file run:

```
cat ./vol/results
```



Best practices for creating images

- Include only **necessary context** -- use a **.dockerignore** file (like .gitignore in git)
- Avoid installing **unnecessary packages** -- it will consume extra disk space.
- **Use cache**. Add context which changes a lot (for example, the source code of your project) at the end of Dockerfile -- it will utilize Docker cache effectively.
- **Be careful with volumes**. You should remember what data is in volumes. Because volumes are persistent and don't die with the containers, the next container will use data from the volume created by the previous container.
- Use **environment variables** (in RUN, EXPOSE, VOLUME). It will make your Dockerfile more flexible.

Alpine images

A lot of Docker images (versions of images) are created on top of **Alpine Linux** -- this is a lightweight distro that allows you to reduce the overall size of Docker images.

I recommend that you use images based on Alpine for third-party services, such as Redis, Postgres, etc. For your app images, use images based on **buildpack** -- it will be easy to debug inside the container, and you'll have a lot of pre-installed system-wide requirements.

Only you can decide which base image to use, but you can get the maximum benefit by using one basic image for all images, because in this case the cache will be used more effectively.

Example 4: Connection between containers

Docker compose -- is an CLI utility used to connect containers with each other.

You can install docker-compose [via pip](#):

```
sudo pip install docker-compose
```



In this example, I am going to connect Python and Redis containers.

```
version: '3.6'
services:
  app:
    build:
      context: ./app
    depends_on:
      - redis
    environment:
      - REDIS_HOST=redis
    ports:
      - "5000:5000"
  redis:
    image: redis:3.2-alpine
    volumes:
      - redis_data:/data
volumes:
  redis_data:
```



Go to **examples/compose** and execute the following command:

```
docker-compose up
```



Console output:



```
Building app
Step 1/9 : FROM python:3.6.3
3.6.3: Pulling from library/python
f49cf87b52c1: Pull complete
7b491c575b06: Pull complete
b313b08bab3b: Pull complete
51d6678c3f0e: Pull complete
09f35bd58db2: Pull complete
1bda3d37eead: Pull complete
9f47966d4de2: Pull complete
9fd775bfe531: Pull complete
Digest:
sha256:cdef88d8625cf50ca705b7abfe99e8eb33b889652a9389b017eb46a6d2f1aaf3
```

```
Status: Downloaded newer image for python:3.6.3
---> a8f7167de312
Step 2/9 : ENV BIND_PORT 5000
---> Running in 3b6fe5ca226d
Removing intermediate container 3b6fe5ca226d
---> 0b84340fa920
Step 3/9 : ENV REDIS_HOST localhost
---> Running in a4f9a1d6f541
Removing intermediate container a4f9a1d6f541
---> ebe63bf5959e
Step 4/9 : ENV REDIS_PORT 6379
---> Running in fd06aa65fd33
Removing intermediate container fd06aa65fd33
---> 2a581c31ff4f
Step 5/9 : COPY ./requirements.txt /requirements.txt
---> 671093a12829
Step 6/9 : RUN pip install -r /requirements.txt
---> Running in b8ea53bc6ba6
Collecting flask==1.0.2 (from -r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dacb4696386f
1.0.2-py2.py3-none-any.whl (91kB)
Collecting redis==2.10.6 (from -r /requirements.txt (line 2))
  Downloading
https://files.pythonhosted.org/packages/3b/f6/7a76333cf0b9251ecf49efff635015171f
2.10.6-py2.py3-none-any.whl (64kB)
Collecting click>=5.1 (from flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/34/c1/8806f99713ddb993c5366c362b2f908f1f
6.7-py2.py3-none-any.whl (71kB)
Collecting Jinja2>=2.10 (from flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/7f/ff/ae64bacdfc95f27a016a7bed8e8686763f
2.10-py2.py3-none-any.whl (126kB)
Collecting itsdangerous>=0.24 (from flask==1.0.2->-r /requirements.txt (line
1))
  Downloading
https://files.pythonhosted.org/packages/dc/b4/a60bcdba945c00f6d608d8975131ab3f2f
0.24.tar.gz (46kB)
Collecting Werkzeug>=0.14 (from flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/20/c4/12e3e56473e52375aa29c4764e70d1b8f3
0.14.1-py2.py3-none-any.whl (322kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10->flask==1.0.2->-r
/requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/4d/de/32d741db316d8fdb7680822dd37001ef7a
1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous: started
  Running setup.py bdist_wheel for itsdangerous: finished with status 'done'
  Stored in directory:
/root/.cache/pip/wheels/2c/4a/61/5599631c1554768c6290b08c02c72d7317910374ca602f
```



```
Running setup.py bdist_wheel for MarkupSafe: started
Running setup.py bdist_wheel for MarkupSafe: finished with status 'done'
Stored in directory:
/root/.cache/pip/wheels/33/56/20/ebe49a5c612fffe1c5a632146b16596f9e64676768661e
Successfully built itsdangerous MarkupSafe
Installing collected packages: click, MarkupSafe, Jinja2, itsdangerous,
Werkzeug, flask, redis
Successfully installed Jinja2-2.10 MarkupSafe-1.0 Werkzeug-0.14.1 click-6.7
flask-1.0.2 itsdangerous-0.24 redis-2.10.6
You are using pip version 9.0.1, however version 10.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Removing intermediate container b8ea53bc6ba6
---> 3117d3927951
Step 7/9 : COPY ./app.py /app.py
---> 84a82fa91773
Step 8/9 : EXPOSE $BIND_PORT
---> Running in 8e259617b7b5
Removing intermediate container 8e259617b7b5
---> 55f447f498dd
Step 9/9 : CMD [ "python", "/app.py" ]
---> Running in 2ade293ecb25
Removing intermediate container 2ade293ecb25
---> b85b4246e9f8
```

```
Successfully built b85b4246e9f8
Successfully tagged compose_app:latest
WARNING: Image for service app was built because it did not already exist. To
rebuild this image you must use `docker-compose build` or `docker-compose up
--build`.
```

```
Creating compose_redis_1 ... done
```

```
Creating compose_app_1 ... done
```

Attaching to compose_redis_1, compose_app_1

```
redis_1 | 1:C 08 Jul 18:12:21.851 # Warning: no config file specified, using
the default config. In order to specify a config file use redis-server
/path/to/redis.conf
```

```
redis_1 |  
redis_1 |  
redis_1 | Redis 3.2.12 (00000000/0)  
64 bit  
redis_1 |  
redis_1 | Running in standalone mode  
redis_1 | Port: 6379  
redis_1 | PID: 1  
redis_1 |  
redis_1 | http://redis.io  
redis_1 |  
redis_1 |  
redis_1 |  
redis_1 |
```

```

redis_1 | 
redis_1 | 1:M 08 Jul 18:12:21.852 # WARNING: The TCP backlog setting of 511
redis_1 | cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower
redis_1 | value of 128.
redis_1 | 1:M 08 Jul 18:12:21.852 # Server started, Redis version 3.2.12
redis_1 | 1:M 08 Jul 18:12:21.852 # WARNING overcommit_memory is set to 0!
redis_1 | Background save may fail under low memory condition. To fix this issue add
redis_1 | 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the
redis_1 | command 'sysctl vm.overcommit_memory=1' for this to take effect.
redis_1 | 1:M 08 Jul 18:12:21.852 # WARNING you have Transparent Huge Pages
redis_1 | (THP) support enabled in your kernel. This will create latency and memory
redis_1 | usage issues with Redis. To fix this issue run the command 'echo never >
redis_1 | /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your
redis_1 | /etc/rc.local in order to retain the setting after a reboot. Redis must be
redis_1 | restarted after THP is disabled.
redis_1 | 1:M 08 Jul 18:12:21.852 * The server is now ready to accept
redis_1 | connections on port 6379
app_1    | * Serving Flask app "app" (lazy loading)
app_1    | * Environment: production
app_1    | WARNING: Do not use the development server in a production
app_1    | environment.
app_1    | Use a production WSGI server instead.
app_1    | * Debug mode: on
app_1    | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
app_1    | * Restarting with stat
app_1    | * Debugger is active!
app_1    | * Debugger PIN: 170-528-240

```

web browser and check it.

How to use docker-compose is a topic for a separate tutorial. To get started, you can play with some images from Docker Hub. If you want to create your own images, follow the best practices listed above. The only thing I can add in terms of using docker-compose is that you should **always give explicit names to your volumes** in docker-compose.yml ((if the image has volumes). This simple rule will save you from an issue in the future when you'll be inspecting your volumes.

```

version: '3.6'
services:
  ...
  redis:
    image: redis:3.2-alpine
    volumes:
      - redis_data:/data
volumes:
  redis_data:

```



In this case **redis_data** will be the name inside the docker-compose.yml file; for the real volume name, it will be prepended with project name prefix.

To see volumes run:

```
docker volume ls
```



Console output:

DRIVER	VOLUME NAME
local	apptest_redis_data



Without an explicit volume name, there will be UUID. Here's an example from my local machine:

DRIVER	VOLUME NAME
local	
local	ec1a5ac0a2106963c2129151b27cb032ea5bb7c4bd6fe94d9dd22d3e72b2a41b
local	f3a664ce353ba24dd43d8f104871594de6024ed847054422bbdd362c5033fc4c
local	f81a397776458e62022610f38a1bfe50dd388628e2badc3d3a2553bb08a5467f
local	f84228acbf9c5c06da7be2197db37f2e3da34b7e8277942b10900f77f78c9e64
local	f9958475a011982b4dc8d8d8209899474ea4ec2c27f68d1a430c94bcc1eb0227
local	ff14e0e20d70aa57e62db0b813db08577703ff1405b2a90ec88f48eb4cdc7c19
local	polls_pg_data
local	polls_public_files
local	polls_redis_data
local	projectdev_pg_data
local	projectdev_redis_data



Docker way

Docker has some restrictions and requirements, depending on the architecture of your system (applications that you pack into containers). You can ignore these requirements or find some workarounds, but in this case, you won't get all the benefits of using Docker. My strong advice is to follow these recommendations:

- **1 application = 1 container.**
- Run process in the **foreground** (don't use systemd, upstart or any other similar tools).
- **Keep data out of container** -- use volumes.

- **Do not use SSH** (if you need to step into container you can use `docker exec` command).
- **Avoid manual configurations** (or actions) inside container.

Conclusion

To summarize this tutorial, alongside with IDE and Git, Docker has become a must-have developer tool. It's a production-ready tool with a rich and mature infrastructure.

Docker can be used on all types of projects, regardless of size and complexity. In the beginning, you can start with [compose](#) and [Swarm](#). When the project grows, you can migrate to cloud services like [Amazon Container Services](#) or [Kubernetes](#).

Like standard containers used in cargo transportation, wrapping your code in Docker containers will help you build faster and more efficient CI/CD processes. This is not just another technological trend promoted by a bunch of geeks -- it's a new paradigm that is already being used in the architecture of large companies like [PayPal](#), [Visa](#), [Swisscom](#), [General Electric](#), [Splink](#), etc.

РЕЛИЗЫ

Релизы не опубликованы

Пакетов

Пакеты не опубликованы