KARPOV.COURSES >>>
КОНСПЕКТ

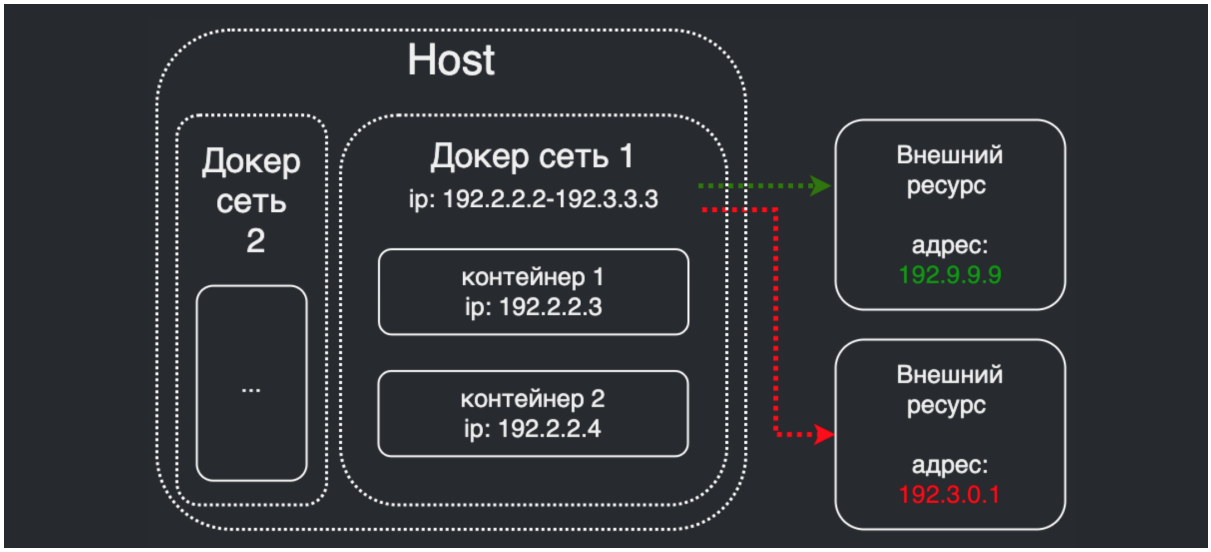
> Конспект >5 урок > Docker Swarm + Container Lifecycle

> Оглавление

> Проблемы Docker-Compose

Прежде чем мы начнём знакомиться с Docker Swarm, давайте обсудим пару моментов, связанных с Docker-Compose.

Связь, которая создаётся в Docker-Compose между сервисами, позволяющими пользоваться Service Discovery "из коробки", обеспечивается за счёт Docker-сетей. Каждый раз, когда мы запускаем Docker-Compose, создаются не только сервисы, но ещё и сеть для сервисов. Соответственно, если часто использовать Docker-Compose, то будет создаваться много докеровских сетей. Если же на этой машине есть ещё либо production, либо какие-то корпоративные ресурсы в ваших подсетях, то могут появиться проблемы.



На схеме выше изображено следующее: пусть есть Host, на котором развёрнуто несколько Docker-Compose'ов, например Docker-сеть 1 и Docker-сеть 2. Это две сети, созданные, соответственно, двумя разными Compose'ами. Справа есть два блока с внешними ресурсами (в данном случае всё находится в одной подсети). На каждую Docker-сеть выделяется какой-то диапазон адресов (подсеть). Когда таких сетей становится много, свободный диапазон может попросту закончиться. Это чревато тем, что в случае попытки обращения к внешнему ресурсу с адресом, который пересекается с одним из диапазонов Docker-сети (в нашем случае с 1-й), контейнер (например, контейнер 1) уже не сможет добраться до него, потому что пакет не сможет покинуть даже саму Docker-сеть, поскольку сетевые механизмы Docker'а "подумают", что он находится внутри сети 1, и, соответственно, пакет будет направлен по адресу внутри подсети. Таким образом, этот внешний ресурс будет экранирован вашей Docker-сетью. В логах вы увидите, что этот внешний ресурс был недоступен. К сожалению, этот эффект будет распространён на весь Host: вы потеряете доступ к этому внешнему ресурсу. Это произойдёт из-за того, что эти подсети и интерфейсы к ним хотя и являются Docker-абстракцией, но тем не менее создаются на уровне ОС и начинают применяться уже всеми процессами.

Эту проблему можно решить довольно быстро, если вы понимаете, какая Docker-сеть является для вас проблемной. Вы можете удалить все контейнеры из неё и отключить саму сеть. Таким образом, доступ к внешнему ресурсу восстановится.

Ранее мы сказали, что такое может возникнуть, когда свободные диапазоны для сетей закончатся, но на самом деле это может произойти в любой момент, когда Docker будет создавать Docker-сеть, т.к. он не сканирует вашу сеть на наличие свободных слотов для своего диапазона. Это может случиться и на первой сети.

> Docker Swarm

Теперь перейдём к Swarm-кластеру. Чтобы понять, что это такое, необходимо изучить его функционал. Из названия уже следует, что это какой-то распределённый набор машин, объединённых в вычислительную группу. Основное отличие от Docker-Compose заключается в том, что он может управлять сервисами на нескольких хостах: есть несколько Docker-демонов, которыми мы раньше пользовались, но теперь мы объединим их в кластер и с какой-то одной машины будем управлять нашими контейнерами на всех кластерах и всех машинах, запуская команды на одной машине. В рамках курса мы рассмотрим Swarm-кластеры в Single-Host режиме, т.е. у нас будет одна машина, которая будет одновременно и worker'ом и manager'ом кластера. Мы будем писать на ней команды для кластера, и на ней же они будут исполняться.

Описание Docker-Compose можно использовать и для деплоя в Docker Swarm, потому что они выполняют схожий функционал: перезапуск, rollback, спецификации образов. Также Docker-Compose уже использовал понятие сервиса, которого не было в Docker. А для Docker Swarm сервис — это центральная сущность.

Давайте воспользуемся Docker-Compose файлом с предыдущего quickstart'а, чтобы задеплоить сервисы `web` и `redis` в Swarm-кластер. Вот наш Docker-compose файл:

```
version: "3.9"
services:
  web:
    image: "45_web:latest"
```

```
    ports:
      - "5000:5000"
  redis:
    image: "redis:latest"
```

Используя команду `docker stack deploy -c docker-compose.yml quickstart`, мы можем задеплоить несколько сервисов сразу. Заметьте, что мы не можем больше использовать директиву `build` в описании нашего файла, потому что для создания сервисов и деплоя в Swarm-кластер мы уже должны иметь все наши образы собранными на машине или в registry, которая сконфигурирована для Swarm-кластера.

Результат будет следующим:

```
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker stack deploy -c docker-compose.
yml quickstart

Creating service quickstart_web
Creating service quickstart_redis
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker service ls
ID             NAME               MODE         REPLICAS   IMAGE          PORTS
05heynre6dh1   quickstart_redis   replicated   0/1        redis:latest
vf16ksgw33um   quickstart_web     replicated   0/1        45_web:latest   *:5000->5000/tcp
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker ps
CONTAINER ID   IMAGE          COMMAND       CREATED         STATUS                PORTS      NA
MES
ba69c1116f0f   45_web:latest   "flask run"   6 seconds ago   Up Less than a second   5000/tcp   qu
ickstart_web.1.xf4qp67fxkuk3ipqueemu7lyt
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 %
```

Запустился только один контейнер

Видим, что сервисы начинают создаваться. Но они запускаются не все сразу — сначала запустился наш контейнер с `web`-сервисом. Это связано с тем, что выполнение таски для Swarm-кластера — не совсем тривиальная задача (он создаёт внутренние контейнеры для выполнения, которые мы не видим).

```
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker ps
CONTAINER ID   IMAGE          COMMAND       CREATED          STATUS         PORTS      NAMES
ba69c1116f0f   45_web:latest   "flask run"   13 seconds ago   Up 7 seconds   5000/tcp   quickstart
_web.1.xf4qp67fxkuk3ipqueemu7lyt
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS
  NAMES
0f6b164a45cc   redis:latest    "docker-entrypoint.s…"   19 seconds ago   Up 15 seconds   6379/tcp
  quickstart_redis.1.391rjjai0t8390yep1ze0dbj2
ba69c1116f0f   45_web:latest   "flask run"              31 seconds ago   Up 24 seconds   5000/tcp
  quickstart_web.1.xf4qp67fxkuk3ipqueemu7lyt
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 %
```

Запустились уже все контейнеры

Затем запустился и второй сервис. Через команду `docker logs <container id>` можно увидеть логи и убедиться, что всё в порядке. На самом деле вместе с тем произошло создание Docker-сети для Swarm'а для объединения этих контейнеров: сначала были созданы два контейнера, а затем они были связаны через сеть.

> Масштабирование сервиса через Swarm

Теперь давайте попробуем отмасштабировать сервис как и на прошлом занятии: для этого нам нужно отправить команду для обновления сервиса `web`. Воспользуемся следующей командой:

```
docker service update --replicas <replicas number> <service name>
```

```
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker service ls
ID             NAME               MODE         REPLICAS   IMAGE          PORTS
05heynre6dh1   quickstart_redis   replicated   1/1        redis:latest
vf16ksgw33um   quickstart_web     replicated   1/1        45_web:latest   *:5000->5000/tcp
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker service update --replicas 2 quickstart_we
b
quickstart_web
overall progress: 2 out of 2 tasks
1/2: running
2/2: running
verify: Service converged
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS      NAMES
3c546e9fe8a9   45_web:latest   "flask run"              25 seconds ago   Up 18 seconds   5000/tcp   quicksta
rt_web.2.lppijf6gnzgb2y9ix6ykri2cm
0f6b164a45cc   redis:latest    "docker-entrypoint.s…"   3 minutes ago    Up 3 minutes    6379/tcp   quicksta
rt_redis.1.391rjjai0t8390yep1ze0dbj2
ba69c1116f0f   45_web:latest   "flask run"              4 minutes ago    Up 3 minutes    5000/tcp   quicksta
rt_web.1.xf4qp67fxkuk3ipqueemu7lyt
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker logs 3c546e9fe8a9
 * Serving Flask app 'app.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.23.0.5:5000/ (Press CTRL+C to quit)
```

Можно увидеть, что первая реплика уже есть и идёт создание только второй реплики. Также после этого происходит верификация того, что сервис запустился и не падает. С помощью команды `docker ps` можно увидеть эти две реплики и опять же с помощью `docker logs <container id>` посмотреть логи.

Обратите внимание, что когда мы делали это через Docker-compose у нас не получилось запустить два контейнера одновременно: мы пришли к выводу, что понадобится дополнительный балансировщик. А сейчас мы смогли отмасштабировать сервис, причём контейнеры сразу не упали и даже запустились с использованием своих портов, даже несмотря на то что в docker-compose файле стоит инструкция для их проброса. Это связано с тем, что Swarm-кластер содержит в себе ещё больше абстракций, которые не только связывают между собой Docker-сети на разных машинах, но и дают возможность балансировки запросов между контейнерами.

Можно отправить много запросов через `curl -X GET localhost:5000` и увидеть, что возвращаются ответы. Если же посмотреть в логи контейнеров, можно увидеть, что каждый из них обработал несколько запросов:

```
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker logs 3c546e9fe8a9
 * Serving Flask app 'app.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.23.0.5:5000/ (Press CTRL+C to quit)
10.0.0.2 - - [16/Jun/2021 20:05:54] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:07:58] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:07:59] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:07:59] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:08:00] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:08:00] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:08:00] "GET / HTTP/1.1" 200 -
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker logs ba69c1116f0f
 * Serving Flask app 'app.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.23.0.3:5000/ (Press CTRL+C to quit)
10.0.0.2 - - [16/Jun/2021 20:07:39] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:07:58] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:07:59] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:07:59] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:08:00] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:08:00] "GET / HTTP/1.1" 200 -
10.0.0.2 - - [16/Jun/2021 20:08:01] "GET / HTTP/1.1" 200 -
```

Таким образом, мы задеплоили сервис и отмасштабировали его, а балансировка досталась "из коробки". Всё это мы сделали через Docker-compose описание с помощью команд `stack` и `deploy`, но для продакшена так делать не рекомендуется, и вот почему:

1. Совместное описание сервисов. В микросервисной архитектуре каждый сервис имеет своё описание и конфигурацию.
2. В Docker-compose некоторые вещи делались неявно (переиспользовались volume'ы, названия сервисов и сети создавались автоматически), тогда как в Docker-swarm всё делалось явно.

Давайте посмотрим, как управлять сервисами в Swarm через команду `service` (`docker service --help`):

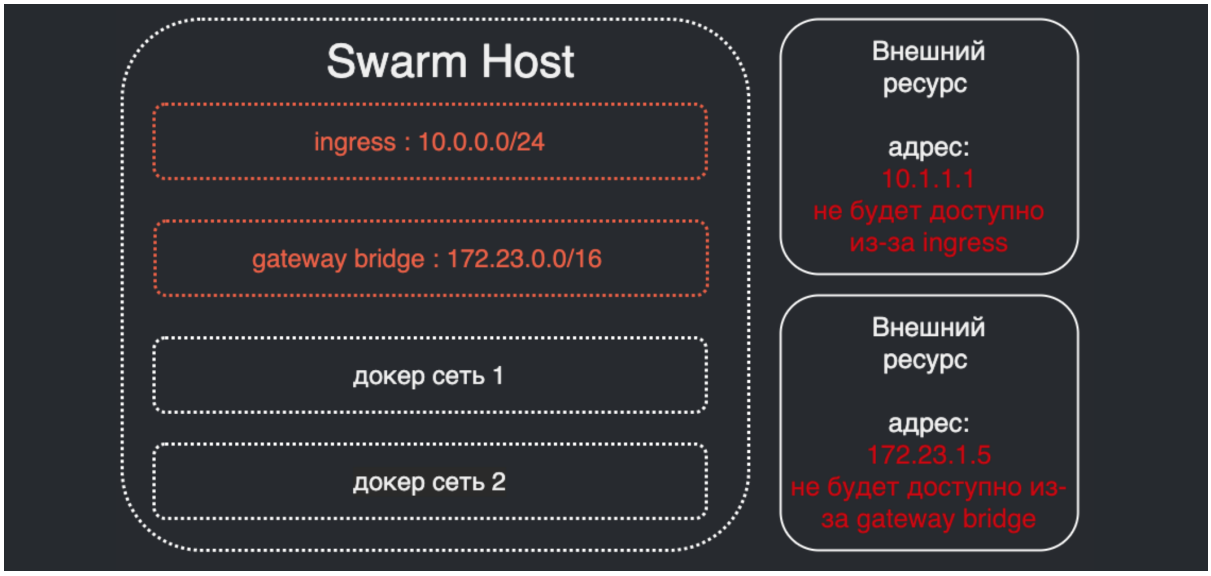
```
(anaconda3-5.3.1) (base) waryak@MacBook-Pro-Vladislav 4.6 % docker service --
help

Usage:  docker service COMMAND

Manage services

Commands:
  create      Create a new service
  inspect     Display detailed information on one or more services
  logs        Fetch the logs of a service or task
  ls          List services
  ps          List the tasks of one or more services
  rm          Remove one or more services
  rollback    Revert changes to a service's configuration
  scale       Scale one or multiple replicated services
  update      Update a service
```

Список небольшой, но мы можем создавать сервисы через `create`, получать информацию о них с помощью `inspect`, просматривать логи через `logs`, выводить список сервисов через `ls`, удалять сервисы через `remove`, масштабировать их через `scale`, обновлять с их помощью `update` и откатывать изменения через `rollback`. На самом деле в том же Kubernetes всё хоть и гораздо сложнее, но вы, скорее всего, будете пользоваться только аналогами этих команд и вряд ли будете использовать что-то сверх этого. В Swarm также есть также конфиги и секреты, которые могут распространяться по всем узлам кластера. Конечно, в Kubernetes есть некоторые функции, которых нет в Swarm.

Вообще, если вы пообщаетесь с DevOps'ами, то поймёте, что многие из них откровенно не любят Swarm. Однако не из-за того, что Swarm плохой, ненадёжный или нестабильный, а из-за того, что они привыкли к богатому функционалу, большому количеству материалов и крупному сообществу, т.е. всему, что делает Kubernetes настоящим танком. Скорее всего, в нашем случае Swarm — это идеальный вариант, поскольку нам нужно просто деплоить ML-сервисы не с таким большим функционалом, как у крупных брендов. Наши сервисы просто должны надёжно работать, иметь минимум overhead'a, быть простыми в поддержке и настройке и минимальными в части конфигурации для деплоя. Речь идёт не о таком большом количестве контейнеров и узлов в кластере, как, например, в Kubernetes — практикой проверено, что у него может быть несколько десятков и даже сотен нод (узлов) в кластере. При этом за Swarm можно не переживать примерно при 10 узлах в кластере. Что произойдёт, если в Swarm кластере будет больше нод, честно говоря, не совсем ясно, однако практика показывает, что при небольшом количестве узлов, которые на самом деле могут обрабатывать довольно неплохую нагрузку, Swarm действительно обладает минимально необходимым набором абстракций: в нём нет ничего лишнего и если у вас есть какой-то корпоративный Kubernetes или какой-то его аналог, то вы даже можете столкнуться с ситуацией, когда ваш собственный Swarm-кластер на нодах вашей команды при большей нагрузке будет давать более высокую производительность, чем тот же Kubernetes. Это связано с тем, что в нём больше сетевых абстракций и его инфраструктура сложнее, чем у довольно простого Swarm.

> Проблема со Swarm-сетями

Давайте посмотрим на схему, которая очень похожа на ту, что мы видели ранее. Суть проблемы такая же: появляются какие-то сетевые абстракции и подсети, которые перекрывают своим диапазоном внешние ресурсы.



В случае со Swarm добавляется абстракция `ingress`, которая позволяет балансировать трафик между контейнерами, и абстракция `gateway bridge`, которая связывает ноды Swarm между собой. Проблема усложняется тем, что эти две абстракции вы не можете просто так взять и удалить, как это было с сетями в Docker-compose. Эти сети `ingress` и `gateway bridge` создаются один раз при инициализации кластера, и без них Swarm функционировать не сможет. То есть если решать проблему удалением, то придётся удалять весь кластер и создавать его заново, надеясь, что `ingress` и `gateway bridge` займут другие диапазоны. Скорее всего проблему нужно решать настройкой Docker-демонов таким образом, чтобы при инициализации Swarm-кластера сетевые абстракции `ingress` и `gateway bridge` создавались с заранее специфицированными диапазонами.

> Container Lifecycle

В самом начале этого модуля мы договорились, что лучше проектировать stateless контейнеры и проектировать их так, чтобы повышать эфемерность контейнеров (чтобы их было просто запускать, перезапускать, переносить и т.д.). На предыдущих занятиях мы готовили код самого Python-процесса к этим перезапускам: мы узнали, что нужно делать на старте приложения, как внутри отлавливать сигналы и переопределять поведение приложения в этих случаях. Теперь посмотрим на это со стороны контейнеров.

До этого момента нас интересовало только управление деплойментами. Сейчас же мы обсудим жизненный цикл самих контейнеров и для Docker, и для Docker-compose, и для Docker Swarm. Мы попытаемся понять, что нужно делать с нашим контейнером, чтобы мы могли не боясь масштабировать, откатывать и обновлять приложение. В Docker, Swarm и Compose есть перезапуски. Существуют разные политики перезапуска: например, можно перезапускать контейнер, если он "упал" из-за ошибки, или перезапускать контейнер всегда, даже в случае `exit code 0`, но суть от этого не меняется. Контейнеры перезапускаются, когда случилось их завершение и это завершение — единственное, чем мы до этого управляли в приложениях, и единственное, на что реагировали наши системы. Но этого явно недостаточно: мы уже сталкивались со случаями, когда контейнер уже "поднялся", а процесс ещё не готов принимать трафик, потому что подключения ещё не были установлены. Ещё один частый случай — приложение работает, но некорректно. Кроме того, бывают ситуации, когда и контейнер вроде работает, и приложение внутри здорово, но в данный момент оно не может обслуживать новый запрос. В таком случае нужно временно выключить трафик на этот контейнер, при этом оставив контейнер рабочим, чтобы он успел завершить текущие задачи.

Для всего этого есть подходящие инструменты в Kubernetes. В Docker, Swarm и Compose (далее DSC) нет всего того, что есть в Kubernetes, но есть почти всё то же самое с минимальными отличиями. Поэтому рассмотрим определения этих инструментов и их аналоги в DSC.

> Readiness Probe

Readiness Probe как раз помогает в последнем случае, когда и приложение, и контейнер работоспособны, но очень сильно нагружены и нужно временно приостановить подачу трафика на этот узел. Именно эту probe у себя в Swarm мы имплементировать не сможем, т.к. нужно иметь возможность отключать контейнер, оставляя его живым и работающим. К сожалению, в Swarm такого механизма нет, и сами мы его сделать не сможем, потому что не работаем с Service Discovery Swarm'a. Если Readiness Probe для вас является ключевой функцией, тогда, видимо, придётся воспользоваться Kubernetes-кластером.

> Liveness Probe

Liveness Probe проверяет, "живо" ли приложение внутри контейнера. Под этим можно понимать всё что угодно, начиная от парсинга логов с проверкой, что всё хорошо, и заканчивая отправкой запросов на специальные заранее подготовленные "ручки" приложения с анализом полученных ответов и получением текущего статуса приложения.

Liveness Probe — это название из Kubernetes. В DSC это можно реализовать с помощью директивы `healthcheck` в самом Docker-файле или путём передачи её в виде отдельных секций конфигурации в Docker-compose файле.

Мы посмотрим, как эта probe работает и в Dockere/Docker-Compose, и в Docker Swarm, потому что она работает в них немного по-разному. Напомним, что эта probe должна отключить от контейнера трафик и перезапустить сам контейнер. Сначала рассмотрим Docker/Docker-Compose.

Вот так будет выглядеть наш сервис `app.py`:

```
import time
import datetime
import redis
from flask import Flask

time.sleep(1)

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello world! I have been seen {} times.\n'.format(datetime.datetime.now())

@app.route('/healthcheck')
def check():
    return 'ok'
```

У этого приложения есть endpoint `healthcheck`, который всегда возвращает `'ok'`

Также посмотри на `Dockerfile`:

```
FROM python:3.8
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
# RUN apk add  --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
HEALTHCHECK --interval=2s --retries=2 \
  CMD curl -f 'http://localhost:5000/healthcheck'CMD ['flask', 'run']
```

Тут появилась директива `HEALTHCHECK` с флагами и командой "дергания ручки" у endpoint'а `healthcheck`. Если `curl` не отработает или вернёт какой-либо статус ошибки, то флаг `-f` сделает так, что `curl` завершится не с `exit code 0`, а с `exit code 1`. Если же команда в `HEALTHCHECK` завершается с кодом 1, то такой контейнер помечается как нездоровый (unhealthy).

Теперь давайте запустим контейнер и посмотрим, как он будет помечаться.

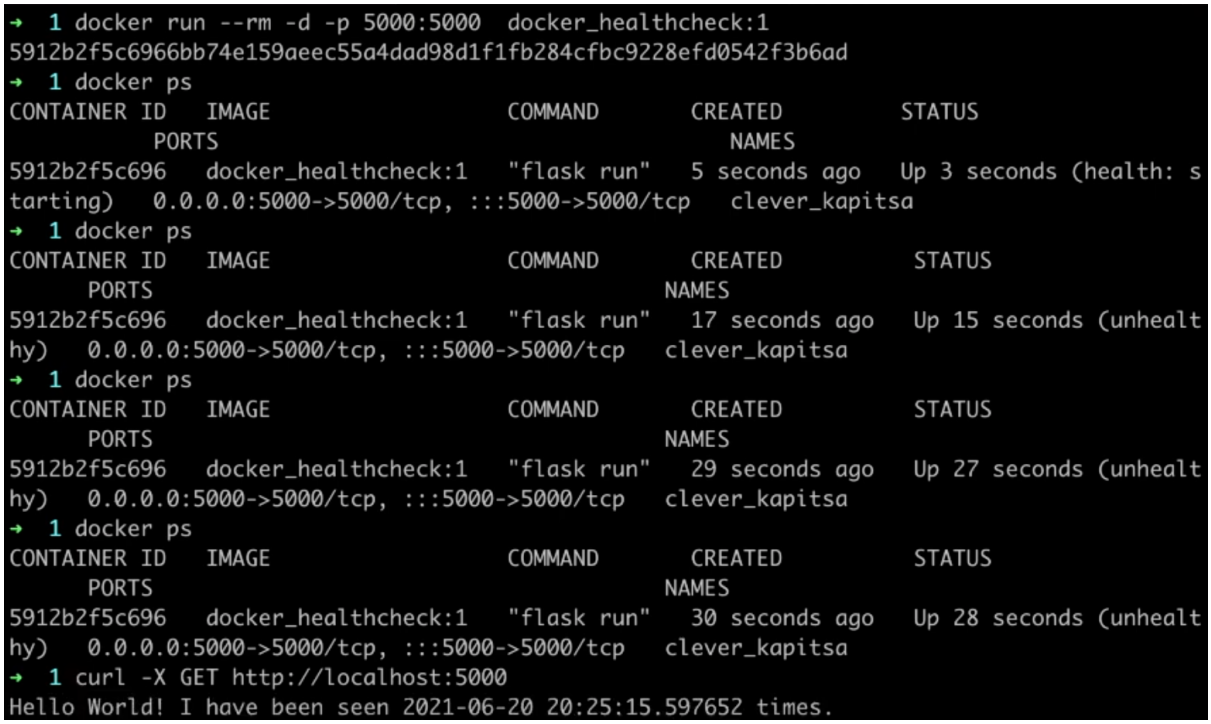
```
➜  0 vim Dockerfile
➜  0 docker run --rm -d docker_healthcheck:0
2b09c70d495a9b44323b5b704ae86b4fd1c4dd6561a26a35200ea335dbce5b22
➜  0 docker logs 2b09c70d495a9
 * Serving Flask app 'app.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.17.0.2:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [20/Jun/2021 20:18:53] "GET /healthcheck HTTP/1.1" 200 -
➜  0 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED          STATUS                    PORTS      NAMES
2b09c70d495a   docker_healthcheck:0   "flask run"   12 seconds ago   Up 10 seconds (healthy)   5000/tcp   rever
ent_noether
```

Видно, что он запустился, на него уже был отправлен один `healthcheck` запрос и он помечается как здоровый (healthy) в графе `STATUS`. Но что же произойдёт, если `healthcheck` вернёт код 1?

Для этого перепишем немного на `Dockerfile`, и изменим endpoint, который будем "дёргать" в healthchek'е. Просто напишем там несуществующий endpoint, а также немного изменим параметры healthcheck'а.

```
FROM python:3.8
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
# RUN apk add  --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
HEALTHCHECK --interval=10s --retries=1 \
  CMD curl -f 'http://localhost:5000/not_existing_endpoint'CMD ['flask', 'run']
```

Тогда получим вот такой результат:



Видим, что контейнер был health:starting, пока он запускался, а потом стал unhealthy. Тем не менее контейнер продолжает работу — он не был "убит" и даже отвечает на запросы. То есть кроме статуса контейнера ничего не изменилось. Получается, что директива `HEALTHCHECK` для Docker-контейнера сама по себе довольно бесполезная. Но если всё, что нужно в случае "нездоровости" контейнера, — это его перезапуск, то, добавив немного кода, можно добиться завершения контейнера при получении невалидного результата `HEALTHCHECK`'а.

Посмотрим на наше приложение. Оно немного изменилось: мы добавили обработку сигналов `SIGINT` и `SIGTERM` (сигнал `SIGKILL` мы поймать, увы, не можем).

```
import time
import datetime
import signal
import redis
from flask import Flask

app = Flask(__name__)

def signal_handler(sig, frame):
    print('Вы отправили сигнал о завершении')
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
signal.signal(signal.SIGTERM, signal_handler)
# signal.signal(signal.SIGKILL, signal_handler) # SIGKILL мы не сможем отловить

time.sleep(20)

@app.route('/')
def hello():
    return 'Hello world! I have been seen {} times.\n'.format(datetime.datetime.now())

@app.route('/healthcheck')
def check():
    return 'ok'
```

Также здесь мы немного переопределяем поведение `SIGINT` и `SIGTERM` на печать соответствующего сообщения и валидное завершение.

Также немного изменим Dockerfile:

```
FROM python:3.8
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
```

```
# RUN apk add  --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
HEALTHCHECK --interval=10s --retries=1 \
  CMD curl -f 'http://localhost:5000/not_existing_endpoint' || bash -c 'kill -s SIGTERM 1'# CMD curl -f 'http://localhost:5000/not_exi

CMD ['flask', 'run']
```

Теперь помимо проверки несуществующего endpoint'а, есть ещё и `SIGTERM`. Оператор `||` позволяет выполнить вторую команду (та, что справа), если первая (та, что слева) завершилась с ошибкой. Оператор `&&` работает наоборот: если первая команда выполнилась, то и вторая выполнится.

В нашем случае, если healthcheck не сработал, мы пошлём `SIGTERM`. Здесь также закомментировано некоторое расширение этого кода, которое, к сожалению, не будет работать на MacOS, а будет работать только на некоторых дистрибутивах Linux. В нём к gracefull shutdown нашего процесса добавляется ещё одна команда, которая будет выполнена только в том случае, если команда, завершающая процесс с помощью `SIGTERM`'а, не выполнилась. Логика в ней следующая: если процесс от `SIGTERM`'а завершился, то контейнер "убит" и следующая команда (после `&&`) не выполнится, однако если же `SIGTERM` не "прибил" контейнер по какой-то причине, хотя `SIGTERM` и вернул код 0, то контейнер останется работать и тогда выполнится следующая команда, которая инициирует `SIGKILL` процессу -1 (сигнал посылается всем процессам в системе — по сути это принудительное завершение контейнера).

Теперь посмотрим, как это сработает:

```
→  2 docker run -d -p 5000:5000  docker_healthcheck:2
e3977d958c30a8387e03e9097766f22d6597fa5569727a52c20c28e3c4c4c594
docker [K
→  2 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED         STATUS                            PORTS
                    NAMES
e3977d958c30   docker_healthcheck:2   "flask run"   5 seconds ago   Up 2 seconds (health: starting)   0.0.0.0:5000->5000/tcp,
:::5000->5000/tcp   nervous_panini
→  2 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED          STATUS                            PORTS
                    NAMES
e3977d958c30   docker_healthcheck:2   "flask run"   12 seconds ago   Up 10 seconds (health: starting)   0.0.0.0:5000->5000/tcp
, :::5000->5000/tcp   nervous_panini
→  2 docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
→  2 docker ps
→  2 dpcler [
→  2 docker ps -a
CONTAINER ID   IMAGE                  COMMAND                  CREATED             STATUS                          PORTS
NAMES
e3977d958c30   docker_healthcheck:2   "flask run"              About a minute ago   Exited (0) About a minute ago
nervous_panini
e69da4bf4c63   d105f40ed330           "flask run"              7 hours ago         Exited (2) 7 hours ago
gallant_golick
0ecc79baa775   45_web                 "flask run"              4 days ago          Exited (137) 4 days ago
45_web_1
c306330ccf3e   45_web                 "flask run"              4 days ago          Exited (137) 4 days ago
45_web_2
b43d234a7521   redis:latest           "docker-entrypoint.s…"   4 days ago          Exited (0) 4 days ago
45_redis_1
```

Видим, что некоторое время контейнер запускался, а потом был убит (нет ни одного запущенного контейнера). С помощью `docker ps -a` можно увидеть, что контейнер действительно был. Также можно посмотреть логи этого docker-контейнера:

```
→  2 docker logs e3977d958c30
 * Serving Flask app 'app.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
Вы отправили сигнал о завершении
```

Последняя строчка говорит нам о завершении.

Отлично. Теперь, чтобы автоматически запускать контейнер, необходимо в команду запуска всего лишь добавить параметр `--restart=always`

```
➜  2 docker run -d --restart=always  -p 5000:5000  docker_healthcheck:2
d09f57b3302b88143d48be680187e258726358dbad83e4fbb06671b8e9042600
docker p
➜  2 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED         STATUS                            PORTS
                          NAMES
d09f57b3302b   docker_healthcheck:2   "flask run"   4 seconds ago   Up 1 second (health: starting)   0.0.0.0:5000-
>5000/tcp, :::5000->5000/tcp   objective_maxwell
➜  2
➜  2 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED         STATUS                             PORTS
                          NAMES
d09f57b3302b   docker_healthcheck:2   "flask run"   9 seconds ago   Up 6 seconds (health: starting)   0.0.0.0:5000
->5000/tcp, :::5000->5000/tcp   objective_maxwell
➜  2 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED          STATUS                             PORTS
                          NAMES
d09f57b3302b   docker_healthcheck:2   "flask run"   12 seconds ago   Up 9 seconds (health: starting)   0.0.0.0:500
0->5000/tcp, :::5000->5000/tcp   objective_maxwell
➜  2 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED          STATUS                         PORTS     NAMES
d09f57b3302b   docker_healthcheck:2   "flask run"   15 seconds ago   Restarting (0) 1 second ago              objec
tive_maxwell
➜  2 docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED          STATUS                             PORTS
                          NAMES
d09f57b3302b   docker_healthcheck:2   "flask run"   26 seconds ago   Up 9 seconds (health: starting)   0.0.0.0:500
0->5000/tcp, :::5000->5000/tcp   objective_maxwell
```

Тогда мы увидим в `STATUS`, что контейнер сам перезапускается.

Swarm и из коробки правильно работает с healthcheck'ами. Если посреди работы приложение вдруг перестанет работать и перестанет работать контейнер, то Swarm автоматически перезапустит этот контейнер, если вы настроили политику перезапуска (и вам уже не нужно писать на Bash соответствующую логику). Как только контейнер пометится как unhealthy, Swarm не только сразу же запланирует его переподнятие, но и отключит трафик на этот контейнер. Поэтому если его поднятие произойдёт не сразу, то в этом нет ничего страшного.

> Startup Probe

Давайте рассмотрим пример. Мы задеплоим одну реплику и масштабируем на вторую. Параллельно с этим мы будем посылать на сервис трафик, который сначала будет идти только на одну существующую реплику, а потом пойдёт уже на две.

Docker-compose файл выглядит так:

```
version: "3.9"
services:
  web:
    image: web:20_sec_startup
    deploy:
      restart_policy:
        condition: on-failure
    ports:
      - "5000:5000"
```

Скрипт с load-тестом с использованием molotov будет следующим:

```
from molotov import scenario

_API = "http://localhost:5000"

@scenario(weight=100)
async def scenario_one(session):
    async with session.get(_API) as resp:
        assert resp.status == 200
```

Само приложение, которое перед запуском будет спать 20 секунд:

```
import time
import datetime
```

```
import redis
from flask import Flask

time.sleep(20)

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello world! I have been seen {} times.\n'.format(datetime.datetime.now())

@app.route('/healthcheck')
def check():
    return 'ok'
```

Предполагается, что в эти 20 секунд оно не сможет обрабатывать трафик, хотя запросы уже будут приходить на этот процесс.

Задеплоим сервис с помощью команды `docker stack deploy -c docker-compose.yml non-zero-downtime-upgrade`

```
➜  1 docker service ls
ID             NAME                                MODE         REPLICAS   IMAGE
      PORTS
f4k6r3cf6li4   non-zero-downtime-upgrade_web   replicated   1/1        web:20_sec_star
tup   *:5000->5000/tcp
```

```
➜  1 molotov
**** Molotov v2.1. Happy breaking! ****
Preparing 1 worker...
OK
SUCCESSES: 1159 | FAILURES: 0 | WORKERS: 1
```

Видим, что один контейнер запустился и все запросы на него обрабатываются корректно. Теперь с помощью команды `docker service update --replicas 2 non-zero-downtime-upgrade_web` масштабируем его (создадим ещё один контейнер).

```
➜  1 docker service update --replicas 2 non-zero-downtime-upgrade_web
"docker service update" requires exactly 1 argument.
See 'docker service update --help'.

Usage:  docker service update [OPTIONS] SERVICE

Update a service
➜  1 docker service update --replicas 2 non-zero-downtime-upgrade_web
non-zero-downtime-upgrade_web
overall progress: 2 out of 2 tasks
1/2: running
2/2: running
verify: Service converged
```

После того как контейнер "поднялся", как мы помним, наше приложение спит 20 секунд. В течение этого времени оно не может отвечать на запросы, и наш molotov показывает, что у нас пошли фейловые (непрошедшие) запросы:

```
➜  1 molotov
**** Molotov v2.1. Happy breaking! ****
Preparing 1 worker...
OK
SUCCESSES: 12675 | FAILURES: 2397 | WORKERS: 1
```

Так отработал Swarm-сервис, в котором не был настроен никакой healthcheck. Теперь посмотрим, что будет с тем же самым сервисом, если мы настроим для него healthcheck'и. В этот раз мы создадим его командой Swarm'а `create`,

поэтому теперь логику healthcheck'ов нужно будет прописать в Dockerfile.

```
FROM python:3.8
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
# RUN apk add  --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
HEALTHCHECK --interval=10s --retries=3 \
  CMD curl -f 'http://localhost:5000/not_existing_endpoint'

CMD ['flask', 'run']
```

В этот раз он будет отрабатывать суммарно в течение 30 секунд, что больше, чем наш "сон" после "подъёма" контейнера. Поэтому Swarm healthcheck не завершит контейнер раньше времени.

Создаём сервис командой `docker service create --name zero-downtime-upgrade-web web20_sec_startup_with_healthcheck` с помощью заранее собранного образа.

Контейнер поднимется, но, к сожалению, на запросы отвечать не будет — не потому, что не прошло 20 секунд, а потому, что мы не "пробросили" порты. Это можно сделать с помощью команды `docker service update --publish-add published=5000, target=5000 zero-downtime-upgrade-web`. Она завершит контейнер, обновит конфигурацию и запустит его снова уже с портами.

```
→  2 docker ps
CONTAINER ID   IMAGE                                  COMMAND       CREATED         STATUS
                    PORTS      NAMES
3d5bf0cc5073   web:20_sec_startup_with_healthcheck   "flask run"   9 seconds ago   Up 3 second
s (health: starting)   5000/tcp   zero-downtime-upgrade-web.1.tumn4odzcu3b9vy3xd1pir034
```

По истечении 20 секунд наш сервис начнёт откликаться. Теперь включим трафик и увидим, что он идёт и запросы обрабатываются верно. Теперь, если "поднять" вторую реплику, мы не увидим ни одного необработанного запроса:

```
→  2 molotov
**** Molotov v2.1. Happy breaking! ****
Preparing 1 worker...
OK
SUCCESSES: 35780 | FAILURES: 0 | WORKERS: 1
```

Мы видим, что до тех пор, пока healthcheck не начнёт отдавать положительный результат на новом контейнере, Swarm Service Discovery не зарегистрирует его как рабочий и не будет посылать на него трафик.

Если же сервис стартует не очень быстро, то может случиться так, что healthcheck'и не дождутся и вырубят приложение. На помощь может прийти флаг `--start-period` у директивы `HEALTHCHECK`.

При совместном запуске приложения (хотя это больше относится к Compose), вам может захотеться использовать какие-нибудь костыли/приколы, чтобы приложение не "падало" на старте. В таком случае можно использовать w8-for-it и другие обёртки.

> Советы

То, какие у приложения будут Probe и политики перезапусков, не входит в 12-factor — их можно спроектировать совсем по-разному. Поэтому здесь мы просто приведём тезисы, которые помогут задать правильный контекст, когда вы будете решать, как именно сделать у себя в проекте:

- Слишком жёсткие healthcheck'и, которые часто перезапускают приложения, как правило, спамят систему и генерируют много логов.
- Если в цепочке запросов виновным в "падении" оказался не ваш сервис, то, скорее всего, его перезапускать нет смысла, поскольку перезапуск будет логироваться и сбивать с толку, когда вы будете искать настоящую проблему.
- Добавление в healthcheck'и внешних зависимостей может показаться хорошей идеей, но в какой-то момент вы забудете про этот нюанс и будете страдать, разбираясь, в чём дело. Так делать не рекомендуем.

- Ситуация, когда база "моргнула" (некоторое время не отвечала) и все реплики тут же ушли на перезапуск, не выглядит профессиональной. Возможно, стоит сделать healthcheck'и более толерантными или вообще их не делать. Представьте, что вы отдаёте 99% информации из кэша, а 1% из базы. Нужно ли перезапускать контейнеры из-за того, что база "моргнула"?
- Возможно, на ранней стадии развития проекта/сервиса лучше избавиться от того, что может скрыть проблему. А вот на стадии зрелости перезапусками решать проблему отсутствия мониторинга и экстренных оповещений уже не стоит.

> Дополнительные материалы

- Про ключевые концепции Docker Swarm
- Про Docker Healthcheck и Kubernetes Probe
- Про Startup приложения
- Про Healthcheck
- Bash Healthcheck, который "убьёт" `kill pid=1`
- Про Rolling Update в Swarm