Учебник по Docker для начинающих

Полное руководство по Docker и Docker Compose для новичков

Создано для изучения контейнеризации

Содержание

1 Что такое Docker?

Docker — это платформа для контейнеризации, которая позволяет упаковать приложение и все его зависимости (библиотеки, настройки, окружение) в контейнер. Контейнеры — это изолированные, легковесные среды, которые работают одинаково на любом компьютере. Представьте контейнер как чемодан: всё, что нужно для работы приложения, упаковано внутри, и вы можете взять его куда угодно.

1.1 Зачем нужен Docker?

- **Консистентность**: Приложение работает одинаково на ноутбуке, сервере или в облаке.
- **Экономия ресурсов**: Контейнеры используют меньше ресурсов, чем виртуальные машины.
- **Упрощение разработки**: Не нужно настраивать окружение вручную.
- **Масштабируемость**: Легко запускать несколько копий приложения.

2 Базовая терминология

Чтобы начать работать с Docker, нужно понять основные понятия:

Образ (Image) Шаблон, содержащий приложение, библиотеки и настройки. Образ — это как рецепт торта, который нельзя изменить. Пример: `nginx:latest`.

Контейнер (Container) Запущенный экземпляр образа. Если образ — рецепт, то контейнер — готовый торт. Контейнеры изолированы друг от друга.

Dockerfile Текстовый файл с инструкциями для создания образа. Это как поваренная книга для Docker.

Реестр (Registry) Хранилище образов, например, Docker Hub (`https://hub.docker.com`). Это как библиотека рецептов.

Тег (Tag) Метка версии образа, например, `python:3.9-slim`.

Docker Compose Инструмент для управления несколькими контейнерами через YAML-файл. Это как дирижёр, управляющий оркестром.

Том (Volume) Способ хранения данных вне контейнера, чтобы они не исчезли при его удалении. Это как внешний жёсткий диск.

Сеть (Network) Механизм для связи между контейнерами или с внешним миром. Это как Wi-Fi для контейнеров.

Зачем это нужно? Тома сохраняют данные (например, базу данных), сети позволяют контейнерам обмениваться информацией, а реестры упрощают доступ к готовым образам.

3 Создание образов с помощью Dockerfile

Dockerfile — это текстовый файл с инструкциями для сборки Docker-образа. Каждая инструкция создаёт слой, что делает сборку быстрой и эффективной за счёт кэширования.

3.1 Основные инструкции Dockerfile

FROM Указывает базовый образ, с которого начинается сборка. Это фундамент вашего образа.

RUN Выполняет команды во время сборки, например, установка пакетов.

COPY Копирует файлы или папки в образ.

ADD Похоже на `COPY`, но поддерживает архивы и URL (используйте с осторожностью).

ENV Задаёт переменные окружения, доступные в контейнере.

ARG Определяет аргументы, доступные только во время сборки.

WORKDIR Устанавливает рабочую директорию для последующих команд.

EXPOSE Документирует порты, которые использует контейнер.

CMD Задаёт команду по умолчанию для контейнера. Может быть переопределена.

ENTRYPOINT Определяет исполняемый файл, который всегда запускается.

USER Указывает пользователя для выполнения команд (для безопасности).

VOLUME Создаёт точку монтирования для тома.

3.2 Простой пример Dockerfile

```
# Используем базовый образ Python
FROM python:3.9-slim

# Устанавливаем рабочую директорию
WORKDIR /app

# Копируем файл зависимостей
COPY requirements.txt .

# Устанавливаем зависимости
RUN pip install --no-cache-dir -r requirements.txt

# Копируем приложение
COPY . .

# Задаём переменную окружения
ENV PORT=8000

# Указываем порт
EXPOSE 8000

# Запускаем приложение
CMD ["python", "app.py"]
```

Разбор:

- `FROM python:3.9-slim`: Используем минимальный образ Python.
- `WORKDIR /app`: Все команды выполняются в папке `/app`.

- `COPY requirements.txt .`: Копируем файл зависимостей отдельно для кэширования.
- `RUN pip install`: Устанавливаем зависимости, `–no-cache-dir` уменьшает размер.
- `EXPOSE 8000`: Документируем порт.
- `CMD ["python "app.py"]`: Запускаем приложение.

Команды для сборки и запуска:

- `docker build -t myapp:1.0 .`: Собираем образ с тегом `myapp:1.0`.
- `docker run -d -p 8000:8000 myapp:1.0`: Запускаем контейнер, связывая порты.

Что происходит? Docker создаёт образ, слой за слоем, кэшируя неизменённые части. При запуске контейнер исполняет `python app.py` и становится доступен на порту 8000.

3.3 Сложный пример Dockerfile

Dockerfile для Node.js приложения с пользователем и оптимизациями:

```
# Используем базовый образ Ubuntu
FROM ubuntu:20.04

# Устанавливаем аргумент сборки
ARG APP_VERSION=1.0.0

# Устанавливаем переменные окружения
ENV NODE_ENV=production
ENV PORT=3000
ENV APP_VERSION=$APP_VERSION

# Устанавливаем зависимости
RUN apt-get update && \
    apt-get install -y nodejs npm && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Создаём пользователя
RUN useradd -m appuser

# Устанавливаем рабочую директорию
WORKDIR /home/appuser/app

# Копируем package.json и устанавливаем зависимости
COPY package.json package-lock.json ./
RUN npm ci --production

# Копируем остальной код
COPY . .

# Меняем владельца файлов
RUN chown -R appuser:appuser /home/appuser/app

# Переключаемся на пользователя
USER appuser

# Указываем порт
```

```
EXPOSE 3000

# Определяем точку монтирования
VOLUME /home/appuser/app/logs

# Задаём точку входа
ENTRYPOINT ["node"]

# Задаём команду по умолчанию
CMD ["server.js"]
```

Разбор:

- `ARG APP_VERSION=1.0.0`: Определяет аргумент `APP_VERSION`, который можно передать при сборке с помощью флага `-build-arg APP_VERSION=2.0`.
- `ENV NODE_ENV=production`: Задаёт переменную окружения `NODE_ENV`, указывающую Node.js работать в продакшен-режиме для оптимизации.
- `ENV PORT=3000`: Указывает порт, на котором приложение будет доступно.
- `ENV APP_VERSION=$APP_VERSION`: Использует значение аргумента `APP_VERSION` для задания переменной окружения.
- `RUN apt-get update && ...`: Устанавливает Node.js и npm, затем очищает кэш пакетов, чтобы уменьшить размер образа.
- `RUN useradd -m appuser`: Создаёт не-root пользователя `appuser` для повышения безопасности.
- `WORKDIR /home/appuser/app`: Устанавливает рабочую директорию для всех последующих команд.
- `COPY package.json package-lock.json ./`: Копирует файлы зависимостей отдельно для использования кэширования слоёв.
- `RUN npm ci -production`: Устанавливает зависимости, используя точные версии из `package-lock.json`, только для продакшена.
- `COPY . .`: Копирует остальной код приложения в образ.
- `RUN chown -R appuser:appuser`: Назначает владельцем файлов пользователя `appuser`.
- `USER appuser`: Переключает контекст на не-root пользователя для всех последующих команд.
- `EXPOSE 3000`: Документирует, что контейнер использует порт 3000.
- `VOLUME /home/appuser/app/logs`: Создаёт точку монтирования для хранения логов вне контейнера.
- `ENTRYPOINT ["node"]`: Указывает, что node всегда запускается как исполняемый файл.
- `CMD ["server.js"]`: Задаёт файл `server.js` как команду по умолчанию для запуска.

Команды:

- `docker build -t nodeapp:1.0 .`: Собирает образ с тегом `nodeapp:1.0`.
- `docker run -d -p 3000:3000 -v logs:/home/appuser/app/logs nodeapp:1.0`: Запускает контейнер с томом для логов, связывая порт 3000.

Что происходит? Образ собирается с минимальным размером за счёт очистки кэша и использования не-root пользователя. Приложение запускается от имени `appuser`, а логи сохраняются в томе.

3.4 Многоэтапная сборка (Multi-Stage Build)

Многоэтапная сборка позволяет создавать компактные образы, разделяя процесс сборки и выполнения. Пример: сборка React-приложения.

```
# Этап 1: Сборка приложения
FROM node:16 AS builder
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

# Этап 2: Финальный образ
FROM nginx:alpine
COPY --from=builder /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Разбор:

- `FROM node:16 AS builder`: Первый этап использует образ Node.js для сборки React-приложения.
- `RUN npm run build`: Создаёт статические файлы приложения.
- `FROM nginx:alpine`: Второй этап использует лёгкий образ Nginx для сервировки файлов.
- `COPY –from=builder`: Копирует только результат сборки из первого этапа.

Преимущества:

- Финальный образ минимален, так как не содержит Node.js и зависимости.
- Повышенная безопасность за счёт исключения лишних инструментов.

Команды:

- `docker build -t reactapp:1.0 .`: Собирает образ.
- `docker run -d -p 80:80 reactapp:1.0`: Запускает Nginx-сервер.

Что происходит? Первый этап создаёт статические файлы, второй — сервирует их через Nginx. Образ содержит только необходимые файлы, что делает его компактным.

4 Docker Compose: управление многоконтейнерными приложениями

Docker Compose позволяет управлять несколькими контейнерами через один YAML-файл (`docker-compose.yml`). Это упрощает запуск сложных приложений, таких как веб-сервер с базой данных.

4.1 Зачем нужен Docker Compose?

- **Упрощение**: Один файл заменяет множество команд `docker run`.
- **Сети**: Автоматически создаёт сети для связи контейнеров.
- **Консистентность**: Конфигурация переносима между средами.
- **Разработка**: Удобен для локального тестирования.

Пример: Без Compose вы вручную запускаете контейнеры для WordPress и MySQL, настраиваете порты и сети. Compose делает это одной командой.

4.2 Структура docker-compose.yml

Файл использует YAML-формат. Основные ключи:

version Версия синтаксиса (рекомендуется `"3.8"`).

services Описывает контейнеры (сервисы).

networks Настраивает сети.

volumes Определяет тома для данных.

build Указывает сборку образа.

environment Переменные окружения.

depends_on Зависимости между сервисами.

healthcheck Проверка состояния сервиса.

deploy Настройки ресурсов (CPU, память).

restart Политика перезапуска (`always`, `unless-stopped`).

secrets Управление секретными данными.

4.3 Простой пример Docker Compose

```yaml
version: "3.8"
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - web-data:/usr/share/nginx/html
    networks:
      - app-network
  db:
    image: mysql:8.0
    environment:
      - MYSQL_ROOT_PASSWORD=secret
      - MYSQL_DATABASE=mydb
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - app-network
volumes:
```

```
  web-data:
  db-data:
networks:
  app-network:
    driver: bridge
```

Разбор:

- `services`: Два сервиса — web (Nginx) и db (MySQL).
- `ports`: Порт 80 хоста связан с портом 80 контейнера.
- `environment`: Задаёт пароль и имя базы данных для MySQL.
- `volumes`: Именованные тома для сохранения данных Nginx и MySQL.
- `networks`: Сеть `app-network` для связи сервисов.

Команды:

- `docker-compose up -d`: Запускает оба сервиса в фоновом режиме.
- `docker-compose down`: Останавливает и удаляет контейнеры, сети и тома (если не указан `-volumes`, тома сохраняются).

Что происходит? Compose создаёт сеть `app-network`, запускает MySQL и Nginx, настраивает тома и порты. Nginx доступен на `localhost:80`.

4.4 Сложный пример Docker Compose

Пример приложения с веб-приложением (Python Flask), базой данных (PostgreSQL) и кэшем (Redis).

```yaml
version: "3.8"
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    environment:
      - FLASK_ENV=production
      - DATABASE_URL=postgresql://user:password@db:5432/myapp
      - REDIS_URL=redis://redis:6379/0
    depends_on:
      db:
        condition: service_healthy
      redis:
        condition: service_started
    volumes:
      - app-logs:/app/logs
    networks:
      - backend
    deploy:
      resources:
        limits:
          cpus: "0.5"
          memory: 512M
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:5000/health"]
      interval: 30s
```

```
      timeout: 10s
      retries: 3
    restart: unless-stopped
  db:
    image: postgres:13
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=myapp
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user"]
      interval: 10s
      timeout: 5s
      retries: 5
  redis:
    image: redis:6.2
    volumes:
      - redis-data:/data
    networks:
      - backend
    restart: always
volumes:
  app-logs:
  db-data:
  redis-data:
networks:
  backend:
    driver: bridge
```

Разбор:

- `build`: Сервис app собирается из локального файла `Dockerfile`, указанного в поле `dockerfile`. Это позволяет создать образ для Flask-приложения.
- `depends_on`: Сервис app ожидает, пока:
  - db достигнет состояния `service_healthy` (база данных готова к работе).
  - `redis` достигнет состояния `service_started` (сервис запущен).
- `environment`: Задаёт переменные окружения:
  - `FLASK_ENV=production`: Указывает Flask работать в продакшен-режиме.
  - `DATABASE_URL`: Строка подключения к PostgreSQL, где db — имя сервиса, используемое как хост.
  - `REDIS_URL`: Строка подключения к Redis, где `redis` — имя сервиса.
- `healthcheck`: Настраивает проверки состояния:
  - Для app: Проверяет доступность эндпоинта `/health` с помощью `curl`.
  - Для db: Проверяет готовность PostgreSQL с помощью `pg_isready`.
- `deploy`: Ограничивает ресурсы для app: максимум 0.5 CPU и 512 МБ памяти.
- `restart`: Политики перезапуска:
  - app: Перезапускается, если не остановлен вручную (`unless-stopped`).

  - `redis`: Перезапускается всегда (`always`).
- `volumes`: Создаёт тома для:
  - Логов приложения (`app-logs`).
  - Данных PostgreSQL (`db-data`).
  - Данных Redis (`redis-data`).
- `networks`: Все сервисы используют сеть `backend` с драйвером `bridge` для общения.

Команды:

- `docker-compose build`: Собирает образ для `app` на основе `Dockerfile`.
- `docker-compose up -d`: Запускает все сервисы в фоновом режиме.
- `docker-compose logs app`: Показывает логи сервиса `app`.
- `docker-compose down -volumes`: Удаляет контейнеры, сети и тома.

Что происходит? Compose создаёт сеть `backend`, запускает `db` и `redis`, собирает `app`, настраивает связи, тома и порты. Приложение доступно на `localhost:5000`.

4.5 Дополнительные возможности Docker Compose

- **Переменные окружения**: Используйте файл `.env` для хранения конфигурации:

```
DATABASE_URL=postgresql://user:password@db:5432/myapp
REDIS_URL=redis://redis:6379/0

```

- **Масштабирование**: Запустите несколько копий сервиса:

```
docker-compose up --scale app=3

```

- **Секреты**: Храните пароли безопасно:

```
secrets:
  db_password:
    file: ./db_password.txt

```

- **Перезапуск**: Настройте политику перезапуска:

```
restart: unless-stopped

```

5 Команды Docker

Команды Docker управляют образами, контейнерами, сетями и томами. Ниже приведён список с флагами и примерами, связанными с примерами выше.

5.1 Основные команды

docker build Собирает образ из Dockerfile.

```
    docker build -t nodeapp:1.0 --build-arg APP_VERSION=1.0.0 .

```

Флаги:

- `-t`: Задаёт имя и тег образа (например, `nodeapp:1.0`).
- `–build-arg`: Передаёт аргументы в `ARG` (например, `APP_VERSION`).
- `–no-cache`: Игнорирует кэш сборки для полной пересборки.

Зачем? Создаёт образ, например, для Node.js приложения из раздела 4.3. **Связь с примером:** Используется для сборки `nodeapp:1.0` из сложного Dockerfile.

docker run Запускает контейнер.

```
    docker run -d -p 8080:80 --name myapp -v logs:/app/logs --restart=
    always myapp:1.0

```

Флаги:

- `-d`: Запускает контейнер в фоновом режиме.
- `-p`: Сопоставляет порты хоста и контейнера (например, `8080:80`).
- `–name`: Задаёт имя контейнера.
- `-v`: Подключает том (например, `logs:/app/logs`).
- `–restart`: Политика перезапуска (`always`, `unless-stopped`).

Зачем? Запускает контейнер, например, веб-сервер из раздела 4.2. **Связь с примером:** Используется для запуска `myapp:1.0` с портом 8000.

docker ps Показывает запущенные контейнеры.

```
    docker ps -a -q

```

Флаги:

- `-a`: Показывает все контейнеры, включая остановленные.
- `-q`: Выводит только ID контейнеров.

Зачем? Проверяет, какие контейнеры работают, например, после `docker run`.

docker stop Останавливает контейнер.

```
    docker stop myapp

```

Зачем? Грациозно останавливает контейнер, например, `myapp`.

docker rm Удаляет контейнер.

```
docker rm -f myapp

```

Флаги:

- `-f`: Принудительно удаляет работающий контейнер.

Зачем? Освобождает ресурсы после остановки.

docker images Список образов.

```
docker images -q

```

Флаги:

- `-q`: Выводит только ID образов.

Зачем? Показывает доступные образы, например, `nodeapp:1.0`.

docker rmi Удаляет образ.

```
docker rmi -f myapp:1.0

```

Флаги:

- `-f`: Принудительно удаляет образ, даже если он используется.

Зачем? Удаляет ненужные образы, освобождая место.

docker pull Загружает образ из реестра.

```
docker pull nginx:latest

```

Зачем? Получает образ, например, `nginx:latest` для Compose-примера.
Связь с примером: Используется для загрузки `nginx:latest` в разделе 5.3.

docker exec Выполняет команду в работающем контейнере.

```
docker exec -it myapp bash

```

Флаги:

- `-it`: Запускает интерактивный терминал.

Зачем? Для отладки или выполнения команд внутри контейнера.

docker logs Показывает логи контейнера.

```
docker logs --tail=10 myapp

```

Флаги:

- `–tail`: Ограничивает количество строк (например, 10).
- `-f`: Следит за логами в реальном времени.

Зачем? Для диагностики проблем в контейнере.

6 Команды Docker Compose

Команды Compose управляют многоконтейнерными приложениями, описанными в `docker-compose.yml`.

docker-compose up Запускает все сервисы.

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

```

Флаги:

- `-d`: Запускает в фоновом режиме.
- `–build`: Пересобирает образы перед запуском.
- `–scale app=3`: Запускает 3 экземпляра сервиса `app`.

Зачем? Запускает приложение, например, из раздела 5.4. **Связь с примером:** Используется для запуска Flask-приложения, PostgreSQL и Redis.

docker-compose down Останавливает и удаляет сервисы.

```
    docker-compose down --volumes --rmi all

```

Флаги:

- `–volumes`: Удаляет именованные тома.
- `–rmi all`: Удаляет все образы, созданные Compose.

Зачем? Очищает ресурсы после работы. **Связь с примером:** Удаляет сервисы и тома из раздела 5.4.

docker-compose ps Показывает статус сервисов.

```
    docker-compose ps -q

```

Флаги:

- `-q`: Выводит только ID контейнеров.

Зачем? Проверяет, какие сервисы работают.

docker-compose logs Показывает логи сервисов.

```
    docker-compose logs --tail=10 app

```

Флаги:

- `–tail`: Ограничивает количество строк.
- `-f`: Следит за логами в реальном времени.

Зачем? Для отладки, например, проверки логов `app` из раздела 5.4.

docker-compose build Собирает образы для сервисов.

```
    docker-compose build --no-cache

```

Флаги:

- `–no-cache`: Игнорирует кэш сборки.

Зачем? Создаёт образы, например, для app из раздела 5.4. **Связь с примером:** Собирает образ Flask-приложения.

docker-compose restart Перезапускает сервисы.

```
    docker-compose restart app

```

Зачем? Перезапускает указанный сервис, например, app.

docker-compose pause Приостанавливает сервисы.

```
    docker-compose pause app

```

Зачем? Временно приостанавливает сервис без остановки.

docker-compose unpause Возобновляет приостановленные сервисы.

```
    docker-compose unpause app

```

Зачем? Возобновляет работу сервиса.

7 Полезные советы для новичков

- **Кэширование**: Размещайте `COPY` кода в конце Dockerfile, чтобы использовать кэш для неизменённых слоёв.
- **Очистка**: Используйте `docker system prune -a` для удаления неиспользуемых контейнеров, образов и сетей.
- **Безопасность**: Всегда используйте `USER` для запуска от не-root пользователя.
- **Диагностика**: Используйте `docker inspect <container>` для получения подробной информации о контейнере.
- **Документация**: Изучайте официальные ресурсы: https://docs.docker.com и https://docs.docker.com/compose/.