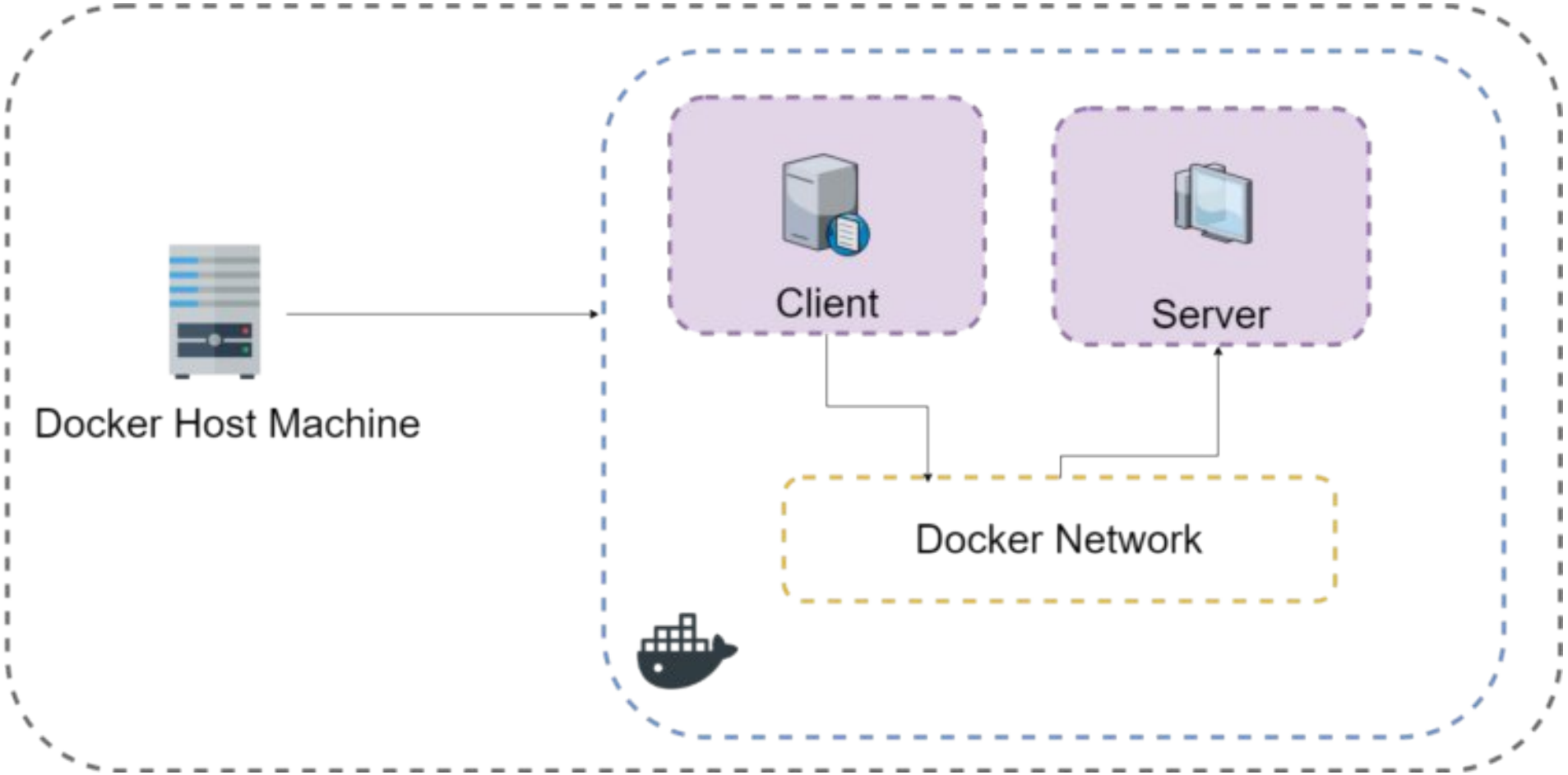
Масштабируемость и отказоустойчивость Docker-кластеров

Балашов Антон

План лекции

- Docker – сетевое взаимодействие
- Масштабируемость и отказоусточивость приложений
- Кластеры Docker
- Балансировка нагрузки приложений в кластере

Docker – сетевое взаимодействие



Docker – сетевое взаимодействие

CNM
Design spec

Libnetwork
Implements CNM

Plugins
Network topology

Docker Network Drivers

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
6df045e07b11        bridge              bridge              local
9d45c14e604a        host                host                local
b285e48f2c94        none                null                local
```

- **Bridge network** при запуске Docker автоматически создается сеть типа мост по умолчанию. Недавно запущенные контейнеры будут автоматически подключаться к нему.

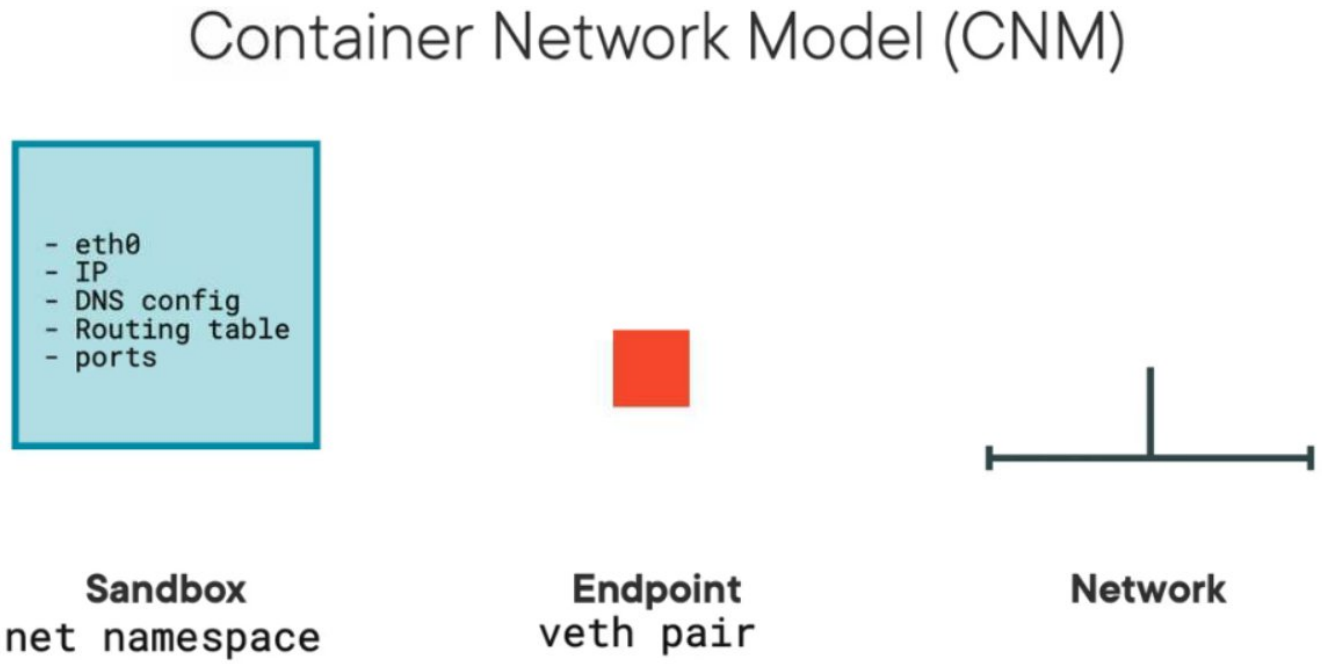
- **Host network :** удаляет сетевую изоляцию между контейнером и хостом Docker и напрямую использует сеть хоста. Если вы запускаете контейнер, который привязывается к порту 80, и вы используете хост-сеть, приложение контейнера доступно через порт 80 по IP-адресу хоста. Означает, что вы не сможете запускать несколько веб-контейнеров на одном хосте, на одном и том же порту, так как порт теперь является общим для всех контейнеров в сети хоста.

- **None network :** в сети такого типа контейнеры не подключены ни к одной сети и не имеют доступа к внешней сети или другим контейнерам. Итак, эта сеть используется, когда вы хотите полностью отключить сетевой стек в контейнере.

- **Overlay network :** Создает внутреннюю частную сеть, которая охватывает все узлы, участвующие в кластере. Таким образом, оверлейные сети облегчают обмен данными между хост-машиной и автономным контейнером или между двумя автономными контейнерами на разных демонах Docker.

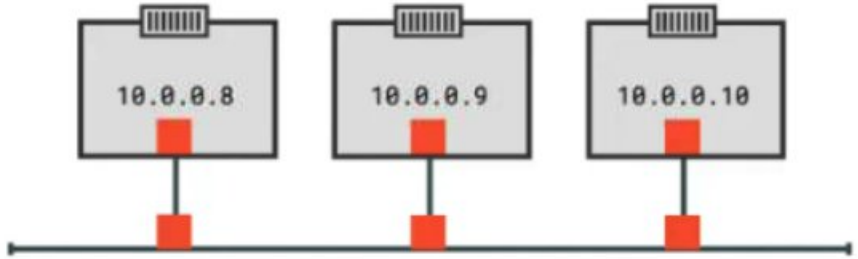
- **Macvlan network :** Некоторые приложения, особенно устаревшие приложения, отслеживающие сетевой трафик, ожидают прямого подключения к физической сети. В такой ситуации вы можете использовать сетевой драйвер Macvlan для назначения MAC-адреса виртуальному сетевому интерфейсу каждого контейнера, что делает его физическим сетевым интерфейсом, напрямую подключенным к физической сети.

Docker – Bridge Network



Veth(Virtual Ethernet) — использование сетевого стека хост системы. На хосте создается мост который подключается к одному из физических интерфейсов. При запуске контейнера с таким типом сети, на хосте создается специальный виртуальный интерфейс коммутируемый к мосту. Этот виртуальный интерфейс (хост системы) фактически и использует контейнер для взаимодействия с внешней средой.

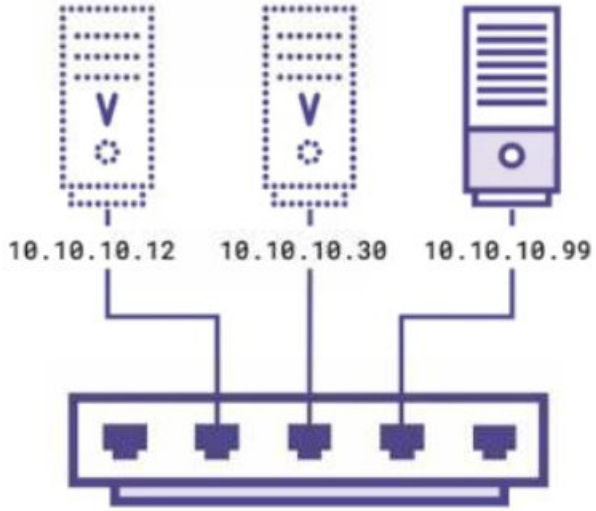
Docker – Bridge Network



802.11b bridge (a.k.a. layer 2 switch)

IP range: 10.0.0.0/24

...



802.11b bridge (a.k.a. layer 2 switch)

IP range: 10.10.10.0/24

...

Пример работы контейнеров с сетью

```
root@LAPTOP-1UADVU1A:~# docker network create --driver=bridge test-bridge
87b32ac2d822b46426dd9c52129154fff246d4eefb0eb417a64a4156a1d87dd2
root@LAPTOP-1UADVU1A:~# docker run -it -d --name A1 --network test-bridge alpine ash
ad757a8d94266d1d8787a2914ccd158688a9a1b9a4e040f73682265eb54bf5eb
root@LAPTOP-1UADVU1A:~# docker run -it -d --name A2 --network test-bridge alpine ash
3406817db9996bebafa95ea2208e8766f2e49e942f2ef7cf7ce335582c2db650
root@LAPTOP-1UADVU1A:~# docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS          PORTS     NAMES
3406817db999   alpine    "ash"     11 seconds ago   Up 10 seconds             A2
ad757a8d9426   alpine    "ash"     18 seconds ago   Up 16 seconds             A1
root@LAPTOP-1UADVU1A:~# docker container attach A1
/ # ping -c 2 A2
PING A2 (172.24.0.3): 56 data bytes
64 bytes from 172.24.0.3: seq=0 ttl=64 time=0.258 ms
64 bytes from 172.24.0.3: seq=1 ttl=64 time=0.157 ms

--- A2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.157/0.207/0.258 ms
/ # exit
```

Пример работы контейнеров с сетью

```
root@LAPTOP-1UADVU1A:~# docker inspect test-bridge
[
    {
        "Name": "test-bridge",
        "Id": "87b32ac2d822b46426dd9c52129154fff246d4eefb0eb417a64a4156a1d87dd2",
        "Created": "2022-08-31T11:08:54.712241987Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.24.0.0/16",
                    "Gateway": "172.24.0.1"
                }
            ]
        },
```

Масштабируемость

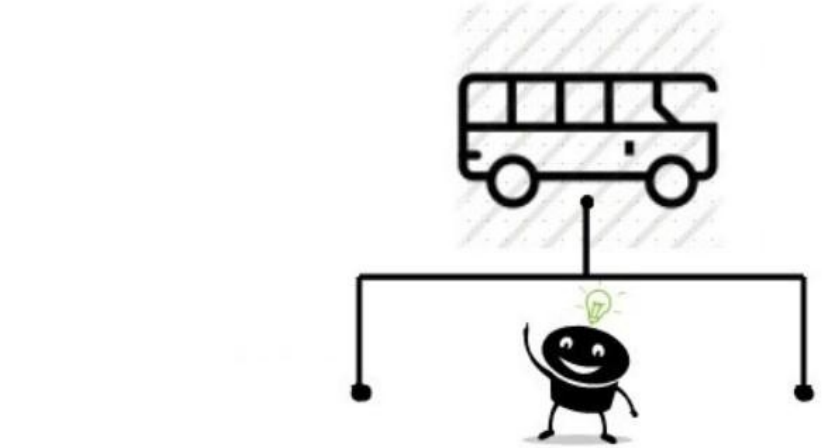
Масштабируемость при облачных вычислениях — это возможность быстро и без труда увеличить или уменьшить размер либо мощность ИТ-решения или ресурса. В то время как термин «масштабируемость» может означать способность любой системы справиться с растущим объемом работы, в контексте горизонтального и вертикального масштабирования речь часто идет о базах данных и больших объемах данных.

Масштабируемость позволяет адаптироваться к огромным объемам разнообразных данных и управлять ими, изменяя объемы данных и шаблоны рабочих нагрузок, которые создаются в облаке, на мобильных устройствах, в социальных сетях и источниках больших данных.

Масштабируемость

- **Вертикальное масштабирование**: наращивание или снижение вычислительной мощности или ресурсов по мере необходимости. Для автоматической адаптации к требованиям рабочих нагрузок применяется либо изменение уровней производительности, либо эластичные пулы ресурсов.

- **Горизонтальное масштабирование**: добавление дополнительных мощностей или разделение одного узла на узлы меньшего размера с использованием сегментирования. Обеспечивает ускоренное и более удобное управление приложением на серверах.



Вертикальное масштабирование

Вертикальное масштабирование используется, когда необходимо быстро реагировать на проблемы с производительностью, которые нельзя решить с помощью классической методики оптимизации кода. Вертикальное масштабирование помогает справиться с пиками в рабочих нагрузках, когда текущий уровень производительности не может удовлетворить все требования. Вертикальное увеличение масштаба позволяет добавлять дополнительные ресурсы, чтобы легко адаптироваться к пиковым рабочим нагрузкам. Затем, если ресурсы больше не нужны, можно выполнить вертикальное уменьшение масштаба, чтобы вернуться к исходному состоянию и сократить затраты.

Вертикальное масштабирование выполняется в следующих случаях:

- Рабочие нагрузки достигают определенного ограничения производительности, например ограничения, касающегося ЦП или операций ввода-вывода.
- Нужно быстро реагировать, чтобы устранить проблемы с производительностью, которые нельзя решить путем классической оптимизации базы данных.
- Требуется решение, позволяющее изменять уровни служб, чтобы адаптироваться к изменению требований к задержке.

Горизонтальное масштабирование

Горизонтальное масштабирование применяют когда не удается получить достаточно ресурсов для рабочих нагрузок даже на самых высоких уровнях производительности. При горизонтальном масштабировании копии приложения распределяются между серверами. Масштаб каждого сервера можно вертикально увеличивать или уменьшать по отдельности.

Горизонтальное масштабирование выполняется в следующих случаях:

- Географически распределенные приложения, каждое из которых должно работать в своем регионе.
- Глобальный сценарий сегментирования (например, балансировка нагрузки) с большим количеством географически распределенных клиентов.
- Если ограничения производительности превышаются даже на самых высоких уровнях производительности службы.

Отказоустойчивость приложений

Отказоустойчивость — свойство технической системы сохранять свою работоспособность после отказа одной или нескольких её составных частей. **Отказоустойчивость** определяется количеством единичных отказов составных частей (элементов) системы, после наступления которых сохраняется работоспособность системы в целом. Базовый уровень **отказоустойчивости** подразумевает защиту от отказа одного любого элемента. Поэтому основной способ повышения **отказоустойчивости** это избыточность.

Реализация отказоустойчивости:

- DR (disaster recovery)- копия системы может быть поднята в другом ЦОД. Существует ряд требований к реализации такого сценария.

Программно реализованная отказоустойчивость для каждого из компонент и их взаимодействия

Программно реализованная отказоустойчивость для каждого из компонент и их взаимодействия

- **Low level fault tolerant services -** система должна состоять из независимых подсистем, каждая из которых должна быть отказоустойчивой.
- **Single point of failure -** Избежание архитектуры, где вся система рушиться при падении одного из компонент. Достичь это можно либо используя принцип избыточности, либо делать компоненты по возможности независимыми, чтобы при падении одного из компонент, переставала работать только часть функциональности, а остальные части системы продолжали работать.
- **Избыточность (Redundancy) - в** системе существует несколько копий компонент. И при падении одного из них система продолжает работать. При таком подходе проектирования можно выделить следующие стратегии:

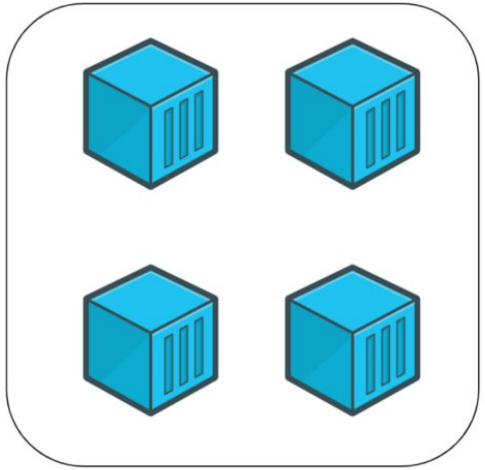
Стратегии избыточности

- **Active-Active-** система одновременно работает с двумя идентичными компонентами. Запрос на обработку поступает одновременно на два компонента, одновременно обрабатывающие запрос. Если один из таких компонентов рушится, то вся работа автоматически переходит только на второй компонент. В качестве минусов можно выделить, увеличенное количество трафика и время на его обработку, а так же дополнительная серверная инфраструктура в постоянно рабочем режиме потребления ресурсов.

- **Active-Passive-** только один постоянно работающий компонент, в случае падения автоматически поднимается второй, восстанавливает состояние и берет на себя всю работу. Минусом является время восстановления состояния
- **Балансировка нагрузки (Load Balancing) -** несколько идентичных компонентов, между которыми распределяется нагрузка по заданному правилу. В отличие от вышеописанной active-active стратегии, здесь каждую задачу выполняет только один компонент. Данный механизм идеально подходит для stateless компонент, иначе для отказоустойчивости вам постоянно придется синхронизировать состояние. Например в случае веб-серверов — делать репликацию сессии. В данном решении очень важно иметь как минимум N+1 redundancy, т.е. если для пиковых нагрузок вам необходимо N работающих на всю катушку компонент, то в вашей системе должно присутствовать N+1 таких компонент, так как иначе, если у вас упадет один из элементов

Кластеры Docker

Docker



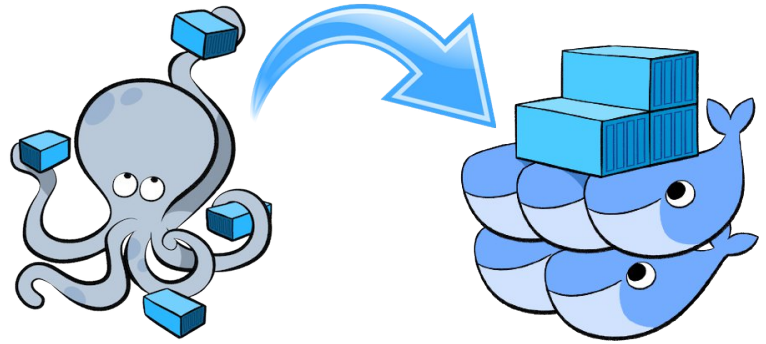
Docker-Compose

Docker Compose — это инструментальное средство, вход состав Docker. Оно предназначено для решения задач, связанных с развёртыванием проектов, состоящих из нескольких контейнеров.

Разница между Docker и Docker Compose

- Docker применяется для управления отдельными контейнерами (сервисами), из которых состоит приложение.
- Docker Compose используется для одновременного управления несколькими контейнерами, входящими в состав приложения. Этот инструмент предлагает те же возможности, что и Docker, но позволяет работать с более сложными приложениями.

Возможности Docker Compose



Несколько изолированных сред на одном хосте

- Возможно создавать несколько сред окружения на одном хосте, используя название проекта в различных контекстах:
- Создать на одном хосте несколько копий одного и того же окружения.
- Изолировать разные проекты на хосте, которые могут использовать сервисы с одинаковыми названиями.

Пересоздаются только измененные контейнеры

Compose кэширует конфигурацию, которая была использована для создания контейнера. Если перезапустить контейнер без изменений, то будут использованы существующие файлы. Такое повторное использование обеспечивает быстрое внесение изменений в проект.

Возможности Docker Compose

Команды запуска

- Создание контейнера для служб и запуск среды в фоновом режиме

docker-compose up -d

- проверить активность среды:

docker-compose ps

- Посмотреть логи:

docker-compose logs

- Приостановить работу контейнерной среды без изменения текущего состояния контейнеров:

docker-compose pause

- Возобновить работу после паузы:

docker-compose unpause

- Остановить выполнение контейнера без удаления связанных с ним данных:

docker-compose stop

- Удалить контейнеры, сети и тома, которые связаны с контейнерной средой:

docker-compose down

- Удалить образ, из которого собирается среда:

docker image rm name:tag

Описание среды docker-compose

Файл docker-compose.yml должен начинаться с тега версии.

version: "3.2"

Следует учитывать, что docker-composes работает с сервисами. 1 сервис = 1 контейнер.

Сервисом может быть клиент, сервер, сервер баз данных...

Раздел, в котором будут описаны сервисы, начинается с ‘services’.

services:

Название сервиса

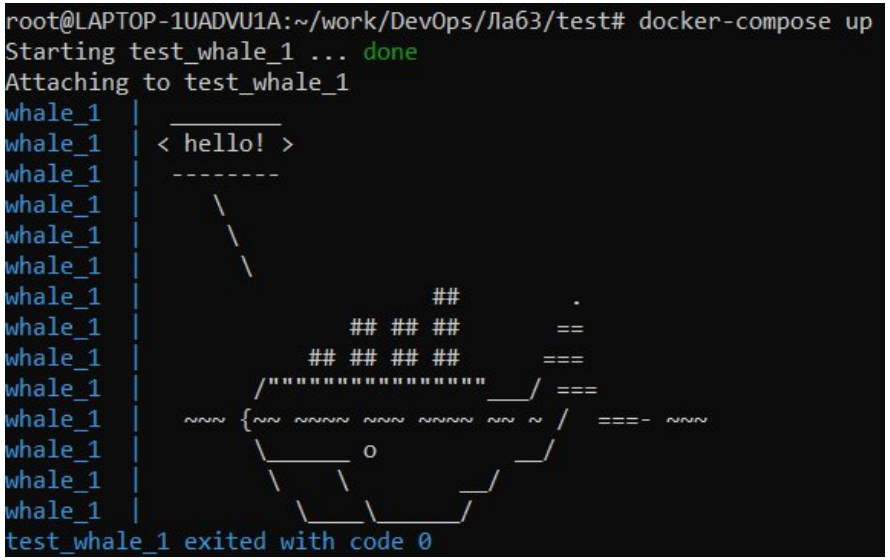
  whale:

Docker образ, который мы хотим запустить в контейнере

    image: docker/whalesay

Команда, которую нужно запустить после скачивания образа.

    command: ["cowsay", "hello!"]



YAML

Основные цели:

- быть понятным человеку;
- поддерживать структуры данных, родные для языков программирования;
- быть переносимым между языками программирования;
- использовать цельную модель данных для поддержки обычного инструментария;
- поддерживать потоковую обработку;
- быть выразительным и расширяемым;
- быть лёгким в реализации и использовании.

YAML Синтаксис

Отступы

- В YAML для разделения информации очень важны отступы. Нужно помнить, что используются только пробелы, табуляция не допускаются.

  При отсутствии отступа перед первым объявлением YAML поймет, что это корень (уровень 0) вашего файла.

Ключ/Значение

- Как и в JSON/JS, в YAML есть синтаксис ключ/значение, значением может быть как число, так и строка, так и список

Списки

- Синтаксис JSON: массив строк

  people: [‘Anne’, ‘John’, ‘Max’]

- Синтаксис дефиса

```
people:
  - Anne
  - John
  - Max
```

Пример среды docker-compose

```
root@LAPTOP-1UADVU1A:~/work/DevOps/Лаб3/python-ui-database# tree -a .
.
├── .env
├── app
│   ├── Dockerfile
│   ├── app.py
│   └── requirements.txt
├── db
│   └── init.sql
└── docker-compose.yml
```

Способы конфигурирования среды

- Переменные окружения

```
web:
    image: «webapp:${TAG}»
```

- .env файл в директории с docekr-compose.yml

```
cat .env
MYSQL_USER=root
MYSQL_ROOT_PASSWORD=root
MYSQL_HOST=db
MYSQL_PORT=3306
```

Способы конфигурирования среды

- environment - тэг

```
web:
  environment:
    PMA_HOST: db
    PMA_PORT: 3306
    PMA_USER: ${MYSQL_USER}
    PMA_PASSWORD: ${MYSQL_ROOT_PASSWORD}
```

- env_file - тэг

```
web:

  env_file:

    - web-variables.env
```

- env_file - ключ

```
docker compose —env-file ./config/.env.dev up
```

Docker-compose.yml

```
version: "3"
services:
  app:
    build: ./app  # контекст для сборки, по-умолчанию ищет Dockerfile в этой директории
    links: # связь контейнеров друг с другом, один из способов организации сетевого взаимодействия
      - db
    ports:  # проброс портов <Host>:<Container>
      - "8090:5000"

  db:
    image: mysql:5.7
    ports:
      - "32000:3306"
    volumes: # привязка каталога на диске хост машины к контейнеру
      - ./db:/docker-entrypoint-initdb.d/:ro

  phpmyadmin:
    image: phpmyadmin/phpmyadmin:latest
    restart: always
    links:
      - db
    environment:
      PMA_HOST: db
      PMA_PORT: 3306
      PMA_USER: ${MYSQL_USER}
      PMA_PASSWORD: ${MYSQL_ROOT_PASSWORD}
    ports:
      - "8080:80"
```

Балансировка нагрузки приложений в кластере

Балансировка нагрузки подразумевает эффективное распределение входящего сетевого трафика между группой бэкенд-серверов. Задача же регулятора — распределить нагрузку между несколькими установленными бэкенд-серверами.

Балансировка нагрузки помогает масштабировать приложение, справляясь со скачками трафика без увеличения расходов на облако. Она также помогает устранить проблему единой точки отказа. Поскольку нагрузка является распределённой, то в случае сбоя одного из серверов — сервис всё равно продолжит работу.

- Nginx — это высокопроизводительный веб-сервер, который также может использоваться в качестве регулятора нагрузки — это процесс распределения веб-трафика между несколькими серверами с помощью Nginx.Он гарантирует, что ни один сервер не будет перегружен и что все запросы будут обработаны своевременно. Nginx использует различные алгоритмы для определения оптимального распределения трафика, а также может быть настроен для обеспечения отказоустойчивости в случае выхода из строя одного из серверов.

- HAProxy – серверное программное обеспечение для обеспечения высокой доступности и балансировки нагрузки для TCP и HTTP приложений, методом распределения входящих запросов на несколько серверов

Настройка балансировки нагрузки nginx

```
http {
  upstream backend {
    server nworker1:80;
    server nworker2:80;
    server nworker3:80;
  }

  server {
    listen 8080 default_server;
    listen [::]:8080 default_server;
    server_name localhost;
    proxy_read_timeout 5m;
    location / {
      proxy_pass http://backend;
    }
  }
}
```

Nginx. Методы балансировки нагрузки

Round Robin

метод балансировки нагрузки, при котором каждому серверу в кластере предоставляется равная возможность обрабатывать запросы. Этот метод часто используется в веб-серверах, где каждый запрос сервера равномерно распределяется между серверами.

Мощность распределяется поочерёдно, что означает, что у каждого сервера будет своё время для выполнения запроса. Например, если у вас есть три вышестоящих сервера, A, B и C, то балансировщик нагрузки сначала распределит нагрузку на A, затем на B и, наконец, на C, прежде чем перераспределить нагрузку на A. Этот метод довольно прост, но имеет некоторую долю ограничений.

Одно из ограничений заключается в том, что некоторые серверы будут простаивать просто потому, что они будут ждать своей очереди. В нашем примере, если A получит задание и выполнит его за секунду, это будет означать, что он будет простаивать до следующего задания. По умолчанию для распределения нагрузки между серверами Nginx использует именно метод round robin.

Nginx. Методы балансировки нагрузки

Round Robin с добавлением веса

Чтобы решить проблему простоя серверов, мы можем использовать server weights (серверные веса), чтобы указать Nginx, какие серверы должны иметь наибольший приоритет. Weighted Round Robin — *один из самых популярных методов балансировки нагрузки*, используемых сегодня.

Этот метод предполагает присвоение веса каждому серверу, а затем распределение трафика между серверами на основе этих весов. Это гарантирует, что серверы с большей пропускной способностью получат больше трафика, и поможет предотвратить перегрузку какого-либо из серверов.

Этот метод часто используется в сочетании с другими методами, такими как Session Persistence, для обеспечения равномерного распределения мощностей на все серверы. Сервер приложения *с наибольшим параметром веса будет иметь приоритет (больше трафика)* по сравнению с сервером с наименьшим числом (весом).

```
upstream app{
    server 10.2.0.100 weight=5;
    server 10.2.0.101 weight=3;
    server 10.2.0.102 weight=1;
 }
```

Nginx. Методы балансировки нагрузки

Least Connection

Метод наименьшего числа соединений (Least Connection) — это популярная техника, используемая для равномерного распределения рабочей нагрузки между несколькими серверами. Метод работает путём маршрутизации каждого нового запроса на соединение — на сервер с наименьшим количеством активных соединений. Это гарантирует, что все серверы используются одинаково и ни один из них не перегружен.

```
upstream app{
    least_conn;
    server 10.2.0.100;
    server 10.2.0.101;
    server 10.2.0.102;
  }
```

Least Connection с добавлением веса

Данный метод используется для распределения рабочей нагрузки между несколькими вычислительными ресурсами (такими как серверы) с целью оптимизации производительности и минимизации времени отклика. Этот метод учитывает количество активных соединений на каждом сервере и присваивает соответствующие веса. Целью является распределение рабочей нагрузки таким образом, чтобы сбалансировать нагрузку и минимизировать время отклика.

Nginx. Методы балансировки нагрузки

IP Hash

Метод балансировки IP Hash использует алгоритм хэширования для определения того, какой сервер должен получить каждый из входящих пакетов. Это полезно, когда за одним IP-адресом находится несколько серверов, и вы хотите убедиться, что каждый пакет с IP-адреса клиента направляется на один и тот же сервер. Он берёт IP-адрес источника и IP-адрес назначения и создаёт уникальный хэш-ключ. Затем он используется для распределения клиента между определёнными серверами.

Это очень важный момент в случае развёртывания canary. Это позволяет нам, разработчикам, выпускать изменения для отдельной части пользователей, чтобы они могли всё протестировать и предоставить отзывы, прежде чем отправлять их в релиз.

Преимущество этого подхода в том, что он может обеспечить более высокую производительность, чем другие методы, такие как round-robin.

URL Hash

Регулировка нагрузки URL Hash также использует алгоритм хэширования для определения того, какой сервер получит каждый из запросов на основе URL.

Он также похож на метод балансировки IP Hash, но разница в том, что мы хэшируем конкретные URL, а не IP. Это гарантирует, что все запросы равномерно распределяются между серверами, обеспечивая лучшую производительность и надёжность.

HaProxy

Алгоритмы балансировки нагрузки.

- Указание серверов в разделе бэкенда позволяет HAProxy использовать эти серверы для балансировки нагрузки в соответствии с алгоритмом циклического перебора, когда это возможно.

- Алгоритмы балансировки используются для определения того, на какой сервер в бэкенде передается каждое соединение. Вот некоторые из полезных опций:

- **Roundrobin:** каждый сервер используется по очереди в соответствии со своим весом. Это самый плавный и честный алгоритм, когда время обработки серверами остается равномерно распределенным. Этот алгоритм является динамическим, что позволяет регулировать вес сервера на лету.
- **Leastconn:** выбирается сервер с наименьшим количеством соединений. Циклический перебор выполняется между серверами с одинаковой нагрузкой. Использование этого алгоритма рекомендуется для длинных сеансов, таких как LDAP, SQL, TSE и т. д., но он не очень подходит для коротких сеансов, таких как HTTP.
- **First:** первый сервер с доступными слотами для подключения получает соединение. Серверы выбираются от самого низкого числового идентификатора до самого высокого, который по умолчанию соответствует положению сервера в ферме. Как только сервер достигает значения maxconn, используется следующий сервер.
- **Source:** IP-адрес источника хешируется и делится на общий вес запущенных серверов, чтобы определить, какой сервер будет получать запрос. Таким образом, один и тот же IP-адрес клиента будет всегда доставаться одному и тому же серверу, в то время как серверы остаются неизменными.

HaProxy

```
defaults
    log global # системный лог
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
    stats uri /status
frontend balancer  # фронтэнд – наш балансировщик
    bind 0.0.0.0:80
    default_backend web_backends
backend web_backends   # фронтэнд – бекенды, воркеры
    balance roundrobin
    server web1 worker1:80
    server web2 worker2:80
```