KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект >4 урок > Docker Runtime + Docker-compose

> Оглавление

> Введение

Сегодня поговорим про Docker, Docker compose и Docker swarm.

Сейчас в контейнерах находится почти всё. Это удобно с точки зрения:

- Пакетирования;
- Изоляции;
- Использования (есть все sdk для докера, удобный CLI);
- Масштабирования — swarm и Kubernetes.

> Пререквизиты

Предполагается, что вы уже знаете:

- Что такое образ, контейнер и registry образов;
- Что образ собирается из докерфайла, а контейнер получается, когда мы запускаем образ с какой-то командой;
- Про флаг `p`, пробрасывающий порты на хост, делающий контейнер доступным извне, и флаг `v`, позволяющий пробрасывать директории с хоста в контейнер;
- Что можно запускать в `detached` режиме.

Сегодня сфокусируемся на трёх вещах:

- Как получить контейнер, работающий максимально близко к паттерну 12 factor apps на Docker compose и Docker swarm;
- Docker compose;
- Docker swarm.

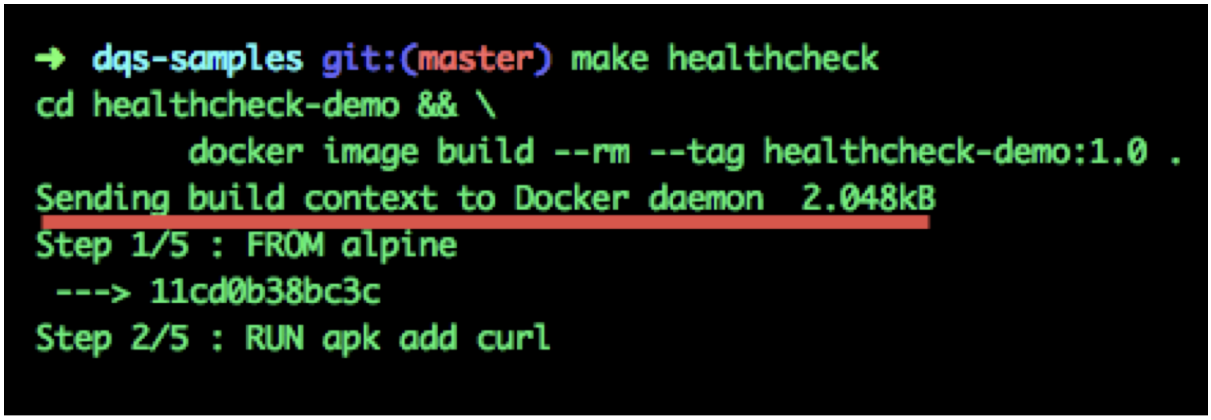
> Нюансы сборки

> Сборка

По-хорошему все сборки должны выполняться очень быстро, чтобы не тормозить процесс разработки.

> Контекст

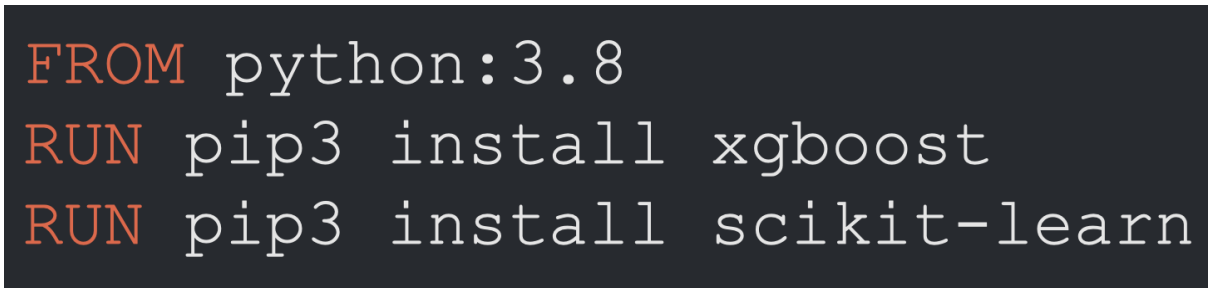
Есть контекст сборки (содержимое папки, в которой используется команда `build`), который загружается в демон при создании образа.



Отправка контекста в демон Docker

Если докерфайл написан неправильно (вы копируете всё без разбора), то размер станет очень большим. Лечится при помощи `--compress`. Это ускорит сборку.

> Повторяющиеся инструкции



Например, pip install написано много раз

Объединение в одну команду даст небольшой выигрыш по времени скачивания и по размеру. Если в `pip install` было много установок пакетов и ничего более, то выигрыш будет небольшим.

Плюсы:

1. В кейсах, когда ставится много зависимостей, может дать ощутимую экономию места;
2. Можно спрятать чувствительные файлы.

Минусы:

1. Мало слоёв — неэффективное кэширование;
2. Больше нет возможности скачать слои параллельно.

> Удаление файлов

Если вы что-то копировали в контейнер, процессили и потом удалили, в слоях, где вы процессили файлы, они останутся даже несмотря на то, что в следующих слоях вы их удалили.

```
FROM python:3.8

COPY dependency dependency #-> файл создается в layer1
RUN echo                   #-> layer2
RUN rm -rf dependency      #-> файл удаляется в layer3,
                               Но не удаляется из layer1
```

Чтобы этого не происходило, необходимо опять же объединять команды под одной инструкцией. Если объединить команды нет возможности или вы уже собрали такой образ, то можно сквошить образ, используя флаг `--squash`.

Есть частный случай этого кейса: при работе с компилируемыми языками (например Scala).

> Multistage building

Для сборки исполняемого файла нужен один сет зависимостей, но для его работы они уже не нужны.

Multistage building позволяет описать в докерфайле несколько этапов сборки, которые будут выполняться по умолчанию один за другим. Образы промежуточных и первых стейджей не будут сохраняться. Каждый следующий стейдж может забирать артефакты у предыдущего.

```
FROM openjdk:some_tag AS builder  # Образ, содержащий все
WORKDIR /app
RUN sbt compile
RUN sbt package  # Сгенерируется project.jar

FROM openjre:some_tag  # Образ, содержащий минимум для рантайма
WORKDIR /root/
COPY --from=builder /app/project.jar .
CMD ["java", "project.jar"]
```

Из первого стейджа копируется артефакт /app/project.jar

Можно описывать все стейджи в одном докерфайле и собирать образы конкретного стейджа, передавая аргумент `--target` и название стейджа.

```
FROM openjdk:some_tag AS compiler  # Образ, содержащий все
CMD ["sbt", "compile"]

FROM openjre:some_tag AS test  # Образ, содержащий минимум для рантайма
CMD ["sbt", "test"]
```

Запуск определенного стейджа:

```
docker build --target compiler -t my:image
```

В результате сборки стейдж test выполняться не будет

Вам могло прийти в голову сделать из Multistaging пайплайн ML, но процесс сборки докера нельзя запустить в демон режиме.

> Builder паттерн и кэширование

Builder паттерн подразумевает вынесение статичных частей сборок в отдельные базовые образы, содержащие статичные зависимости. При сборке новые образы будут использовать уже заготовленные базовые.

Таким образом, сборка сводится к скачиванию базового образа к месту сборки и созданию дополнительных слоёв с новым кодом.

> Иные нюансы

- Чувствительным данным не место в образе.
- Для оптимального использования закодированных слоев важен правильный порядок команд.

- Все зависимости нужно указывать явно, чтобы не выстрелить себе в ногу.

> Docker Runtime

> Сравнение volumes и bind mount

Bind mount	Named volumes
• Очень быстро • Данные прямо на хосте • Изменяют права файлов	• Быстро • Данные требуется копировать с помощью контейнера • Больше порядка

> Пользователи внутри контейнера

Желательно внутри контейнера иметь не `root` пользователя:

- По умолчанию пользователь внутри контейнера — `root` (`uid=0`, `guid=0`);
- Пользователя можно поменять в инструкции Dockerfile через `USER` директиву;
- Пользователя можно поменять при запуске `-user $(uid):$(gid)`.

Но даже если контейнер работает не от рута, то всё равно можно сделать:

```
docker exec -it –user 0:0 non_root_container
```

> Процессы внутри контейнеров

Процессы докера видны из `ps aux`, но внутри контейнера всё изолировано (кроме MacOS), и есть `init` процесс из 3-й лекции.

Запуск контейнера == запуск `init` процесса. Подчеркиваем, контейнер считается запущенным, как только запускается внутренний `init` процесс. То есть если ваш `init` процесс перед стартом что-то скачивает или создаёт сабпроцессы, то Docker, очевидно и справедливо, считает, что процесс уже запущен. Завершение `init` эквивалентно смерти контейнера. Смерть другого процесса контейнер не убивает.

Docker игнорирует `SIGKILL` для `init` процесса.

На этом моменте в голову обычно приходит гениальная идея — сделать так, чтобы `init` процесс жил.

Контраргументы:

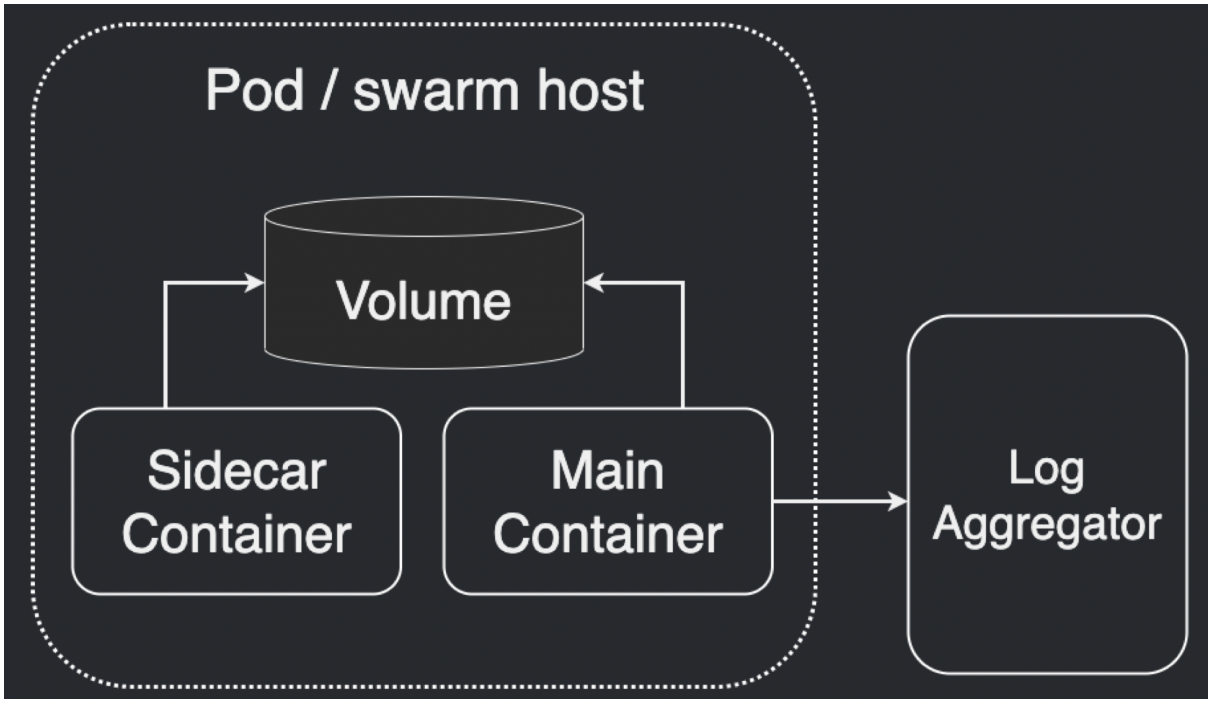
- Логи. В некоторых случаях не получится получать логи из `stdout` подпроцессов. Появляется необходимость разделять логи подроцессов.
- Сокрытие проблем. Если с НЕ `init` процессами в контейнерах возникнут проблемы, то контейнер продолжит работу, и мы не узнаем о проблемах.
- Проброс сигналов. Могут быть проблемы с передачей сигналов подпроцессам.
- Сложность. Усложняется сборка, тестирование и масштабирование.

> Несколько процессов в одном контейнере

Если у вас экстраординарный случай (два процесса имеют высокую связность и нужно всё поместить в один контейнер), тогда можно:

- Использовать для запуска контейнера флаг `-init`. Тогда процесс приложения запустится как подпроцесс, а в качестве `init` процесса запустится `tini` процесс.
- Использовать pm2 и другие высокоуровневые оркестраторы процессов. Не докер (Docker in Docker — это слишком) и не systemd, потому что systemd в контейнере — это очень плохая практика.

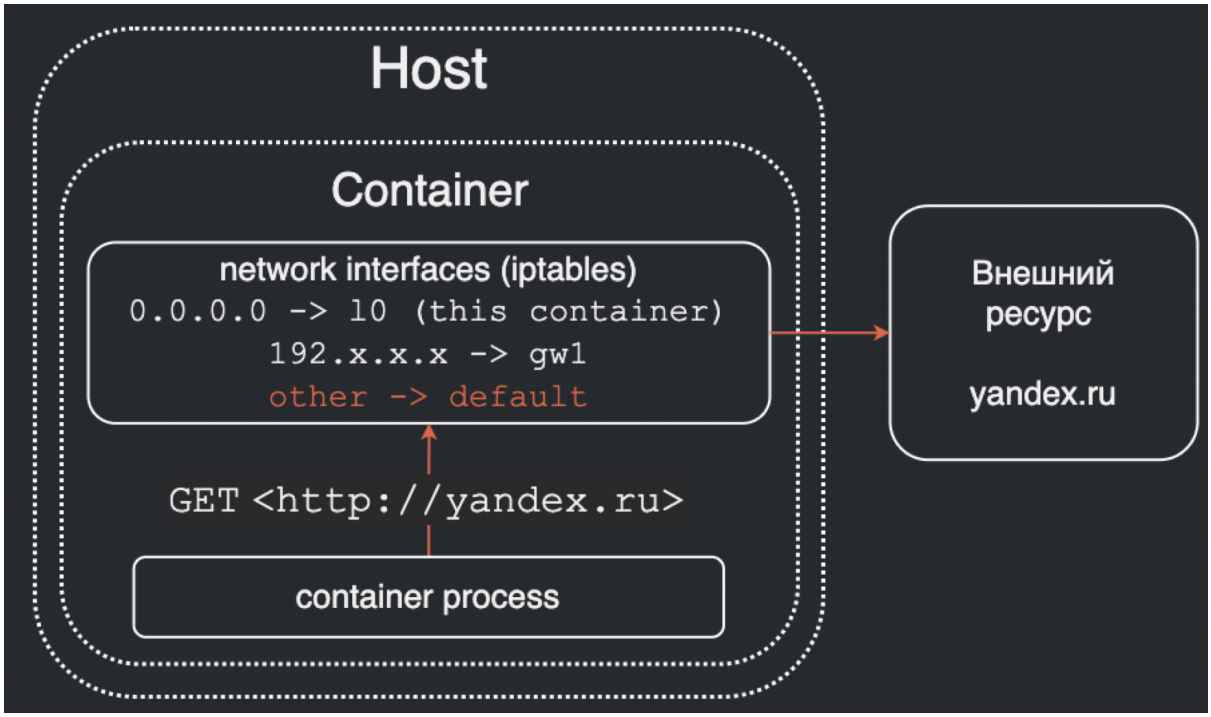
Если у вас есть основной процесс и вы хотите к нему добавить какой-то вспомогательный процесс, например собрать/запроцессить логи, то специально для этого есть Sidecar паттерн. В Kubernetes он есть из коробки, а в Swarm его можно реализовать: вы ставите дополнительные контейнеры рядом со своими контейнерами, в которых будет размещаться расширяющая логика. Связывать контейнеры можно различными способами, главное — оставаться в парадигме "один контейнер — один процесс".



Несколько процессов в одном контейнере

> Сети

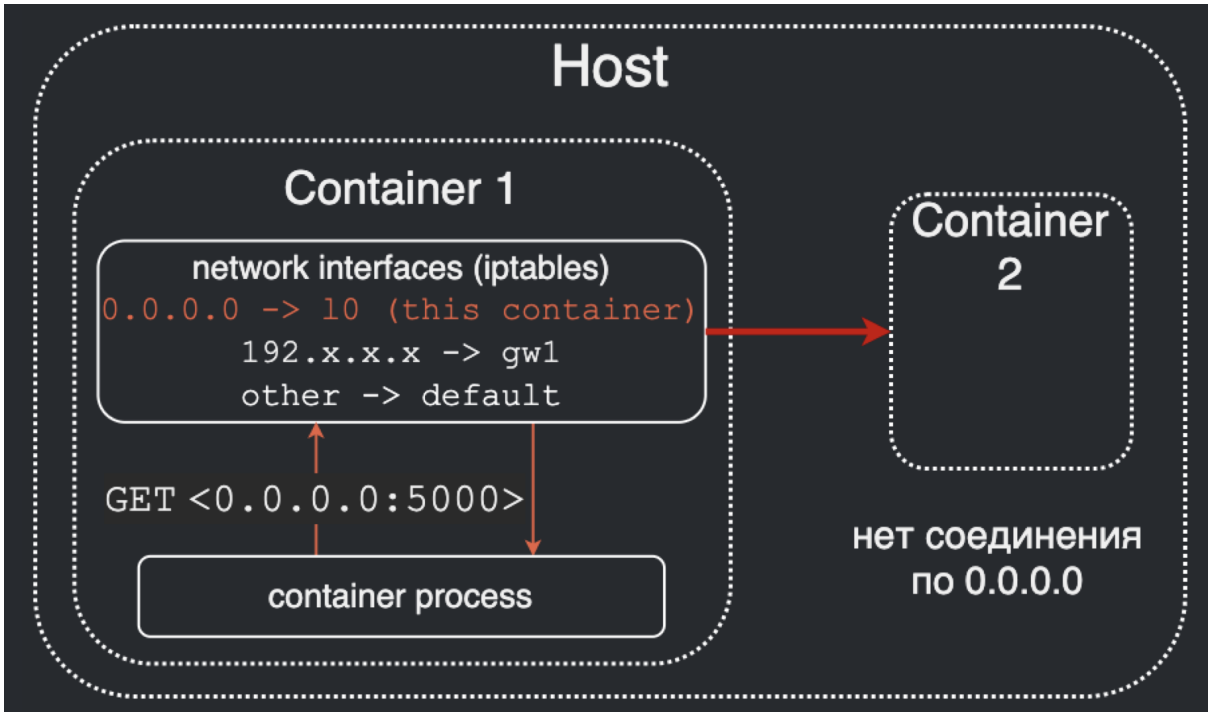
В Docker есть сетевые абстракции, которые как раз позволяют вам иметь несколько контейнеров, в которых приложения работают на одинаковом порту, но на хосте — разные порты.



Запрос на yandex.ru

Как и свой init процесс, контейнер содержит некоторые сетевые интерфейсы, которые тоже изолированы от хоста.

Пример: отправка запроса на yandex.ru из контейнера. Этот запрос идёт на внешний ip адрес, а сетевые механизмы внутри контейнера, откуда отправляется запрос, используют хостовые механизмы. Далее хост делает всё, что нужно.



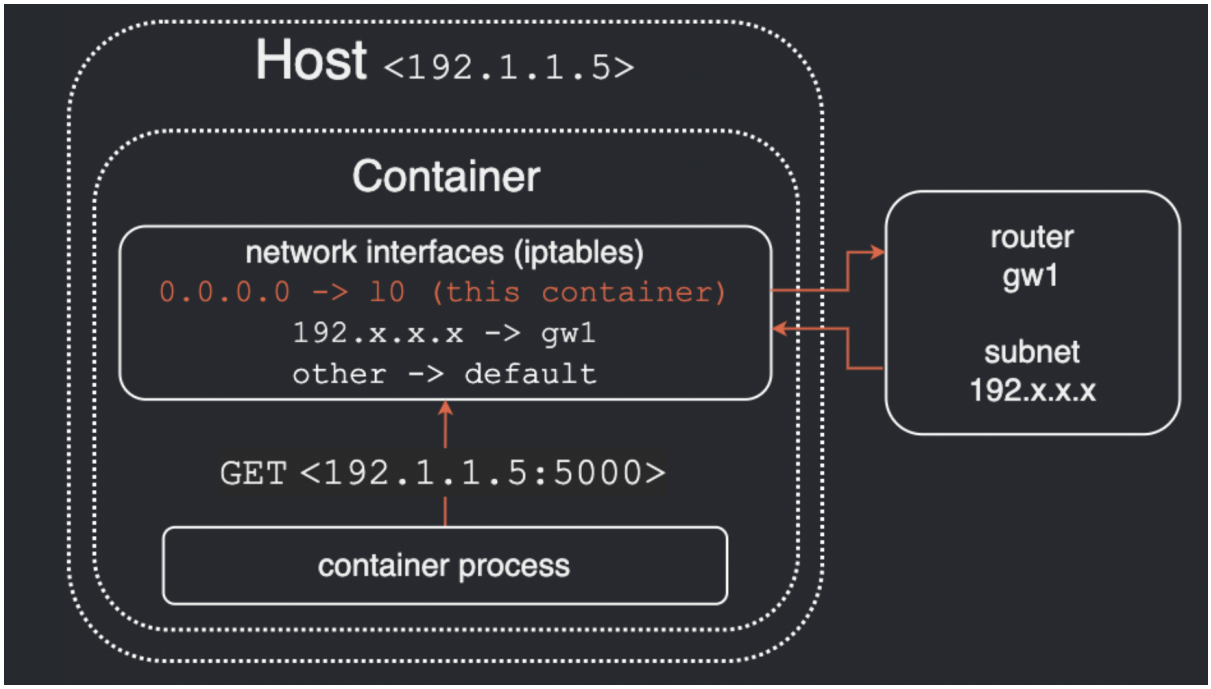
Отправка запроса на сервис, развёрнутый на том же хосте, что и контейнер

При отсутствии Докера запросы будут посылаться на `0.0.0.0` или `127.0.0.1` — с этим кейсом будут проблемы.

Если вы изнутри контейнера будете стучаться на `0.0.0.0`, то это будет внутри контейнера — пакет никуда не выйдет.

Способы это обойти:

- Использование специальных dns имён, которые резолвятся в физический адрес хоста, на котором стоит докер-демон: `host.docker.internal:8000` / `172.17.0.1:8000` (иногда), а не `localhost:8000` / `0.0.0.0:8000`
- Использование режима сети хост: `-network host`, но теперь два разных приложения не смогут использовать один порт, как раньше.
- Указываем либо белый, либо серый ip адрес машины.



Можно найти адрес вашей машины в подсети

> Docker-compose

> Зачем нужен compose

- Связывает приложения с помощью YAML, что, кстати, сразу является и документацией.
- Изолириует docker-compose деплойменты друга от друга (да, контейнеры изолированы сами по себе, но compose ещё больше).
- Упрощает постоянную работу с приложениями:

— например, он находит использованные в предыдущих запусках вольюмы и копирует их в текущий;

— то же самое с сетями;

— пересоздаёт только те контейнеры, которые изменились.

Однако он всё ещё не может деплоить на несколько сервисов: вы будете ограничены одним хостом.

> Фичи compose

```yaml
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

YAML из туториала

Docker-compose будет поднимать два сервиса из `services`.
Для `web` присутствуют инструкции проброса порта из контейнера на хост и др.

```python
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)
```

Код сервиса

В коде создаётся подключение к Redis, использующее dns имя `redis`. Это будет работать, т.к. compose предоставляет механизмы Service discovery, как и Swarm кластер.

> Секции, которые могут понадобиться

```
container_name: имя контейнера, вместо имени по умолчанию
env_file: путь до файла с переменными окружённая
depends_on: список сервисов, которые должны быть подняты перед
restart_policy:
  condition: < none | on-failure | any >
  delay: < s >. Временные зазоры между рестартами
  max_attempt: кол-во попыток рестартануться, после которых рестарты будут завершены
  window: < s >. Время в течение которого решается, успешен ли был рестарт
replicas: кол-во реплик
```

```
rollback_config:
  parallelism: кол-во одновременных апдейтов
  delay: временной зазор
  failure_action: < continue | pause >
  monitor: < s > мониторим, нет ли фейлов
  max_failure_ratio: максимальное кол-во фейлов при котором роллбек продолжается
  order: < start_first | stop_first >
update_config:
   *все то же самое, что и в rollback_config*
```

> Команды управления

Помимо уже знакомых для Docker, в `docker-compose --help` появляется `scale`, позволяющая масштабировать сервисы.

> Полезные ссылки

1. The worst so-called “best practice” for Docker
2. Dockerfile Linter
3. Про оверхед в docker
4. Docker compose common use-cases
5. Use docker compose in production

6. Docker-compose quick start
7. Разные докер-компоузы
8. Стартап приложения