Оглавление

Оглавление

[Github Flow 1](#_6rh462j86pni)

[Введение в Docker 2](#_yixmx3wley30)

[Введение в docker compose 5](#_ykou0up1pc6d)

[Dockerfile 5](#_qqh6d27v2aaf)

[Multi-stage build 8](#_4gz9v9rk1lsk)

[docker compose 9](#_agjulhlnvug8)

# 

# Github Flow

Рассматривается работа от лица разработчика и управляющего, при этом границы весьма условны, поскольку создание репозитория в проектах обычно создает на рядовой программист, а главный программист, ревью кода могут проводить все, а разрешение конфликтов чаще выполняет разработчик.

Работа от лица разработчика:

Создание репозитория — это редкая процедура, которая обычно выполняется один раз на проект. При создании важно сразу настроить базовые файлы: .gitignore для исключения ненужных файлов, README.md с описанием проекта и инструкциями по запуску, а также лицензию, если она требуется. Полезно добавить шаблоны для issue и PR, чтобы стандартизировать описания задач и изменений.

Issue — это задача, баг-репорт или предложение по улучшению. В ней описывается, что нужно сделать: проблема, шаги для воспроизведения (если это баг), или ожидаемый результат. Разработчик выбирает issue, закрепляет за собой, обновляет статус по мере прогресса и комментирует, если возникают вопросы или блокеры.

Pull Request (PR) создается после завершения работы над issue. В PR должно быть понятное название, описание изменений, ссылка на связанный issue (например, "Closes #123"), а также скриншоты или логи, если изменения визуальные или сложные. Разработчик указывает ревьюеров — обычно одного-двух человек, которые лучше всего разбираются в изменяемой части кода.

Работа от лица руководителя

Ревью кода — это проверка соответствия изменений требованиям issue, качества кода, отсутствия багов и уязвимостей, а также наличия тестов. Обратная связь должна быть конкретной: с указанием строк кода и конструктивными предложениями. Важно хвалить за хорошие решения, чтобы мотивировать команду.

Доработка PR требуется, если ревьюер нашел проблемы или предложил улучшения. Разработчик вносит правки, обновляет PR, а ревьюер проверяет изменения повторно. Если правки значительные, лучше обсудить их отдельно — в чате или на встрече.

Конфликты возникают, когда изменения в PR конфликтуют с текущей версией основной ветки. Разработчик должен обновить свою ветку, разрешить конфликты локально и загрузить изменения. Для простых конфликтов можно использовать инструменты GitHub, для сложных — обсудить с командой.

Принятие PR происходит, когда все комментарии решены, конфликтов нет, а тесты (если они есть) проходят. Руководитель нажимает "Merge", выбирает стратегию слияния (например, "Squash and merge") и при необходимости удаляет ветку после слияния.

# Введение в Docker

**Docker** — это платформа для разработки, доставки и запуска приложений в контейнерах. Контейнер — это легковесная, изолированная среда, которая упаковывает приложение со всеми его зависимостями (библиотеками, системными инструментами и т.д.). Это позволяет запускать приложение на любой машине с Docker, не беспокоясь о конфликтах версий или окружения.

**Аналогия**: Представьте контейнер как коробку с готовым набором для сборки мебели. Внутри — всё необходимое, и вам не нужно искать недостающие детали или инструменты.

**Зачем нужен Docker?**

* Единообразие окружений. Разработчики, тестировщики и продакшн работают с одинаковым окружением. Нет фразы "У меня на машине работает!".
* Изоляция. Каждое приложение работает в своем контейнере, не мешая другим.
* Портативность. Контейнеры легко переносятся между машинами, облаками и серверами.
* Экономия ресурсов. Контейнеры легче виртуальных машин, так как используют ядро хостовой ОС.
* Быстрое развертывание. Запуск контейнера занимает секунды, а не минуты.

**Основные понятия:**

* Образ (Image): Шаблон для создания контейнера. Содержит приложение и его зависимости.
* Контейнер (Container): Запущенный экземпляр образа.
* Dockerfile: Файл с инструкциями для сборки образа (например, какие пакеты установить).
* Репозиторий (Repository): Место хранения образов (например, Docker Hub).
* Том (Volume): Механизм для сохранения данных вне контейнера (чтобы данные не пропадали при удалении контейнера).
* Сеть (Network): Позволяет контейнерам общаться друг с другом и с внешним миром

**Скачивание образов с помощью docker pull**

Команда docker pull позволяет скачать готовый образ из репозитория, например Docker Hub. Обычно указывают имя образа и его тег, например nginx:latest. Важно помнить, что тег latest не всегда означает самую стабильную или актуальную версию, поэтому лучше явно указывать нужную версию, например nginx:1.23.3.

**Сохранение и загрузка образов с помощью docker save и docker load**

Команда docker save сохраняет образ в виде tar-архива, что удобно для переноса образов между машинами без доступа к интернету или для резервного копирования. Например, можно сохранить образ nginx в файл my\_nginx.tar. Команда docker load загружает образ обратно из архива. Это полезно, если нужно развернуть одно и то же окружение на нескольких машинах без повторной загрузки из интернета.

**Просмотр запущенных контейнеров с помощью docker ps**

Команда docker ps отображает список запущенных контейнеров. Если добавить флаг -a, то будут показаны все контейнеры, включая остановленные. В выводе команды важно обращать внимание на идентификатор контейнера (CONTAINER ID), его статус (STATUS) и используемые порты (PORTS). Это помогает быстро ориентироваться в текущем состоянии контейнеров.

**Запуск контейнеров с помощью docker run**

Команда docker run создаёт и запускает новый контейнер из указанного образа. Часто используемые флаги включают:

* -d для запуска контейнера в фоновом режиме (detached mode),
* -p для проброса портов, например -p 8080:80 пробрасывает порт 80 контейнера на порт 8080 хостовой машины,
* --name для задания имени контейнера, что упрощает управление им.

Например, запуск контейнера из образа nginx с пробросом порта и именем my\_nginx позволяет быстро развернуть веб-сервер. После запуска можно проверить его работу, открыв в браузере http://localhost:8080.

**Выполнение команд в запущенном контейнере с помощью docker exec**

Команда docker exec позволяет выполнить команду внутри уже запущенного контейнера. Часто используется с флагами -it, которые обеспечивают интерактивный режим с терминалом. Например, можно войти в контейнер my\_nginx и запустить оболочку /bin/bash для отладки или изменения конфигураций. Это удобно, когда нужно оперативно проверить состояние контейнера или внести изменения без его остановки.

**Просмотр логов контейнера с помощью docker logs**

Команда docker logs выводит логи указанного контейнера. Флаг -f позволяет наблюдать за логами в реальном времени, что аналогично команде tail -f в Linux. Это особенно полезно для мониторинга работы приложений в контейнерах, например, когда нужно отследить ошибки или проверить корректность работы сервиса.

**Работа с томами через docker volume**

Тома в Docker используются для сохранения данных вне контейнера, что позволяет избежать потери информации при его удалении или перезапуске. Команда docker volume create создаёт новый том, а флаг -v в команде docker run подключает этот том к контейнеру. Например, том my\_volume можно подключить к пути /app/data внутри контейнера. Это актуально для баз данных, пользовательских файлов или любых данных, которые должны сохраняться между перезапусками.

**Управление контейнерами с помощью docker stop, docker start и docker rm**

Команда docker stop останавливает работающий контейнер, а docker start запускает остановленный контейнер заново. Команда docker rm удаляет контейнер, но перед этим его необходимо остановить. Если контейнер нужно удалить силой, используется флаг -f. Эти команды позволяют гибко управлять жизненным циклом контейнеров, освобождая ресурсы и поддерживая порядок в системе.

# Введение в docker compose

## Dockerfile

**Dockerfile** — это текстовый файл с набором инструкций, которые описывают, как собрать Docker-образ. Он позволяет автоматизировать процесс создания образа, гарантируя, что все зависимости, настройки и файлы будут упакованы корректно. Без Dockerfile сборка образа превратилась бы в рутинную и подверженную ошибкам процедуру, особенно в командах, где несколько разработчиков работают над одним проектом.

Основное преимущество Dockerfile заключается в воспроизводимости: любой разработчик может собрать идентичный образ из одного и того же файла. Кроме того, Dockerfile можно хранить в системе контроля версий, что позволяет отслеживать изменения и при необходимости откатываться к предыдущим версиям. Это делает процесс разработки и развёртывания более прозрачным и управляемым.

Dockerfile состоит из последовательности инструкций, каждая из которых определяет шаг сборки образа. Инструкции записываются в верхнем регистре и выполняются сверху вниз. Каждая инструкция создаёт новый слой в образе, что важно учитывать для оптимизации размера и скорости сборки. Комментарии в Dockerfile начинаются с символа # и помогают объяснить назначение тех или иных инструкций.

Инструкция **FROM** определяет базовый образ, на основе которого будет строиться новый образ. Это может быть любой официальный или кастомный образ из Docker Hub, например, ubuntu, python или nginx. Важно указывать конкретные версии базового образа, чтобы избежать неожиданных изменений при обновлении. Например, лучше использовать python:3.9-slim, чем просто python:latest, так как тег latest может со временем измениться.

Инструкция **RUN** позволяет выполнять команды внутри контейнера во время сборки образа. Она часто используется для установки пакетов, настройки окружения или выполнения других подготовленных действий. Каждая инструкция RUN создаёт новый слой, поэтому для оптимизации сборки рекомендуется объединять несколько команд в одну инструкцию с помощью логического оператора &&. Например, обновление пакетов и установка новых можно объединить в одну строку, чтобы уменьшить количество слоев.

Инструкции **ADD** и **COPY** предназначены для копирования файлов из хостовой машины в образ. Инструкция COPY просто копирует файлы и директории, тогда как ADD обладает дополнительными возможностями, такими как автоматическая распаковка архивов и скачивание файлов по URL. Однако из-за этих дополнительных функций ADD может вести себя менее предсказуемо, поэтому в большинстве случаев предпочтительнее использовать COPY.

Инструкции **CMD** и **ENTRYPOINT** определяют, какая команда будет выполнена при запуске контейнера. Разница между ними заключается в том, как они взаимодействуют с аргументами командной строки. Инструкция CMD задает команду по умолчанию, которую можно переопределить при запуске контейнера. В то время как ENTRYPOINT определяет команду, которая всегда будет выполнена, а аргументы из CMD или командной строки будут добавлены к ней. Например, если в ENTRYPOINT указано ["python"], то аргументы из CMD или командной строки будут передаваться как аргументы для команды python.

Инструкция **ENV** устанавливает переменные окружения, которые будут доступны внутри контейнера. Это удобно для настройки приложения, например, для указания портов, путей к файлам или ключей API. Переменные окружения позволяют гибко настраивать поведение приложения без изменения кода.

Инструкция **ARG** позволяет передавать переменные во время сборки образа. Эти переменные доступны только на этапе сборки и не сохраняются в конечном образе. Они полезны для передачи параметров, которые могут изменяться между сборками, например, версии приложения или пути к файлам.

Инструкция **WORKDIR** задаёт рабочую директорию внутри контейнера, в которой будут выполняться последующие инструкции, такие как RUN, CMD или COPY. Это упрощает работу с путями и делает Dockerfile более читаемым, так как не нужно указывать полные пути к файлам.

Инструкция **USER** определяет, от имени какого пользователя будут выполняться команды в контейнере. По умолчанию используется пользователь root, но для повышения безопасности рекомендуется создавать и использовать отдельного пользователя с ограниченными правами.

Инструкция **VOLUME** объявляет точку монтирования для внешних томов. Это позволяет сохранять данные вне контейнера, что особенно важно для баз данных или конфигурационных файлов, которые должны сохраняться между перезапусками контейнера.

**Контекст сборки** — это набор файлов и папок, которые передаются Docker-демону во время выполнения команды docker build. Контекст определяется путем, указанным в команде сборки. Например, если выполнить команду docker build -t my\_image ., то контекстом станет текущая директория. Важно помнить, что все файлы из контекста отправляются демону, поэтому не стоит включать в него ненужные файлы, такие как .git или временные файлы, чтобы не замедлять процесс сборки.

**Слои** — каждая инструкция в Dockerfile создаёт новый слой в образе. Docker кэширует эти слои, и если инструкция не изменилась, то используется кэшированная версия. Это значительно ускоряет повторную сборку. Однако важно учитывать порядок инструкций: если часто изменяемые файлы, такие как исходный код приложения, копируются в начале Dockerfile, это приведёт к сбросу кэша для всех последующих слоев. Поэтому такие инструкции лучше размещать ближе к концу файла.

Docker **кэширует** слои, что позволяет ускорить процесс сборки образа. Если инструкция или файлы *не изменились*, Docker *использует кэшированный* слой. Однако иногда требуется принудительно обновить зависимости или файлы, игнорируя кэш. В таких случаях можно использовать флаг **--no-cache** при сборке образа. Это заставит Docker выполнить все инструкции заново, не полагаясь на кэш.

## Multi-stage build

**Multi-stage build** — это техника сборки Docker-образов, позволяющая использовать несколько промежуточных контейнеров (ступеней) в одном Dockerfile. Основная идея заключается в том, чтобы разделить процесс сборки на этапы, где на каждом этапе выполняются определенные задачи, а в финальный образ копируются только необходимые файлы. Это позволяет значительно уменьшить размер итогового образа, исключив из него ненужные зависимости, временные файлы и инструменты, которые требовались только на этапе сборки.

Например, при сборке приложения на Go или C++ часто требуются компиляторы, библиотеки и другие инструменты разработки, которые не нужны для запуска уже скомпилированного приложения. С помощью Multi-stage build можно сначала собрать приложение в одном контейнере, а затем скопировать только бинарный файл в финальный, минималистичный образ. Это делает образ более легким и безопасным, так как в нем отсутствуют лишние компоненты, которые могли бы содержать уязвимости.

**Сценарии использования Multi-stage build**

Multi-stage build особенно полезен в случаях, когда процесс сборки приложения требует наличия дополнительных инструментов или зависимостей, которые не нужны для его выполнения. Рассмотрим несколько типичных сценариев.

Для компилируемых языков, таких как C++, Multi-stage build позволяет отделить этап компиляции от этапа запуска. На первом этапе устанавливаются компиляторы и зависимости, собирается бинарный файл, а на втором — бинарник копируется в минималистичный образ, например, на основе alpine, который содержит только необходимые для запуска библиотеки.

В веб-разработке, особенно при использовании фронтенд-фреймворков вроде React или Angular, Multi-stage build помогает отделить этап сборки статических файлов от этапа их развертывания. На первом этапе устанавливаются Node.js и зависимости, собирается статическая версия приложения, а на втором — готовые файлы копируются в легковесный веб-сервер, например, nginx.

Таким образом, Multi-stage build делает образы более компактными, безопасными и быстрыми в развертывании.

## **docker compose**

**Docker Compose** — это инструмент, предназначенный для упрощения управления много контейнерными приложениями. Он позволяет описывать и запускать несколько контейнеров как единое целое, используя конфигурационный файл docker-compose.yml. Это особенно полезно, когда приложение состоит из нескольких сервисов, например, веб-сервера, базы данных, кэша и фоновых задач, которые должны взаимодействовать друг с другом.

Основное преимущество Docker Compose заключается в возможности определить все сервисы, их зависимости, сети и тома в одном файле, а затем запустить их одной командой. Это упрощает развертывание, тестирование и масштабирование приложений, особенно на этапах разработки и тестирования. Docker Compose позволяет избежать ручного запуска каждого контейнера по отдельности, настраивая сети и тома вручную, что экономит время и уменьшает вероятность ошибок.

**Базовые директивы Docker Compose**

Файл **docker-compose.yml** содержит описание сервисов, сетей и томов, необходимых для запуска приложения. Каждый сервис определяется как отдельный блок с указанием образа, портов, переменных окружения, зависимостей и других параметров.

Директива **services** определяет набор сервисов, из которых состоит приложение. Каждый сервис описывается под своим именем и может содержать параметры, такие как **image** (образ, из которого создается контейнер), **build** (путь к Dockerfile, если образ нужно собрать), **ports** (проброс портов), **volumes** (монтирование томов), **environment** (переменные окружения), **depends\_on** (зависимости от других сервисов) и другие.

Директива **networks** позволяет определить сети, которые будут использоваться сервисами для взаимодействия друг с другом. По умолчанию Docker Compose создаёт одну сеть для всех сервисов, но можно явно задать несколько сетей с разными настройками.

Директива **volumes** описывает тома, которые могут быть использованы сервисами для хранения данных. Тома позволяют сохранять данные между перезапусками контейнеров, что важно для баз данных или пользовательских файлов.