



Дарим панель управления ispmanager

Пользуйтесь панелью бесплатно при создании VPS на любом тарифе до конца 2025 года

Реклама ООО «ИТ-ФИНАНС» энд. LINAKAP35

**2469.03**

Рейтинг

RUVDS.comVDS/VPS-хостинг. Скидка 15% по коду **HABR15**[Подписаться](#)

ru_vds 2 мая 2019 в 13:18

Руководство по Docker Compose для начинающих



9 мин



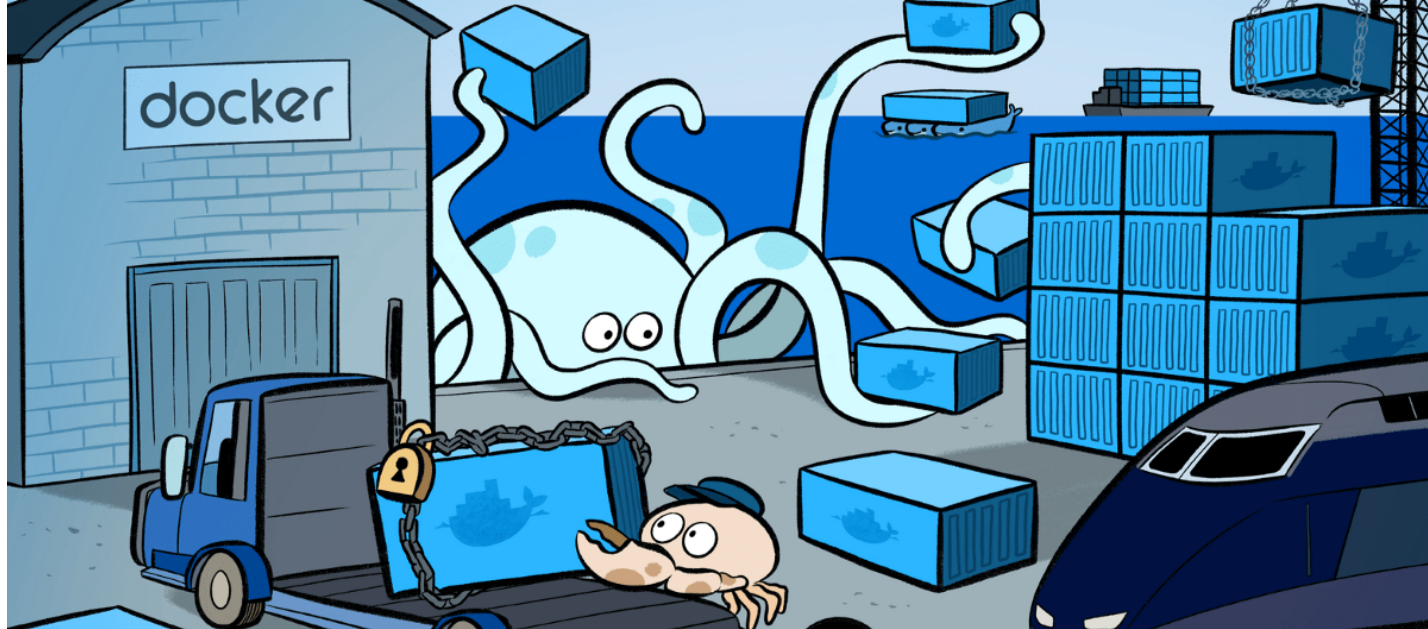
943K

Блог компании RUVDS.com, Виртуализация*, Веб-разработка*

[Тutorial](#)[Перевод](#)

Автор оригинала: Gaël Thomas

Автор статьи, перевод которой мы сегодня публикуем, говорит, что она предназначена для тех разработчиков, которые хотят изучить Docker Compose и идут к тому, чтобы создать своё первое клиент-серверное приложение с использованием Docker. Предполагается, что читатель этого материала знаком с основами Docker. Если это не так — можете взглянуть на [эту](#) серию материалов, на [эту](#) публикацию, где основы Docker рассмотрены вместе с основами Kubernetes, и на [эту](#) статью для начинающих.



Что такое Docker Compose?

Docker Compose — это инструментальное средство, входящее в состав Docker. Оно предназначено для решения задач, связанных с развёртыванием проектов.

Изучая основы Docker, вы могли столкнуться с созданием простейших приложений, работающих автономно, не зависящих, например, от внешних источников данных или от неких сервисов. На практике же подобные приложения — редкость. Реальные проекты обычно включают в себя целый набор совместно работающих приложений.

Как узнать, нужно ли вам, при развёртывании некоего проекта, воспользоваться Docker Compose? На самом деле — очень просто. Если для обеспечения функционирования этого проекта используется несколько сервисов, то Docker Compose может вам пригодиться. Например, в ситуации, когда создают веб-сайт, которому, для выполнения аутентификации пользователей, нужно подключиться к базе данных. Подобный проект может состоять из двух сервисов — того, что обеспечивает работу сайта, и того, который отвечает за поддержку базы данных.

Технология Docker Compose, если описывать её упрощённо, позволяет, с помощью одной команды, запускать множество сервисов.

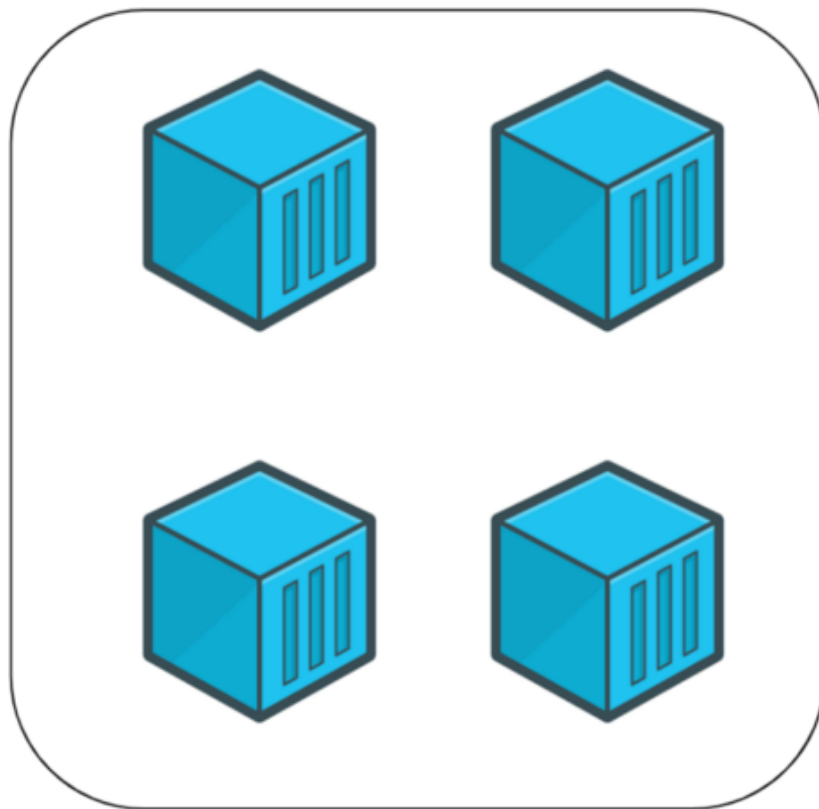
Разница между Docker и Docker Compose

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.



Docker



Docker-Compose

Docker (отдельный контейнер) и Docker Compose (несколько контейнеров)

Типичный сценарий использования Docker Compose

Docker Compose — это, в умелых руках, весьма мощный инструмент, позволяющий очень быстро развёртывать приложения, отличающиеся сложной архитектурой. Сейчас мы рассмотрим пример практического использования Docker Compose, разбор которого позволит вам оценить те преимущества, которые даст вам использование Docker Compose.

Представьте себе, что вы являетесь разработчиком некоего веб-проекта. В этот проект входит два веб-сайта. Первый позволяет людям, занимающимся бизнесом, создавать, всего в несколько щелчков мышью, интернет-магазины. Второй нацелен на поддержку клиентов. Эти два сайта взаимодействуют с одной и той же базой данных.

Ваш проект становится всё популярнее, и оказывается, что мощности сервера, на котором он работает, уже недостаточно. В результате вы решаете перевести весь проект на другую машину.

К сожалению, нечто вроде Docker Compose вы не использовали. Поэтому вам придётся переносить и перенастраивать сервисы по одному, надеясь на то, что вы, в процессе этой работы, ничего не забудете.

Если же вы используете Docker Compose, то перенос вашего проекта на новый сервер — это вопрос, который решается выполнением нескольких команд. Для того чтобы завершить перенос проекта на новое место, вам нужно лишь выполнить кое-какие настройки и загрузить на новый

сервер резервную копию базы данных.

Разработка клиент-серверного приложения с использованием Docker Compose

Теперь, когда вы знаете о том, для чего мы собираемся использовать Docker Compose, пришло время создать ваше первое клиент-серверное приложение с использованием этого инструмента. А именно, речь идёт о разработке небольшого веб-сайта (сервера) на Python, который умеет выдавать файл с фрагментом текста. Этот файл у сервера запрашивает программа (клиент), тоже написанная на Python. После получения файла с сервера программа выводит текст, хранящийся в нём, на экран.

Обратите внимание на то, что мы рассчитываем на то, что вы владеете основами Docker, и на то, что у вас уже установлена платформа Docker.

Приступим к работе над проектом.

1. Создание проекта

Для того чтобы построить ваше первое клиент-серверное приложение, предлагаю начать с создания папки проекта. Она должна содержать следующие файлы и папки:

- Файл `docker-compose.yml`. Это файл Docker Compose, который будет содержать инструкции, необходимые для запуска и настройки сервисов.
- Папка `server`. Она будет содержать файлы, необходимые для обеспечения работы сервера.
- Папка `client`. Здесь будут находиться файлы клиентского приложения.

В результате содержимое главной папки вашего проекта должно выглядеть так:

```
.
├── client/
├── docker-compose.yml
└── server/
2 directories, 1 file
```

2. Создание сервера

Тут мы, в процессе создания сервера, затронем некоторые базовые вещи, касающиеся Docker.

2а. Создание файлов

Перейдите в папку `server` и создайте в ней следующие файлы:

- Файл `server.py` . В нём будет находиться код сервера.
- Файл `index.html` . В этом файле будет находиться фрагмент текста, который должно вывести клиентское приложение.
- Файл `Dockerfile` . Это — файл Docker, который будет содержать инструкции, необходимые для создания окружения сервера.

Вот как должно выглядеть содержимое вашей папки `server/` :

```
.
├── Dockerfile
├── index.html
└── server.py
0 directories, 3 files
```

2b. Редактирование Python-файла.

Добавим в файл `server.py` следующий код:

```
#!/usr/bin/env python3

# Импорт системных библиотек python.
# Эти библиотеки будут использоваться для создания веб-сервера.
# Вам не нужно устанавливать что-то особенное, эти библиотеки устанавливаются вместе

import http.server
import socketserver

# Эта переменная нужна для обработки запросов клиента к серверу.

handler = http.server.SimpleHTTPRequestHandler

# Тут мы указываем, что сервер мы хотим запустить на порте 1234.
# Постарайтесь запомнить эти сведения, так как они нам очень пригодятся в дальнейшем,

with socketserver.TCPServer(("", 1234), handler) as httpd:

    # Благодаря этой команде сервер будет выполняться постоянно, ожидая запросов от к.

    httpd.serve_forever()
```

Этот код позволяет создать простой веб-сервер. Он будет отдавать клиентам файл

`index.html` , содержимое которого позже будет выводиться на веб-странице.

2с. Редактирование HTML-файла

В файл `index.html` добавим следующий текст:

```
Docker-Compose is magic!
```

Этот текст будет передаваться клиенту.

2d. Редактирование файла Dockerfile

Сейчас мы создадим простой файл `Dockerfile` , который будет отвечать за организацию среды выполнения для Python-сервера. В качестве основы создаваемого образа воспользуемся [официальным образом](#), предназначенным для выполнения программ, написанных на Python. Вот содержимое `Dockerfile`:

```
# На всякий случай напоминаю, что Dockerfile всегда должен начинаться с импорта базов
# Для этого используется ключевое слово 'FROM'.
# Здесь нам нужно импортировать образ python (с DockerHub).
# В результате мы, в качестве имени образа, указываем 'python', а в качестве версии -

FROM python:latest

# Для того чтобы запустить в контейнере код, написанный на Python, нам нужно импортир
# Для того чтобы это сделать, мы используем ключевое слово 'ADD'.
# Первый параметр, 'server.py', представляет собой имя файла, хранящегося на компьюте
# Второй параметр, '/server/', это путь, по которому нужно разместить указанный файл
# Здесь мы помещаем файл в папку образа '/server/'.

ADD server.py /server/
ADD index.html /server/

# Здесь мы воспользуемся командой 'WORKDIR', возможно, новой для вас.
# Она позволяет изменить рабочую директорию образа.
# В качестве такой директории, в которой будут выполняться все команды, мы устанавлив

WORKDIR /server/
```

Теперь займёмся работой над клиентом.

3. Создание клиента

Создавая клиентскую часть нашего проекта, мы попутно вспомним некоторые основы Docker.

3a. Создание файлов

Перейдите в папку вашего проекта `client` и создайте в ней следующие файлы:

- Файл `client.py`. Тут будет находиться код клиента.
- Файл `Dockerfile`. Этот файл играет ту же роль, что и аналогичный файл в папке сервера. Именно, он содержит инструкцию, описывающую создание среды для выполнения клиентского кода.

В результате ваша папка `client/` на данном этапе работы должна выглядеть так:

```
.
├── client.py
└── Dockerfile
0 directories, 2 files
```

3b. Редактирование Python-файла

Добавим в файл `client.py` следующий код:

```
#!/usr/bin/env python3

# Импортируем системную библиотеку Python.
# Она используется для загрузки файла 'index.html' с сервера.
# Ничего особенного устанавливать не нужно, эта библиотека устанавливается вместе с P

import urllib.request

# Эта переменная содержит запрос к 'http://localhost:1234/'.
# Возможно, сейчас вы задаётесь вопросом о том, что такое 'http://localhost:1234'.
# localhost указывает на то, что программа работает с локальным сервером.
# 1234 - это номер порта, который вам предлагалось запомнить при настройке серверного

fp = urllib.request.urlopen("http://localhost:1234/")

# 'encodedContent' соответствует закодированному ответу сервера ('index.html').
# 'decodedContent' соответствует раскодированному ответу сервера (тут будет то, что м

encodedContent = fp.read()
decodedContent = encodedContent.decode("utf8")

# Выводим содержимое файла, полученного с сервера ('index.html').

print(decodedContent)
```



```
# Закрываем соединение с сервером.
```

```
fp.close()
```

Благодаря этому коду клиентское приложение может загрузить данные с сервера и вывести их на экран.

3с. Редактирование файла Dockerfile

Как и в случае с сервером, мы создаём для клиента простой Dockerfile , ответственный за формирование среды, в которой будет работать клиентское Python-приложение. Вот код клиентского Dockerfile :

```
# То же самое, что и в серверном Dockerfile.

FROM python:latest

# Импортируем 'client.py' в папку '/client/'.

ADD client.py /client/

# Устанавливаем в качестве рабочей директории '/client/'.

WORKDIR /client/
```

4. Docker Compose

Как вы могли заметить, мы создали два разных проекта: сервер и клиент. У каждого из них имеется собственный файл Dockerfile . До сих пор всё происходящее не выходит за рамки основ работы с Docker. Теперь же мы приступаем к работе с Docker Compose. Для этого обратимся к файлу docker-compose.yml , расположенному в корневой папке проекта.

Обратите внимание на то, что тут мы не стремимся рассмотреть абсолютно все команды, которые можно использовать в docker-compose.yml . Наша главная цель — разобрать практический пример, дающий вам базовые знания по Docker Compose.

Вот код, который нужно поместить в файл docker-compose.yml :

```
# Файл docker-compose должен начинаться с тега версии.
# Мы используем "3" так как это - самая свежая версия на момент написания этого кода.

version: "3"
```



```
# Следует учитывать, что docker-compose 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:

```
# Как уже было сказано, мы собираемся создать клиентское и серверное приложения.  
# Это означает, что нам нужно два сервиса.  
# Первый сервис (контейнер): сервер.  
# Назвать его можно так, как нужно разработчику.  
# Понятное название сервиса помогает определить его роль.  
# Здесь мы, для именования соответствующего сервиса, используем ключевое слово 'ser'
```

server:

```
# Ключевое слово "build" позволяет задать  
# путь к файлу Dockerfile, который нужно использовать для создания образа,  
# который позволит запустить сервис.  
# Здесь 'server/' соответствует пути к папке сервера,  
# которая содержит соответствующий Dockerfile.
```

build: server/

```
# Команда, которую нужно запустить после создания образа.  
# Следующая команда означает запуск "python ./server.py".
```

command: python ./server.py

```
# Вспомните о том, что в качестве порта в 'server/server.py' указан порт 1234.  
# Если мы хотим обратиться к серверу с нашего компьютера (находясь за пределами к  
# мы должны организовать перенаправление этого порта на порт компьютера.  
# Сделать это нам поможет ключевое слово 'ports'.  
# При его использовании применяется следующая конструкция: [порт компьютера]:[пор  
# В нашем случае нужно использовать порт компьютера 1234 и организовать его связь  
# 1234 контейнера (так как именно на этот порт сервер  
# ожидает поступления запросов).
```

ports:

```
- 1234:1234
```

```
# Второй сервис (контейнер): клиент.  
# Этот сервис назван 'client'.
```

client:

```
# Здесь 'client/' соответствует пути к папке, которая содержит  
# файл Dockerfile для клиентской части системы.
```

```
build: client/
```

```
# Команда, которую нужно запустить после создания образа.
```

```
# Следующая команда означает запуск "python ./client.py".
```

```
command: python ./client.py
```

```
# Ключевое слово 'network_mode' используется для описания типа сети.
```

```
# Тут мы указываем то, что контейнер может обращаться к 'localhost' компьютера.
```

```
network_mode: host
```

```
# Ключевое слово 'depends_on' позволяет указывать, должен ли сервис,
```

```
# прежде чем запуститься, ждать, когда будут готовы к работе другие сервисы.
```

```
# Нам нужно, чтобы сервис 'client' дождался бы готовности к работе сервиса 'serve
```

```
depends_on:
```

```
- server
```

5. Сборка проекта

После того, как в `docker-compose.yml` внесены все необходимые инструкции, проект нужно собрать. Этот шаг нашей работы напоминает использование команды `docker build`, но соответствующая команда имеет отношение к нескольким сервисам:

```
$ docker-compose build
```

6. Запуск проекта

Теперь, когда проект собран, пришло время его запустить. Этот шаг нашей работы соответствует шагу, на котором, при работе с отдельными контейнерами, выполняется команда `docker run`:

```
$ docker-compose up
```

После выполнения этой команды в терминале должен появиться текст, загруженный клиентом с сервера: `Docker-Compose is magic!`.

Как уже было сказано, сервер использует порт компьютера `1234` для обслуживания запросов клиента. Поэтому, если перейти в браузере по адресу <http://localhost:1234/>, в нём будет отображена страница с текстом `Docker-Compose is magic!`.

Полезные команды

Рассмотрим некоторые команды, которые могут вам пригодиться при работе с Docker Compose.

Эта команда позволяет останавливать и удалять контейнеры и другие ресурсы, созданные командой `docker-compose up` :

```
$ docker-compose down
```

Эта команда выводит журналы сервисов:

```
$ docker-compose logs -f [service name]
```

Например, в нашем проекте её можно использовать в таком виде: `$ docker-compose logs -f [service name]` .

С помощью такой команды можно вывести список контейнеров:

```
$ docker-compose ps
```

Данная команда позволяет выполнить команду в выполняющемся контейнере:

```
$ docker-compose exec [service name] [command]
```

Например, она может выглядеть так: `docker-compose exec server ls` .

Такая команда позволяет вывести список образов:

```
$ docker-compose images
```

Итоги

Мы рассмотрели основы работы с технологией Docker Compose, знание которых позволит вам пользоваться этой технологией и, при желании, приступить к её более глубокому изучению. [Вот](#) репозиторий с кодом проекта, который мы здесь рассматривали.

Уважаемые читатели! Пользуетесь ли вы Docker Compose в своих проектах?

Теги: Docker, Docker Compose, разработка

Хабы: Блог компании RUVDS.com, Виртуализация, Веб-разработка

♦ +34

📖 996



💬 14 +14

Редакторский дайджест



Присылаем лучшие статьи раз в месяц

Электронпочта



RUVDS.com

VDS/VPS-хостинг. Скидка 15% по коду **HABR15**

[Telegram](#) [ВКонтакте](#) [Twitter](#)



572

Карма

375.2

Рейтинг

@ru_vds

Пользователь

Подписаться



Комментарии 14



👤 **vanyas** 2 мая 2019 в 15:59

Тут недавно уже был пост, по то, как не надо делать докерфайлы, почитайте. (я про :latest и два ADD)



Ответить



👤 **MasMaX** 3 мая 2019 в 07:04 ✎ ^

И WORKDIR надо ставить перед ADD. Это ведь разные слои и в текущем порядке последний слой будет заново генерироваться при каждом изменении файла проекта.

Плюс прописывание команда только в композ файле, это не лучшее решение. Оставлять образ без точки входа нельзя. Любой образ должен уметь запускаться командой start без допов.




Ответить



👤 **iNibbler** 21 ноя 2019 в 18:40 ^

Как статья называется? Подскажите pls.

↑  ↓ Ответить  ...


o  **reallord** 2 мая 2019 в 17:58

Блин и тут учат делать :latest!

Ну почему во всех доках для нубов ВСЕГДА latest!


А потом мы удивляемся что выросло целое поколение DevOps инженеров, которые совершат одни и те же дурацкие ошибки. А как их не совершать, если в самой первой инструкции в жизни инженера написано FROM xxxxxxxx:latest

↑  ↓ Ответить  ...

o  **karavan_750** 3 мая 2019 в 19:46 ^

А что, если образы брать из своего репозитория, где версионирование под достаточным контролем?

↑  ↓ Ответить  ...

o  **fpinger** 3 мая 2019 в 01:42



:latest имеет право на жизнь. Например добавляем третий контейнер с adminer для работы с БД на этапе разработки. Это будет вспомогательный контейнер с утилитой, которая уберётся из продакшена.

↑  ↓ Ответить  ...

o  **MasMaX** 3 мая 2019 в 07:05 ^

Обычно его ставят для своих образов. В которых ты точно уверен и знаешь что под капотом.



↑  ↓ Ответить  ...

o  **fpinger** 3 мая 2019 в 08:11  ^

В контексте docker-compose для сборки окружения разработки :latest вполне себе подходит. Причём даже над Dockerfile для такого контейнера нет смысла заморачиваться. Конкретные версии нужны для того, что будет и в разработке, и в продакшене.


Всегда есть нечто вспомогательное.

↑  ↓ Ответить  ...

o  **vanyas** 3 мая 2019 в 10:10  ^

Да как может подходить хоть куда-то python:latest, вот завтра выйдет питон 4, не совместимый с питон 3, и ваш питон скрипт просто не запустится

↑  ↓ Ответить  ...

o  **fpinger** 3 мая 2019 в 11:55 ^

А вот adminer:latest вполне. Вообще вы не думаете, а повторяете чужие слова. Для вас контекста для размышления нет в моей фразе:

> Конкретные версии нужны для того, что будет и в разработке, и в продакшене.
Ещё разработчиков создавших возможность :latest опустите за возможность для конкретных ситуаций.

↑  ↓ Ответить  ...


o  **DieSlogan** 3 мая 2019 в 17:54

Файл docker-compose должен начинаться с тега версии.
Мы используем «3» так как это — самая свежая версия на момент написания этого кода.

version: «3»




Я использую 2.4, потому что у меня в файле еще настройки ограничения по процессорам и RAM. Если использовать ветку 3, то там основной упор идет на swarm, которого у меня нет.

↑  ↓ Ответить  ...

o  **kest70** 2 сен 2023 в 04:42

Ктонибудь может подсказать, как то, что собирает docker-compose запихать в отдельный контейнер docker - образ, и запускать его также отдельно?

↑  ↓ Ответить  ...


o  **Securityhigh** 13 окт 2023 в 05:28  

Если такой варианты существует и ты его нашел – расскажи пожалуйста.

Тоже столкнулся с этой проблемой, у меня докер-реестр в котором образы и докер-компас, который приходится таскать с собой на серверы, костыльно получается, все автоматизированно, а подобным довольно весомым ситуациям внимания не уделено.

Как бы оно и понятно, компас – просто надстройка для управления оркестром докеров, но все же :)

↑  ↓ Ответить  ...

o  **trabl** 15 сен 2023 в 22:54

Отличная статья для новичков, очень доступным языком написано. А про лучшие практики: версии образов, количество слоёв и прочее это уже тема отдельной статьи, сюда если включить всю эту мишуру статья станет менее доступна для понимания, имхо.

↑  ↓ Ответить  ...

Зарегистрируйтесь на Хабре, чтобы оставить комментарий