Docker и Docker Compose: общая инструкция

Table of Contents

1. Скачать PDF

Скачать инструкцию в PDF

2. Основные понятия

Docker позволяет запускать приложения в изолированных окружениях — контейнерах.

Основная цепочка работы:

```
Dockerfile -> Image -> Container
```

Основные сущности:

- **Dockerfile** — инструкция по сборке приложения.
- **Image / образ** — готовый шаблон файловой системы приложения.
- **Container / контейнер** — запущенный экземпляр образа.
- **Volume** — постоянное хранилище данных.
- **Network** — сеть для связи контейнеров.
- **Registry** — хранилище образов, например Docker Hub.

Контейнер представляет собой изолированный процесс со своей файловой системой, сетью и деревом процессов.

3. Проверка установки

```
docker --version
docker info
docker compose version
```

Современная команда Compose:

```
docker compose
```

Старая команда:


```
docker-compose
```

Вариант с дефисом относится к устаревшей standalone-версии Compose.

Если команда `docker compose` не работает:

```
sudo apt update
sudo apt install docker-compose-plugin
```

Проверка:

```
docker compose version
```

4. Проверка работы Docker

```
sudo docker run hello-world
```

При выполнении команды Docker:

1. Ищет образ локально.
2. Если образ отсутствует — скачивает его из Registry.
3. Создаёт контейнер.
4. Запускает основной процесс контейнера.

5. Показывает результат работы.

5. Работа с образами

5.1. Просмотр образов

```
docker images
```

Полная форма:

```
docker image ls
```

5.2. Скачивание образа

```
docker pull nginx
```

С конкретным тегом:

```
docker pull nginx:1.27
```

Если тег не указан, обычно используется `latest`. В рабочих проектах лучше фиксировать конкретную версию.

Пример:


```
FROM nginx:1.27
```

5.3. Удаление образа

```
docker rmi nginx
```

По ID:

```
docker rmi 4cad75abc83d
```

Принудительно:

```
docker rmi -f nginx
```

5.4. Подробная информация об образе

```
docker image inspect nginx
```

5.5. История слоёв

```
docker history nginx
```

5.6. Очистка неиспользуемых образов

```
docker image prune
```

Включая все образы, которые не используются контейнерами:

```
docker image prune -a
```

6. Запуск контейнера

Базовая команда:

```
docker run IMAGE
```

Пример:

```
docker run nginx
```

6.1. Запуск в фоне

```
docker run -d nginx
```

Опция `-d` означает detached mode.

6.2. Задание имени

```
docker run -d --name web nginx
```

После этого контейнером можно управлять по имени:

```
docker stop web
docker logs web
docker inspect web
```

6.3. Проброс порта

```
docker run -d \
  --name web \
  -p 8080:80 \
  nginx
```

Формат:

```
-p ПОРТ_ХОСТА:ПОРТ_КОНТЕЙНЕРА
```

В данном случае:

```
localhost:8080 -> container:80
```

Проверка:

```
curl http://localhost:8080
```

Важно: `EXPOSE 80` в Dockerfile сам по себе не публикует порт. Для доступа с хоста требуется опция `-p`.

6.4. Переменные окружения

```
docker run -d \
  --name app \
  -e APP_ENV=production \
  -e APP_PORT=8080 \
  myapp
```

Из файла:

```
docker run --env-file .env myapp
```

6.5. Подключение каталога хоста

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


```
  nginx
```

Где:

- `$(pwd)/html` — каталог хоста.
- `/usr/share/nginx/html` — каталог внутри контейнера.
- `ro` — режим только для чтения.

Более явный синтаксис:

```
docker run -d \
  --name web \
  --mount type=bind,source="$(pwd)/html",target=/usr/share/nginx/html,readonly \
  nginx
```

6.6. Подключение Docker Volume

```
docker volume create app-data
```

```
docker run -d \
  --name postgres \
  -e POSTGRES_PASSWORD=secret \
  -v app-data:/var/lib/postgresql/data \
  postgres
```

6.7. Автоматическое удаление контейнера

```
docker run --rm alpine echo "Hello"
```

6.8. Интерактивный запуск

```
docker run --rm -it ubuntu bash
```

- `-i` — оставить STDIN открытым.
- `-t` — создать псевдотерминал.
- `--rm` — удалить контейнер после завершения.

7. Просмотр контейнеров

7.1. Только запущенные

```
docker ps
```

Полная форма:

```
docker container ls
```

7.2. Все контейнеры

```
docker ps -a
```

7.3. Только ID

```
docker ps -q
```

Все контейнеры:

```
docker ps -aq
```

7.4. Последний созданный контейнер

```
docker ps -l
```

7.5. Фильтрация

```
docker ps --filter status=running
docker ps --filter name=web
```

7.6. Настраиваемый вывод

```
docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}\t{{.Ports}}'
```

8. Управление контейнерами

8.1. Остановка

```
docker stop web
```

Задать тайм-аут:

```
docker stop -t 30 web
```

8.2. Запуск остановленного контейнера

```
docker start web
```

8.3. Перезапуск

```
docker restart web
```

8.4. Принудительное завершение

```
docker kill web
```

Обычно сначала следует использовать `docker stop`, а `docker kill` — только если контейнер завис.

8.5. Приостановка процессов

```
docker pause web
docker unpause web
```

8.6. Удаление контейнера

```
docker rm web
```

Принудительно:

```
docker rm -f web
```

8.7. Удаление всех остановленных контейнеров

```
docker container prune
```

9. Логи контейнера

```
docker logs web
```

Следить за логами в реальном времени:

```
docker logs -f web
```

Последние 100 строк:

```
docker logs --tail 100 web
```

За последний час:

```
docker logs --since 1h web
```

С отметками времени:

```
docker logs -t web
```

Удобная комбинация:

```
docker logs -f --tail 100 web
```

Приложение желательно настраивать так, чтобы оно выводило логи в `stdout` и `stderr`.

10. Выполнение команд внутри контейнера

10.1. Открытие shell

Для Ubuntu или Debian:

```
docker exec -it web bash
```

Для Alpine и минимальных образов:

```
docker exec -it web sh
```

10.2. Выполнение одной команды

```
docker exec web ls -la /app
docker exec web env
docker exec postgres psql -U postgres
```

10.3. Выполнение команды от root


```
docker exec -u root -it web sh
```

11. Диагностика контейнера

11.1. Подробная конфигурация

```
docker inspect web
```

Получить IP:

```
docker inspect \
  -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
  web
```

Посмотреть состояние:

```
docker inspect \
  -f '{{.State.Status}} {{.State.ExitCode}}' \
  web
```

11.2. Использование ресурсов


```
docker stats
```

Один контейнер:

```
docker stats web
```

Один снимок:

```
docker stats --no-stream
```

11.3. Процессы внутри контейнера

```
docker top web
```

11.4. Изменённые файлы

```
docker diff web
```

12. Копирование файлов

Из хоста в контейнер:


```
docker cp config.json web:/app/config.json
```

Из контейнера на хост:

```
docker cp web:/app/logs ./logs
```

Для постоянной конфигурации лучше использовать bind mount или включать файлы в образ.

13. Сборка собственного образа

Пример структуры проекта:

```
myapp/
□□□ Dockerfile
□□□ .dockerignore
□□□ requirements.txt
□□□ app.py
```

Пример Dockerfile:

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .
```


```
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["python", "app.py"]
```

13.1. Сборка образа

```
docker build -t myapp:1.0 .
```

Где:

- `-t myapp:1.0` — имя и тег образа.
- `.` — build context.

13.2. Запуск собранного образа

```
docker run -d \
  --name myapp \
  -p 8000:8000 \
  myapp:1.0
```

13.3. Пересборка

```
docker build -t myapp:1.1 .
```

Без использования кэша:

```
docker build --no-cache -t myapp:1.1 .
```

Подробный вывод:

```
docker build --progress=plain -t myapp:1.1 .
```

13.4. Build arguments

Dockerfile:

```
ARG APP_VERSION
RUN echo "Version: ${APP_VERSION}"
```

Сборка:

```
docker build \
  --build-arg APP_VERSION=1.5.0 \
  -t myapp:1.5.0 .
```

14. Файл .dockerignore

Пример:

```
.git
.gitignore
node_modules
venv
__pycache__
*.log
.env
.idea
.vscode
README.md
```

Файл `.dockerignore` исключает ненужные данные из build context, ускоряет сборку и уменьшает риск случайного попадания секретов в образ.

15. Сети Docker

15.1. Просмотр сетей

```
docker network ls
```

15.2. Создание сети

```
docker network create app-network
```

15.3. Запуск контейнеров в одной сети

```
docker run -d \
  --name database \
  --network app-network \
  postgres
```

```
docker run -d \
  --name backend \
  --network app-network \
  mybackend
```

Контейнер `backend` сможет обращаться к базе по имени:

```
database:5432
```

15.4. Подключение существующего контейнера

```
docker network connect app-network web
```

15.5. Отключение

```
docker network disconnect app-network web
```

15.6. Удаление сети

```
docker network rm app-network
```

16. Volumes

16.1. Просмотр

```
docker volume ls
```

16.2. Создание

```
docker volume create postgres-data
```

16.3. Информация


```
docker volume inspect postgres-data
```

16.4. Удаление

```
docker volume rm postgres-data
```

16.5. Очистка неиспользуемых volumes

```
docker volume prune
```

Удаление контейнера не удаляет named volume автоматически.

17. Docker Compose

Docker Compose используется, когда приложение состоит из нескольких контейнеров, например:

```
frontend
backend
database
redis
nginx
```

Вся конфигурация описывается в файле `compose.yaml`.

18. Пример compose.yaml

```
services:
  database:
    image: postgres:16
    environment:
      POSTGRES_DB: app
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
    volumes:
      - postgres-data:/var/lib/postgresql/data
    networks:
      - backend-network
    restart: unless-stopped

  backend:
    build:
      context: ./backend
    environment:
      DB_HOST: database
      DB_PORT: "5432"
      DB_NAME: app
      DB_USER: app
      DB_PASSWORD: secret
    depends_on:
      - database
    ports:
      - "8080:8080"
    networks:
```


```
      - backend-network
      - frontend-network
    restart: unless-stopped

  frontend:
    build:
      context: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend
    networks:
      - frontend-network
    restart: unless-stopped

volumes:
  postgres-data:

networks:
  backend-network:
  frontend-network:
```

Структура проекта:

```
project/
□□□ compose.yaml
□□□ .env
□□□ frontend/
□   □□□ Dockerfile
```


```
□   □□□ ...
□□□ backend/
    □□□ Dockerfile
    □□□ ...
```

19. Проверка Compose-файла

```
docker compose config
```

Команда проверяет YAML, подставляет переменные и показывает итоговую конфигурацию.

Только проверка:

```
docker compose config --quiet
```

Список сервисов:

```
docker compose config --services
```

Список образов:

```
docker compose config --images
```

20. Запуск Compose-проекта

20.1. Запуск с логами

```
docker compose up
```

Остановка выполняется сочетанием `Ctrl+C`.

20.2. Запуск в фоне

```
docker compose up -d
```

20.3. Сборка и запуск

```
docker compose up -d --build
```

20.4. Запуск одного сервиса

```
docker compose up -d backend
```

20.5. Запуск без зависимостей

```
docker compose up -d --no-deps backend
```

21. Просмотр состояния Compose

```
docker compose ps
```

Все контейнеры, включая остановленные:

```
docker compose ps -a
```

Процессы:

```
docker compose top
```

22. Логи Compose

Все сервисы:

```
docker compose logs
```

В реальном времени:

```
docker compose logs -f
```

Последние 100 строк:

```
docker compose logs --tail 100
```

Конкретный сервис:

```
docker compose logs -f backend
```

Несколько сервисов:

```
docker compose logs -f backend database
```

За последние 30 минут:

```
docker compose logs --since 30m
```

23. Выполнение команд через Compose

Открыть shell:


```
docker compose exec backend sh
```

Если доступен Bash:

```
docker compose exec backend bash
```

Выполнить команду:

```
docker compose exec backend env
docker compose exec database psql -U app app
```

От root:

```
docker compose exec -u root backend sh
```

24. Разница между exec и run

`docker compose exec` запускает команду внутри уже работающего контейнера:

```
docker compose exec backend sh
```

`docker compose run` создаёт отдельный временный контейнер сервиса:


```
docker compose run --rm backend python manage.py migrate
```

Другие примеры:

```
docker compose run --rm backend pytest
docker compose run --rm backend npm test
```

25. Остановка Compose-проекта

25.1. Остановка без удаления

```
docker compose stop
```

Повторный запуск:

```
docker compose start
```

25.2. Остановка и удаление контейнеров и сети

```
docker compose down
```

25.3. Удаление вместе с volumes

```
docker compose down -v
```

Эта команда может удалить данные базы данных и другие данные из named volumes.

25.4. Удаление образов

```
docker compose down --rmi local
docker compose down --rmi all
```

25.5. Удаление orphan-контейнеров

```
docker compose down --remove-orphans
```

26. Пересборка проекта

Полная пересборка:

```
docker compose build
docker compose up -d
```

Одной командой:

```
docker compose up -d --build
```

Конкретный сервис:

```
docker compose build backend
docker compose up -d backend
```

Без кэша:

```
docker compose build --no-cache
docker compose up -d
```

Скачать свежие базовые образы:

```
docker compose build --pull
```

Принудительно пересоздать контейнеры:

```
docker compose up -d --force-recreate
```

27. Обновление приложения

Типовой порядок:

```
git pull
docker compose build
docker compose up -d
docker compose ps
docker compose logs --tail 100
```

Краткий вариант:

```
git pull
docker compose up -d --build
docker compose ps
```

Если используются готовые образы из Registry:

```
docker compose pull
docker compose up -d
```

28. Переменные окружения в Compose

Пример `.env`:


```
POSTGRES_DB=app
POSTGRES_USER=app
POSTGRES_PASSWORD=change-me
BACKEND_PORT=8080
```

Пример `compose.yaml`:

```
services:
  database:
    image: postgres:16
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}

  backend:
    build: ./backend
    ports:
      - "${BACKEND_PORT}:8080"
```

Проверка подстановки:

```
docker compose config
```

Файлы с секретами не следует коммитить:

```
.env
```


```
.env.*
!.env.example
```

Пример `.env.example`:

```
POSTGRES_DB=app
POSTGRES_USER=app
POSTGRES_PASSWORD=replace-me
BACKEND_PORT=8080
```

29. depends_on и готовность сервиса

Простая запись:

```
depends_on:
  - database
```

Она задаёт порядок запуска контейнеров, но не гарантирует, что база уже готова принимать подключения.

Более надёжный вариант:

```
services:
  database:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
```


```
    healthcheck:
      test:
        - CMD-SHELL
        - pg_isready -U postgres
      interval: 5s
      timeout: 3s
      retries: 10

  backend:
    build: ./backend
    depends_on:
      database:
        condition: service_healthy
```

Само приложение также должно уметь повторять подключение после временной ошибки.

30. Политики перезапуска

```
restart: "no"
```

Не перезапускать автоматически.

```
restart: always
```

Всегда перезапускать.


```
restart: unless-stopped
```

Перезапускать, кроме случая ручной остановки.

```
restart: on-failure
```

Перезапускать только при ненулевом коде завершения.

Для обычного сервера часто используется:

```
restart: unless-stopped
```

31. Ограничение ресурсов

```
docker run -d \
  --name app \
  --memory 512m \
  --cpus 1.5 \
  myapp
```

Посмотреть лимиты:

```
docker inspect app
```

Изменить лимиты:

```
docker update \
  --memory 1g \
  --cpus 2 \
  app
```

32. Очистка Docker

Посмотреть занимаемое место:

```
docker system df
docker system df -v
```

Удалить остановленные контейнеры:

```
docker container prune
```

Удалить неиспользуемые образы:

```
docker image prune
```

Удалить неиспользуемые сети:


```
docker network prune
```

Удалить неиспользуемые volumes:

```
docker volume prune
```

Общая очистка:

```
docker system prune
```

Более агрессивная:

```
docker system prune -a
```

Включая volumes:

```
docker system prune -a --volumes
```

Последняя команда может удалить неиспользуемые volumes с важными данными.

33. Остановка всех контейнеров

Правильный вариант с одинаковыми правами:


```
sudo docker stop $(sudo docker ps -q)
```

Удалить все контейнеры:

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

Проблемный вариант:

```
sudo docker stop $(docker ps -aq)
```

Команда внутри `$()` выполняется от обычного пользователя. Если у него нет доступа к Docker socket, список контейнеров окажется пустым.

Безопасный вариант:

```
docker ps -q | xargs -r docker stop
```

С `sudo`:

```
sudo docker ps -q | xargs -r sudo docker stop
```

34. Работа без sudo

Добавить пользователя в группу Docker:

```
sudo usermod -aG docker "$USER"
```

Применить изменения в текущем терминале:

```
newgrp docker
```

Проверить:

```
docker ps
```

Членство в группе `docker` фактически даёт пользователю привилегии уровня root.

35. Типовой порядок работы с одиночным контейнером

```
# 1. Собрать образ
docker build -t myapp:1.0 .

# 2. Проверить образ
docker images
```


```
# 3. Запустить контейнер
docker run -d \
  --name myapp \
  -p 8080:8080 \
  myapp:1.0

# 4. Проверить состояние
docker ps

# 5. Посмотреть логи
docker logs -f --tail 100 myapp

# 6. Проверить приложение
curl http://localhost:8080

# 7. Зайти внутрь
docker exec -it myapp sh

# 8. Остановить
docker stop myapp

# 9. Удалить контейнер
docker rm myapp
```

36. Типовой порядок работы с Compose

```
# 1. Перейти в каталог проекта
cd ~/project
```


```
# 2. Проверить файлы
ls -la

# 3. Проверить конфигурацию
docker compose config

# 4. Собрать и запустить
docker compose up -d --build

# 5. Проверить состояние
docker compose ps

# 6. Посмотреть логи
docker compose logs -f --tail 100

# 7. Посмотреть логи сервиса
docker compose logs -f backend

# 8. Выполнить команду внутри сервиса
docker compose exec backend sh

# 9. Перезапустить сервис
docker compose restart backend

# 10. Остановить проект
docker compose down
```

37. Типовой порядок диагностики

37.1. Проверить контейнеры

```
docker compose ps -a
```

Основные состояния:

- `Up` — контейнер работает.
- `Exited (0)` — процесс завершился успешно.
- `Exited (1)` — процесс завершился с ошибкой.
- `Restarting` — контейнер постоянно падает и перезапускается.
- `unhealthy` — контейнер не проходит healthcheck.

37.2. Посмотреть логи

```
docker compose logs --tail 200 SERVICE
```

37.3. Проверить конфигурацию

```
docker compose config
```

37.4. Проверить порты

```
docker compose ps
ss -lntp
```

37.5. Проверить процессы

```
docker compose exec SERVICE ps aux
```

37.6. Проверить переменные окружения

```
docker compose exec SERVICE env
```

37.7. Проверить сеть

```
docker compose exec backend getent hosts database
```

```
docker compose exec backend ping database
```

В минимальном образе команда `ping` может отсутствовать.

37.8. Проверить HTTP внутри контейнера

```
docker compose exec frontend wget -qO- http://localhost:3000
```

Или:

```
docker compose exec frontend curl http://localhost:3000
```

37.9. Посмотреть причину завершения

```
docker inspect CONTAINER \
  --format 'status={{.State.Status}} exit={{.State.ExitCode}} error={{.State.Error}}'
```

37.10. Пересобрать с подробным выводом

```
docker compose build --no-cache --progress=plain SERVICE
```

38. Команды, которые нужно запомнить

38.1. Docker


```
docker ps
docker ps -a
docker images
docker pull IMAGE
docker build -t IMAGE:TAG .
docker run -d --name NAME -p HOST:CONTAINER IMAGE
docker logs -f CONTAINER
docker exec -it CONTAINER sh
docker inspect CONTAINER
docker stats
docker stop CONTAINER
docker start CONTAINER
docker restart CONTAINER
docker rm CONTAINER
docker rmi IMAGE
docker system df
docker system prune
```

38.2. Docker Compose

```
docker compose config
docker compose build
docker compose up -d
docker compose up -d --build
docker compose ps
docker compose logs -f
docker compose logs -f SERVICE
docker compose exec SERVICE sh
```


```
docker compose restart SERVICE
docker compose stop
docker compose start
docker compose down
docker compose down -v
docker compose pull
```

39. Главное различие команд

Команда	Назначение
`docker build`	Собирает образ.
`docker run`	Создаёт и запускает новый контейнер.
`docker start`	Запускает уже существующий контейнер.
`docker exec`	Запускает дополнительную команду внутри работающего контейнера.
`docker stop`	Останавливает контейнер.
`docker rm`	Удаляет контейнер.
`docker rmi`	Удаляет образ.
`docker compose build`	Собирает образы сервисов.
`docker compose up`	Создаёт и запускает Compose-проект.

Команда	Назначение
`docker compose stop`	Останавливает, но сохраняет контейнеры.
`docker compose down`	Останавливает и удаляет контейнеры и сети проекта.
`docker compose exec`	Выполняет команду в запущенном сервисе.
`docker compose run`	Создаёт отдельный временный контейнер сервиса.

40. Практическая формула работы

Для большинства проектов достаточно следующего цикла:

```
docker compose config
docker compose up -d --build
docker compose ps
docker compose logs -f
```

После изменения исходного кода:

```
docker compose up -d --build
```

Для диагностики:

```
docker compose ps -a
```


```
docker compose logs --tail 200 SERVICE
docker compose exec SERVICE sh
```

Для остановки:

```
docker compose down
```

Для полного сброса, включая данные:

```
docker compose down -v
```

Команду `docker compose down -v` следует использовать только тогда, когда данные в volumes действительно больше не нужны.