Подсказки по Docker

Ссылка на документацию https://docs.docker.com/build/concepts/dockerfile

Скачать PDF

Скачать инструкцию в PDF

Базовое создание Dockerfile для Python-приложения и запуск контейнера

1. Сердце контейнеризации: Создание Dockerfile

Итак, у нас есть "стройматериалы" (наше python-приложение) и "завод" (установленный Docker). Теперь нам нужен "чертеж", по которому завод будет собирать наш продукт. В мире Docker этот чертеж называется Dockerfile.

Что такое Dockerfile?

Это простой текстовый файл. В нем мы по шагам, команда за командой, описываем, как собрать образ нашего приложения. Думайте об этом как о рецепте: возьмите то, добавьте это, запустите вот так.

Прямо в корне нашей папки my-python-app, рядом с app.py и requirements.txt, создайте файл с именем Dockerfile.

IMPORTANT	У файла не должно быть расширения. Просто Dockerfile.

А теперь давайте напишем наш "рецепт" шаг за шагом. Откройте этот файл и начнем добавлять в него строки.

Шаг 1: Выбираем фундамент (FROM)

Используем официальный образ Python как основу

```
FROM python:3.11-slim
```

Каждый образ Docker строится на основе другого, "родительского" образа. Команда FROM — это всегда первая строчка в Dockerfile. Здесь мы говорим: "Взять за основу официальный образ с уже установленным Python версии 3.11".

Почему slim? Это "облегченная" версия образа. В ней нет лишних пакетов и документации, которые нам не нужны для запуска приложения. В результате наш итоговый образ будет весить значительно меньше. Золотое правило: всегда старайтесь делать образы как можно меньше.

Шаг 2 : Указываем рабочую папку (WORKDIR)

Устанавливаем рабочую директорию внутри контейнера

```
WORKDIR /app
```

Команда WORKDIR делает две вещи: создает директорию /app внутри нашего будущего контейнера (если ее нет) и делает ее текущей. Это значит, что все последующие команды (COPY, RUN и т.д.) будут выполняться из этой папки. Это хорошая практика, чтобы не разбрасывать файлы нашего приложения по всей файловой системе контейнера.

Шаг 3: Устанавливаем зависимости (COPY, RUN)

Копируем файл с зависимостями и устанавливаем их

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

А вот и первый хитрый момент, который отличает хороший Dockerfile от плохого. Казалось бы, почему не скопировать сразу все файлы проекта?

Дело в том, как Docker кэширует слои. Каждый RUN, COPY, ADD создает новый слой. Когда вы пересобираете образ, Docker проверяет, изменилась ли команда или файлы, которые она использует. Если нет — он берет готовый слой из кэша.

Зависимости в requirements.txt меняются редко, а вот наш код в app.py — постоянно. Разделяя эти шаги, мы добиваемся того, что Docker будет заново устанавливать все библиотеки (долгий процесс) только в том случае, если мы изменим файл requirements.txt. Если же мы поменяем только код в app.py, Docker возьмет уже готовый слой с установленными библиотеками из кэша, и сборка пройдет за секунды.

--no-cache-dir — полезный флаг, который говорит pip не сохранять кэш скачанных пакетов. Это еще один способ уменьшить итоговый размер образа.

Шаг 4: Копируем код нашего приложения (COPY)

Копируем весь остальной код приложения


```
COPY . .
```

Теперь, когда зависимости установлены, мы копируем все остальные файлы из текущей директории (первая точка) в рабочую директорию /app внутри контейнера (вторая точка).

Шаг 5: Сообщаем порт (EXPOSE)

"Сообщаем" Docker, что наше приложение будет работать на порту 5000

```
EXPOSE 5000
```

Команда EXPOSE — это, по сути, документация. Она не открывает порт по-настоящему, но сообщает Docker (и человеку, который будет этот образ использовать), что приложение внутри контейнера ожидает подключения на порту 5000.

Шаг 6: Указываем команду для запуска (CMD)

Указываем команду, которая запустится при старте контейнера

```
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

Это финальный аккорд. Команда CMD определяет, что будет выполнено при запуске контейнера. Мы запускаем наш gunicorn, приказываем ему слушать на всех интерфейсах (0.0.0.0) на порту 5000 и указываем, что запускать нужно объект app из модуля app (app.py).

Итоговый рецепт

Вот как должен выглядеть ваш Dockerfile целиком:

```
# 1. Используем официальный образ Python как основу
FROM python:3.11-slim

# 2. Устанавливаем рабочую директорию внутри контейнера
WORKDIR /app

# 3. Копируем файл с зависимостями и устанавливаем их (используем кэш)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 4. Копируем весь остальной код приложения
COPY . .

# 5. "Сообщаем" Docker, что наше приложение будет работать на порту 5000
EXPOSE 5000

# 6. Указываем команду, которая запустится при старте контейнера
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

2. Сборка и запуск: Оживляем наш контейнер

Все подготовительные работы завершены. Сейчас мы выполним всего две команды в терминале, чтобы собрать и запустить наше приложение.

Убедитесь, что вы находитесь в терминале (командной строке) внутри вашей папки my-python-app, там же, где лежат все наши файлы.

Шаг 1: Собираем образ с помощью docker build

Эта команда читает наш Dockerfile и выполняет все инструкции по очереди, создавая в итоге готовый образ.

Выполните в терминале:

```
docker build -t my-python-app .
```

Давайте разберем, что здесь происходит:

- docker build: Это основная команда для сборки образа.
- -t my-python-app: Флаг -t (от "tag") позволяет нам присвоить образу имя и, опционально, тег. Мы назвали наш образ my-python-app. Это гораздо удобнее, чем работать со случайным ID, который Docker присвоил бы ему по умолчанию.
- . (точка в конце): Это очень важная часть. Точка указывает на контекст сборки. Она говорит Docker: "Ищи Dockerfile в текущей директории (.) и отсюда же бери все файлы для копирования в образ (например, app.py и requirements.txt)".

После запуска команды вы увидите, как Docker поочередно выполняет каждый шаг из нашего Dockerfile: FROM, WORKDIR, COPY, RUN... Если все прошло успешно, в конце вы увидите сообщение о том, что образ собран.

Чтобы убедиться, что образ действительно создался, можно выполнить команду:


```
docker images
```

Вы должны увидеть в списке свежесозданный образ my-python-app.

Шаг 2: Запускаем контейнер из образа с помощью docker run

Образ — это, по сути, шаблон, выключенный "слепок" нашего приложения. Чтобы он ожил, из него нужно запустить контейнер. Контейнер — это работающий экземпляр образа.

Выполните эту команду:

```
docker run -d -p 8080:5000 --name my-running-app my-python-app
```

Эта команда выглядит сложнее, но тут все логично:

- docker run: Основная команда для запуска контейнера.
- -d: От "detached". Этот флаг запускает контейнер в фоновом режиме. Ваш терминал сразу освободится, а контейнер продолжит работать "за сценой".
- -p 8080:5000: Флаг -p (от "publish") — ключевой момент. Он "пробрасывает" порт изнутри контейнера наружу, на нашу машину. Мы связываем порт 8080 на нашем компьютере (хост-машине) с портом 5000 внутри контейнера. Именно на 5000-м порту слушает наше приложение (gunicorn).
- --name my-running-app: Мы даем нашему работающему контейнеру человеческое имя my-running-app. Это поможет нам в дальнейшем им управлять (останавливать, смотреть логи и т.д.).
- my-python-app: В конце мы указываем имя образа, из которого мы хотим создать контейнер.

Шаг 3: Проверяем результат!

Самый приятный момент. Откройте ваш любимый браузер и перейдите по адресу:

http://localhost:8080

Если вы все сделали правильно — ваше приложение заработает.

Поздравляю! Вы только что упаковали Python-приложение в контейнер и запустили его. Ваш код теперь работает в полностью изолированной среде, и вы можете быть уверены, что он точно так же запустится на любой другой машине, где есть Docker.

3. Управление контейнерами: Основные команды

Наше приложение работает в фоновом режиме, но оно не превратилось в неуправляемый черный ящик. У нас есть полный контроль над его жизненным циклом. Вот базовый "джентльменский набор" команд, который вы будете использовать постоянно.

1. Посмотреть, что запущено: docker ps

Эта команда — ваш главный инструмент мониторинга. Она показывает список всех активных контейнеров.

Выполните в терминале:

```
docker ps
```

Вы увидите аккуратную табличку с информацией о вашем контейнере: * его ID, из какого образа он создан (my-python-app), * статус (Up for...), * информацию о портах (0.0.0.0:8080□5000/tcp), * его имя (my-running-app).

Лайфхак: Если хотите увидеть вообще все контейнеры, включая те, что были остановлены, добавьте флаг -a: docker ps -a.

2. Заглянуть внутрь: docker logs

Что, если в приложении произошла ошибка? Или вы просто хотите посмотреть логи gunicorn, чтобы увидеть входящие запросы? Для этого есть команда docker logs.

```
docker logs my-running-app
```

Она выведет в терминал все, что ваше приложение напечатало с момента запуска. Это невероятно полезно для отладки.

Лайфхак: Чтобы следить за логами в реальном времени (как команда tail -f в Linux), добавьте флаг -f: docker logs -f my-running-app. Нажмите Ctrl+C, чтобы выйти из этого режима.

3. Остановить контейнер: docker stop

Когда вам нужно выключить ваше приложение, используйте команду docker stop. Она аккуратно посылает контейнеру сигнал о завершении работы, давая ему время корректно остановиться.

```
docker stop my-running-app
```

Терминал вернет вам имя остановленного контейнера. Если вы теперь снова выполните docker ps, список будет пуст. А вот docker ps -a покажет ваш контейнер со статусом Exited.

4. Запустить остановленный контейнер: docker start

Остановленный контейнер никуда не удаляется. Его можно запустить снова той же командой, что и заводится автомобиль:

```
docker start my-running-app
```

Он запустится с теми же параметрами, с которыми был создан изначально (с тем же пробросом портов и т.д.).

5. Удалить контейнер: docker rm

Когда контейнер вам больше не нужен и вы просто хотите освободить место, его можно удалить.

IMPORTANT | Удалить можно только остановленный контейнер.

```
# Сначала останавливаем, если он еще работает
docker stop my-running-app

# Затем удаляем
docker rm my-running-app
```

После этого контейнер исчезнет навсегда. Обратите внимание: удаляется только контейнер, а не образ my-python-

app. Вы в любой момент можете создать новый контейнер из этого образа с помощью команды docker run.

Вот и все! Этих пяти команд — **ps, logs, stop, start, rm** — вам хватит для 90% повседневных задач по управлению контейнерами. Теперь вы не просто создали контейнер, но и умеете им управлять.

6. Лучшие практики и что дальше?

Поздравляю! Вы прошли путь от простого Python-скрипта до работающего, изолированного контейнера. Это уже огромный шаг. Но, как и в любом ремесле, здесь есть свои хитрости, которые отличают новичка от профессионала.

Немного о лучших практиках Вот пара простых советов, которые сразу сделают ваши Docker-образы лучше, меньше и безопаснее.

1. Используйте .dockerignore

Когда вы выполняете команду docker build ., Docker сначала запаковывает весь "контекст" (все файлы и папки в текущей директории) и отправляет его своему движку. В этот архив попадает всё: виртуальное окружение .venv, кэш Python *pycache*, папки .git и другие служебные файлы, которые вашему приложению в контейнере абсолютно не нужны.

Чтобы этого избежать, создайте в корне проекта файл .dockerignore (по аналогии с .gitignore). Все, что вы в нем перечислите, будет проигнорировано при сборке.

Пример хорошего .dockerignore для Python-проекта:

```
# Исключаем виртуальное окружение
.venv
```


```
venv
env

# Исключаем кэш Python
__pycache__/
*.pyc
*.pyo

# Исключаем служебные файлы IDE и ОС
.idea/
.vscode/
.DS_Store

# Исключаем Git
.git
.gitignore
```

Результат: Контекст сборки становится меньше (сборка быстрее), а итоговый образ не содержит мусора.

2. Не работайте под root

По умолчанию все процессы внутри контейнера запускаются от имени суперпользователя (root). С точки зрения безопасности — это плохая идея. Если злоумышленник найдет уязвимость в вашем приложении, он получит права root внутри контейнера, что может привести к серьезным последствиям.

Правильный подход — создать отдельного, непривилегированного пользователя и запускать приложение от его имени.

Добавьте эти строчки в ваш Dockerfile перед командой CMD:

```
# ... после COPY . .

# Создаем пользователя appuser
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

# Переключаемся на этого пользователя
USER appuser

# Указываем команду для запуска (она выполнится от имени appuser)
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

Это простой шаг, который значительно повышает безопасность вашего контейнера.

Что дальше? Ваш путь в мире Docker

Docker — это целая вселенная. Вы сделали первый и самый важный шаг, но впереди еще много интересного. Вот что я настоятельно рекомендую изучить дальше:

Docker Compose: дирижер для вашего оркестра

Наше приложение пока одиноко. Но что, если ему понадобится база данных (например, PostgreSQL) или кэш (Redis)? Запускать каждый контейнер отдельной docker run командой с кучей параметров — путь в никуда.

Здесь на сцену выходит Docker Compose. Это инструмент, который позволяет в простом YAML-файле описать всю связку сервисов вашего приложения: веб-сервер, базу данных, фоновые обработчики и т.д. Одной командой docker-

compose up вы поднимаете все это хозяйство, и сервисы могут общаться друг с другом по сети, которую Docker Compose создает автоматически.

Это логичный следующий шаг в вашем обучении. Умение работать с Docker Compose — это уже стандарт де-факто для локальной разработки в большинстве компаний.

Container Registry: ваш личный склад образов

Сейчас ваши образы хранятся только на вашем компьютере. Чтобы использовать их на сервере или поделиться с коллегой, их нужно куда-то загрузить. Для этого существуют реестры контейнеров. Самый известный — Docker Hub. Вы можете бесплатно хранить там свои публичные образы (и несколько приватных) и скачивать их где угодно с помощью команды docker pull.

Полезная статья об оптимизации Dockerfile

Ссылка на оригинал статьи

Конспект статьи: это разбор эволюции Dockerfile для Go-приложения от «лишь бы запустилось» до production-ready multi-stage образа.

1. Плохой стартовый Dockerfile

Пример начинается с простого варианта:

```
FROM golang:latest
COPY . .
```


```
RUN go mod tidy && go build .
CMD ["./myapp"]
```

Проблемы:

- `golang:latest` делает сборку невоспроизводимой: завтра `latest` может указывать на другую версию Go.
- `COPY . .` тащит в образ всё: `.git`, README, тесты, временные файлы, потенциальные секреты.
- `go mod tidy` внутри Dockerfile плохо, потому что может менять `go.mod` и `go.sum`.
- Образ получается тяжёлый: в статье указан пример около `1.3 ГБ`.
- Сборка медленная: кеш слоёв часто инвалидируется.
- Нет `LABEL`, поэтому непонятно, кто поддерживает образ и что это за версия.
- Контейнер запускается от `root`, что плохо для безопасности.

2. Первые улучшения

Что добавляют:

```
FROM golang:1.25.7-alpine
LABEL ...
WORKDIR /opt/app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build .
```


```
CMD ["./myapp"]
```

Ключевые идеи:

- фиксируем версию базового образа вместо `latest`;
- используем `alpine`, чтобы уменьшить размер;
- добавляем `LABEL maintainer`, `version`, `description`;
- задаём `WORKDIR`, чтобы не работать в корне файловой системы;
- сначала копируем только `go.mod` и `go.sum`, затем делаем `go mod download` — так зависимости лучше кешируются;
- `go mod download` предпочтительнее `go mod tidy`, потому что не меняет манифесты зависимостей;
- добавляем `.dockerignore`, минимум:

```
.git
README.md
```

Но этого ещё мало: в финальном образе всё ещё остаются исходники, кеш, Go-компилятор, тесты и возможные секреты; контейнер всё ещё работает не от отдельного пользователя.

3. Multi-stage build

Дальше сборку и запуск разделяют на две стадии:

- `builder` на базе `golang:1.25.7-alpine`;

- финальный runtime-образ на базе `alpine:3.21`.

```
# Stage 1: сборка зависимостей.
FROM golang:1.25.7-alpine AS builder

WORKDIR /build

COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o /build/myapp .

# Stage 2: финальный образ.
FROM alpine:3.21

LABEL maintainer="andrey@express42.com" \
      version="0.2" \
      description="Backend API"

RUN apk add --no-cache postgresql17-client

WORKDIR /opt/app

ENV API_TOKEN="super-secret-token-123"

RUN addgroup -S appuser --gid 2100 &amp;amp;&amp;amp; \
    adduser -S appuser --uid 2101 -G appuser

COPY --chown=appuser:appuser --from=builder /build/myapp .
```


```
USER appuser:appuser

ENTRYPOINT ["./myapp"]
```

На стадии сборки:

- `WORKDIR /build`;
- копируются `go.mod` и `go.sum`;
- выполняется `go mod download`;
- копируется код;
- бинарник собирается командой с параметрами:

```
CGO_ENABLED=0 GOOS=linux go build -o /build/myapp .
```

Смысл параметров:

- `GOOS=linux` явно собирает бинарник под Linux;
- `CGO_ENABLED=0` делает Go-бинарник независимым от системных C-библиотек;
- `-o myapp` фиксирует имя бинарника, чтобы оно не зависело от имени модуля.

В финальном образе:

- нет Go-компилятора;

- копируется только готовый бинарник;
- добавляется runtime-зависимость `postgresql17-client`, потому что приложению нужен `psql` для миграций;
- используется `apk add --no-cache`, чтобы не раздувать образ кешем пакетного менеджера;
- создаётся отдельный пользователь и группа:

```
addgroup -S appuser --gid 2100
adduser -S appuser --uid 2101 -G appuser
```

- бинарник копируется с владельцем:

```
COPY --chown=appuser:appuser --from=builder /build/myapp .
```

- запуск переключается на:

```
USER appuser:appuser
```

- вместо `CMD` используется `ENTRYPOINT ["./myapp"]`.

Разница: `ENTRYPOINT` фиксирует основную команду запуска, а аргументы из docker run добавляются к ней, а не полностью заменяют её.

4. Что ещё исправляют для production

В промежуточном варианте был плохой момент:


```
ENV API_TOKEN="super-secret-token-123"
```

Так делать нельзя: секреты не должны храниться в Dockerfile и попадать в образ.

```
# Stage 1: сборка зависимостей.
FROM golang:1.25.7-alpine AS builder

WORKDIR /build

COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o myapp .

# Stage 2: финальный образ.
FROM alpine:3.21

ARG BUILD_TIME
ARG VERSION
ARG PORT=8080

LABEL com.express42.compiler="Golang" \
      org.opencontainers.image.version=${VERSION} \
      org.opencontainers.image.authors="andrey@express42.com" \
      org.opencontainers.image.description="Backend API" \
      org.opencontainers.image.created=${BUILD_TIME}

WORKDIR /opt/app
```


```
RUN apk add --no-cache postgresql17-client curl &amp;&amp; \
    addgroup -S appuser --gid 2100 &amp;&amp; \
    adduser -S appuser --uid 2101 -G appuser

COPY --chown=appuser:appuser --from=builder /build/myapp .

USER appuser:appuser

EXPOSE ${PORT}
ENV PORT=${PORT}

HEALTHCHECK --interval=30s --timeout=3s \
            CMD curl --fail http://localhost:${PORT}/health || exit 1

ENTRYPOINT ["./myapp"]
CMD ["--help"]
```

Смысл:

- `ARG` позволяет передавать версию и время сборки из CI/CD;
- OCI labels удобны для registry, аудита и корпоративных стандартов;
- `HEALTHCHECK` позволяет Docker понять, живо приложение или зависло;
- `EXPOSE` документирует порт;
- `CMD` задаёт аргументы по умолчанию для `ENTRYPOINT`.

5. Финальный вывод статьи

Автор делает важную оговорку: не все best practices надо слепо засовывать в Dockerfile. В реальной инфраструктуре часть настроек может жить в `docker-compose.yaml`, Kubernetes `Deployment` или другом оркестраторе.

Поэтому из итогового Dockerfile можно убрать:

- `EXPOSE`;
- `HEALTHCHECK`;
- `CMD`;

если это уже описано на уровне инфраструктуры.

Вот наш итоговый оптимизированный, стабильный, multi-stage Dockerfile:

```
# Stage 1: сборка зависимостей.
FROM golang:1.25.7-alpine AS builder

WORKDIR /build

COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o myapp .

# Stage 2: финальный образ.
FROM alpine:3.21
```


```
ARG BUILD_TIME
ARG VERSION

LABEL com.express42.compiler="Golang" \
      org.opencontainers.image.version=${VERSION} \
      org.opencontainers.image.authors="andrey@express42.com" \
      org.opencontainers.image.description="Backend API" \
      org.opencontainers.image.created=${BUILD_TIME}

WORKDIR /opt/app

RUN apk add --no-cache postgresql17-client && \
    addgroup -S appuser --gid 2100 && \
    adduser -S appuser --uid 2101 -G appuser

COPY --chown=appuser:appuser --from=builder /build/myapp .

USER appuser:appuser

# ENTRYPOINT запускает приложение.
ENTRYPOINT ["./myapp"]
```

Итоговый production-подход

Финальная идея такая:

- фиксированный базовый образ, не `latest`;

- multi-stage build;
- зависимости кешируются отдельно от исходников;
- `go mod download` вместо `go mod tidy`;
- `.dockerignore`;
- финальный образ без исходников и компилятора;
- только нужные runtime-зависимости;
- запуск не от `root`;
- `COPY --chown`;
- OCI labels;
- параметры сборки через `ARG`;
- секреты не хранить в образе;
- healthcheck, port и CMD добавлять только если это не дублируется инфраструктурой.

Дополнительные улучшения из статьи: фиксировать базовый образ не только по версии, но и по digest/hash, подключить `hadolint`, добавить README по сборке/запуску, вынести секреты в `secret storage`, добавить vulnerability scanning.