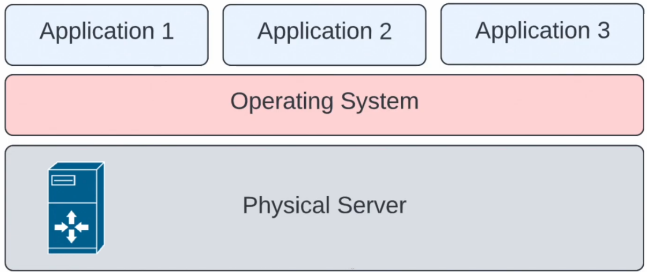
**История появления:**

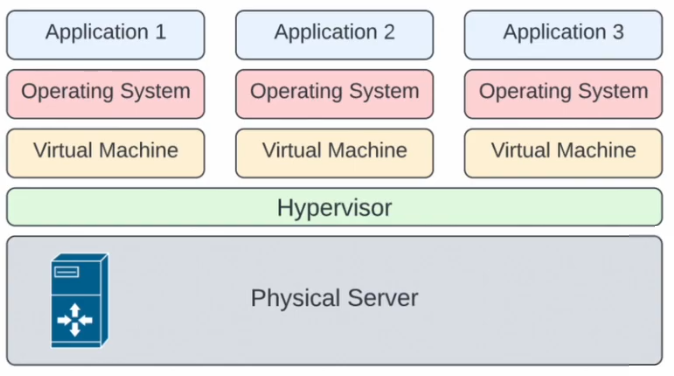
Классическая схема использования одного сервера и одной операционной системы с несколькими запущенными приложениями:

****Проблемы:

1. Несколько приложений хотят занять одинаковый порт
2. Использование одного файла несколькими приложениями приводят к блокировке потока другого
3. Приложения хотят использовать разные версии одной глобальной библиотеки
4. Конкуренция за ресурсы системы (нагрузка на одно из приложений влияет на производительность другого)

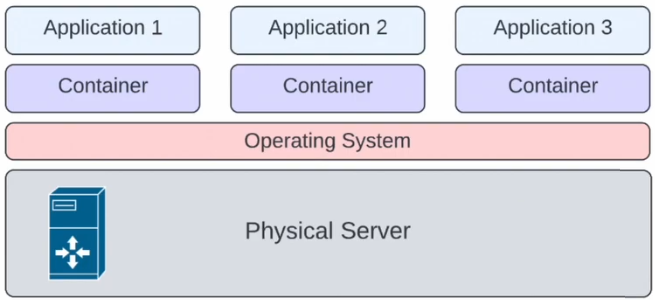
Для решения проблемы нужно изолировать процессы, что может быть достигнуто:

1. Установкой каждого приложения на отдельный сервер (дорого и покупались с запасом, так как сложно оценить нагрузку на приложение)
2. **Виртуализация –** изоляция вычислительных процессов, выполняемых на одном физическом устройстве, и разделение ресурсов между ними.

Типы виртуализации:

**Аппаратная** – на сервер устанавливается host ОС (легковесная, предназначена в основном для запуска гостевых ОС), которая умеет запускать другие операционные системы (виртуальные машины) и выделяет для каждой собственные ресурсы.

Минусы: для каждого процесса своя ОС + доп потребление ресурсов для host ОС

****

**Виртуализация на уровне ОС** (контейнеризация) **–** изолирует процессы и ресурсы для них в рамках одной операционной системы (виртуальные машины заменены на контейнеры)

Не эмулируют операционную систему, поэтому контейнеры поднимаются быстро

Виртуализация на уровне ОС использует механизм изоляции Linux namespaces, который позволяет каждому процессу видеть только часть принадлежащих ему ресурсов системы:

1. pid – процессы
2. net – сеть
3. mt – файловая система
4. uts – хост
5. ipс – очереди сообщений
6. user – пользователь
7. cgroup – контроль ресурсов

Так как для реализации ОС виртуализации необходимо ядро Linux, то при запуске Docker на Linux OC используется чистая виртуализация на уровне ОС, если же запуск происходит на отличной ОС, то ядро Linux по-прежнему необходимо, что реализуется посредством развертывания Linux виртуальной машины, что является гибридным подходом (аппаратный + ОС уровень).

**Свойства контейнеризации**

1. Изоляция
   1. Контейнер работает в своей изолированной среде (процессы, сеть, файловая система).
   2. Контейнеры не конкурируют друг с другом за ресурсы и не используют общие ресурсы.
2. Самодостаточность
   1. В контейнер упакованы все зависимости, нужные для работы приложения.
3. Переносимость – изолированность от среды и самодостаточность делает возможным запуск одного и того же контейнера на разных ОС с одинаковым поведением
4. Масштабируемость
   1. Контейнеры легко реплицируются (горизонтальное масштабирование) – хорошо ложатся в системы оркестрации (Kubernetes, Docker Swarm).
5. Лёгкость и эффективное использование ресурсов
   1. Контейнеры "разделяют" ядро ОС → потребляют меньше CPU, RAM и диска, чем виртуальные машины.
   2. Быстрый запуск, т.к. не нужно поднимать новую ОС для запуска контейнера

**Docker** – платформа для запуска, создания и управления контейнерами, образами. Позволяет собирать приложения со всеми зависимостями в переносимый докер образ и запускать его в изолированной среде.

Архитектура:

1. Client – api для взаимодействия с Daemon, передает запросы на него, состоит из:
   1. CLI (консоль) – клиент для одиночных команд
   2. Docker Compose – клиент для описания и запуска множества контейнеров через единую схему
2. Daemon (служба) – докер сервер, отвечает за все задачи по созданию контейнеров и все задачи по обработке запросов от клиентов (загружает докер образы, анализирует метаданные, собирает из всех слоев контейнер)
3. Host – машина, на которой запущен Docker Daemon

Разделение на клиентскую часть и исполняющую (daemon) позволяет удалённо управлять любым Docker Engine, независимо от того, где он запущен.

На Windows Docker Client работает на «родной» ОС, а Docker Daemon для Linux-контейнеров запускается внутри легковесной виртуальной машины (WSL2 или Hyper-V), что обеспечивает работу контейнеров Linux на не-Linux системе.

**Image** – шаблон для создания контейнера.

Состоит из:

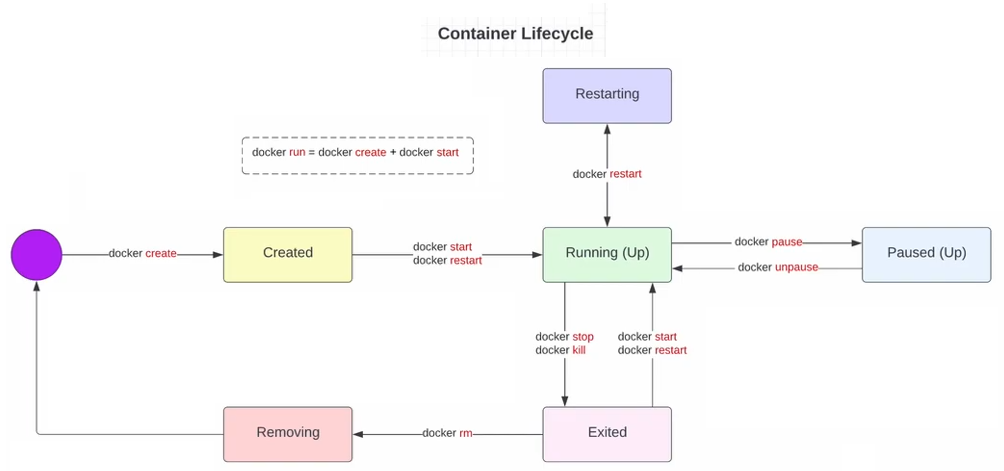
1. Метаданные – служебная информация, которая описывает его зависимости, параметры, как его запускать (описывается командами типа cmd, env, expose, workdir, volume и др.)
2. Слои – снимок файловой системы после выполнения команды (по сути каждый слой изменяет файловую систему, добавляя файлы и директории)
   1. Слои создаются командами типа from, run, copy, add и др.
   2. каждый слой иммутабельный (read-pnly)
   3. кэшируемый для возможности его переиспользования, что определяется хэшсуммой
   4. чтобы уменьшить финальный размер образа нужно использовать Multy-stage build: все промежуточные этапы в build этап, final этап – только то, что нужно для запуска.

**Способы получения image:**

1. dockerfile – содержит инструкции по созданию image (см. ниже)
2. Registry (реестр) – хранилище репозиториев, может быть как локальным, так и удаленным – по умолчанию dockerhub, могут быть облачные или self-hosted
   1. Repository (репозиторий) – набор образов с одинаковым именем, но разными тэгами (версиями)

**Контейнер** – изолированная среда, в которой есть все что нужно для запуска приложения, все приложения запускаются внутри него.

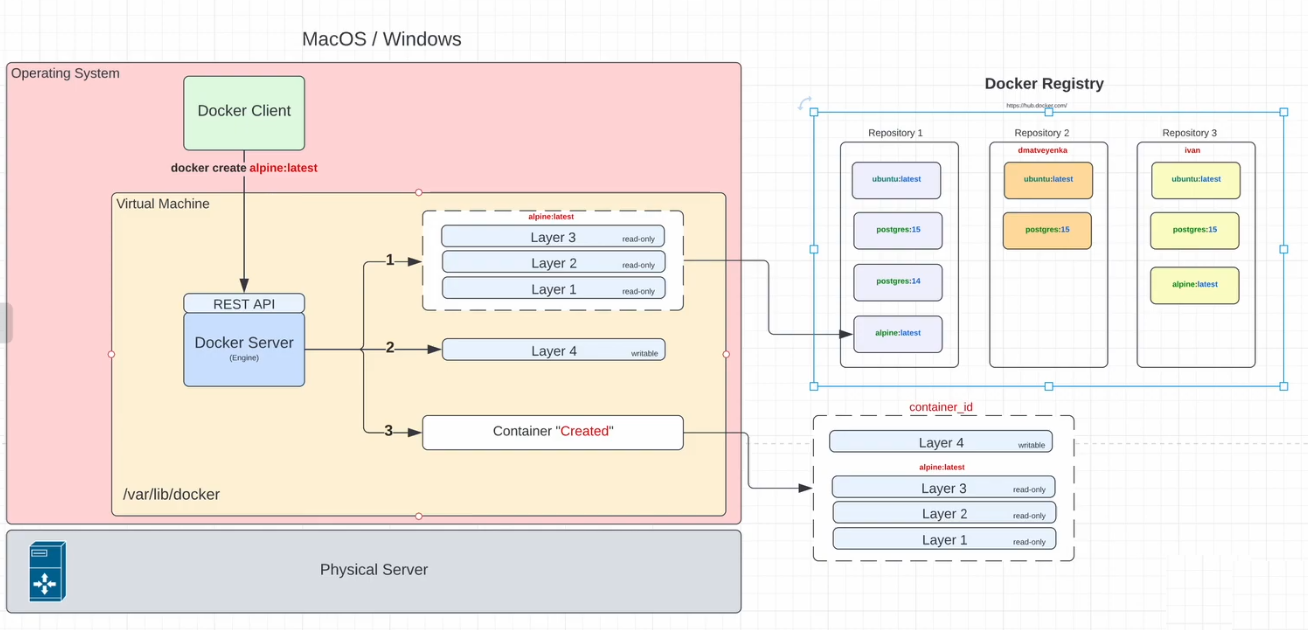
Построение контейнера происходит на базе read-only слоев контейнера, также добавляется дополнительный writable слой (контейнерный), там хранятся все файлы жизнедеятельности контейнера (измененные файлы из образа (копируются в слой из image), новые файлы, кременные файлы и кэш и тд ).

**ЖЦ контейнера**

**Create** (docker create) **–** создание готового к запуску контейнера без его фактического запуска.

1. Не запущен
2. Ресурсы CPU, память, порты и тд еще не выделены

Механика:

1. Отправка команды с CLI на daemon
2. Поиск образа – проверяется локальный кэш, если не найден смотрится указанный registry (если не указан, то по умолчанию используется docker hub) и образ подтягивается оттуда
3. Инициализируется файловая система – послойно накладываются read-only слои образа и добавляется writable layer
4. Подтягиваются метаданные для контейнера – порт, env, параметры ресурсов, volume
5. Контейнер создан и готов к запуску

**Running** (run, start, restart, unpause) – запуск созданного контейнера.

Механика:

1. Поиск контейнера и проверка соответствия его статуса и команды вызова
2. Выделение контейнеру ресурсов, указанных при создании (cpu, сеть, volume)
3. Настройка окружения – подстановка env и volume
4. Инициализация и выполнение main процесса (cmd, entrypoint)

\*Контейнер не может существовать без активного процесса – тушится процесс – тушится и контейнер

**Exited** (stop, kill) – остановка процессов контейнера

Механика:

1. Проверка соответствия его статуса и команды вызова
2. Завершение исполнения процесса
3. Освобождение занятых ресурсов

Stop – посылается сигнал на graceful shutdown – процесс завершает работу самостоятельно (сопровождается exited(0))  
Kill – выставляется timeout (время на самостоятельное завершение процесса), если его не происходит, то происходит принудительное завершение процесса (сопровождается exited(127/137))

\*Состояние writable layer сохраняется

**Removing** (rm) – удаление контейнера

1. Проверка соответствия его статуса и команды вызова (если -f, то можно удалять запущенный контейнер)
2. Уничтожение writable layer и его метаданных
3. Контейнер удален

**Paused** (pause) – заморозка процессов внутри контейнера

Механика:

1. Проверка соответствия его статуса и команды вызова
2. Процесс ставится на паузу
3. Освобождается только CPU (память и RAM остаются занятыми)

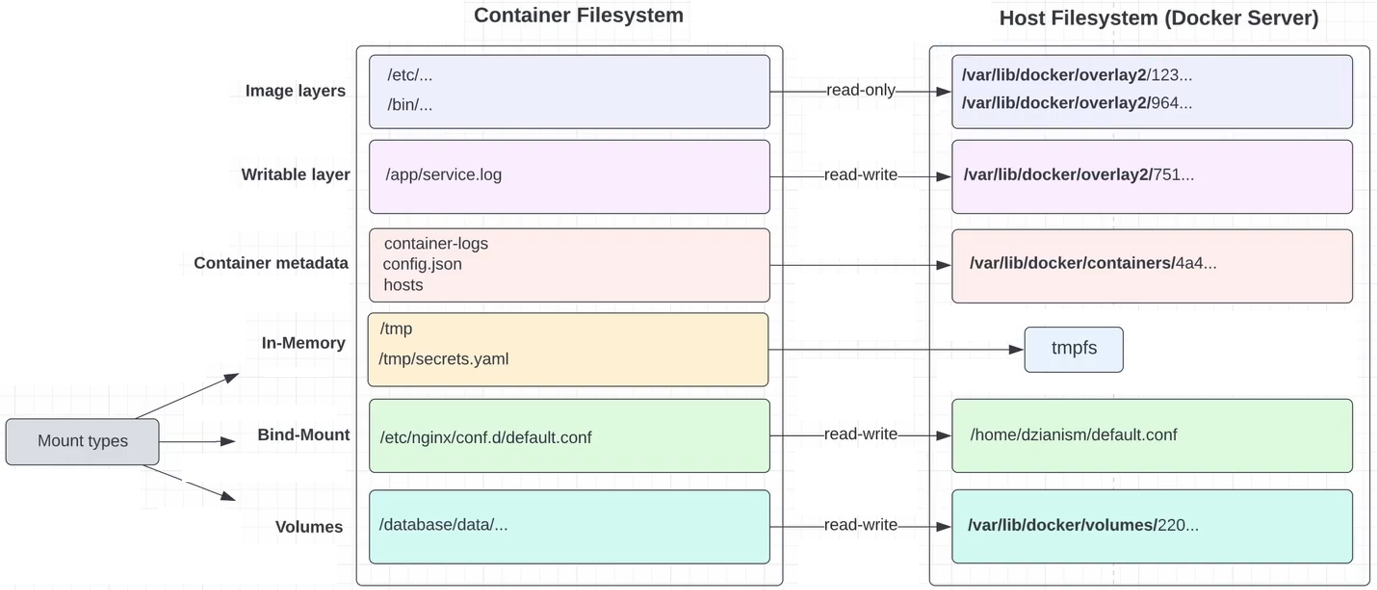
**Restarting** (restart) – перезапуск контейнера

1. Проверка соответствия его статуса и команды вызова (может быть exited, created, running)
2. Выделение ресурсов – повторное резервирование ресурсов для контейнера
3. Запуск процесса – перезапуск процесса

**\***рестарт для running запускается только если прописана политика рестарта (--restart)

**Storage and mount types**

**Storage –** различные механизмы хранения данных контейнеров и образов, которые предоставляют несколько вариантов связывания файловой системы Host и контейнера (6 типов).



**Volumes –** постоянноеизолированное от контейнеров хранилище данных в Docker (не на хост машине).

1. Располагается внутри виртуальной машины, что делает его независимым от окружения и предотвращает возникновение проблем с совместимостью
2. Управляется Docker – автоматически определяется место его хранения и управление им с помощью Docker команд
3. Изоляция данных от самого контейнера – данные не привязаны к конкретному контейнеру, удаление контейнера не приводит к удалению Volume.
4. Может использоваться несколькими контейнерами.

\*Можно отдельно создать Volume: docker volume create <Имя>, если не создавать, то Volume создастся автоматически

**Bind Mounts –** позволяет хранить информацию вне Docker контейнера. Создается проекция директории/файла вне контейнера (на Host). Проекция и исходный файл синхронизированы.

\*Опция **readonly** не позволяет изменять директорию из контейнера

Минусы:

1. Потеря изоляции – разделяемый доступ к информации
2. оверхед при использовании на Windows/mac – необходим маппинг между системами
3. невозможность автоматического переноса данных – нужно перенастраивать путь, также отдельно переносить данные с хоста.
4. Менее производителен, чем volumes.

**Tmpfs –** Temporary file system – временная файловая система, которая работает в оперативной памяти, а не на жестком диске.

\*Можно указывать размер оперативной памяти, и mode – для указания прав доступа внутри контейнера

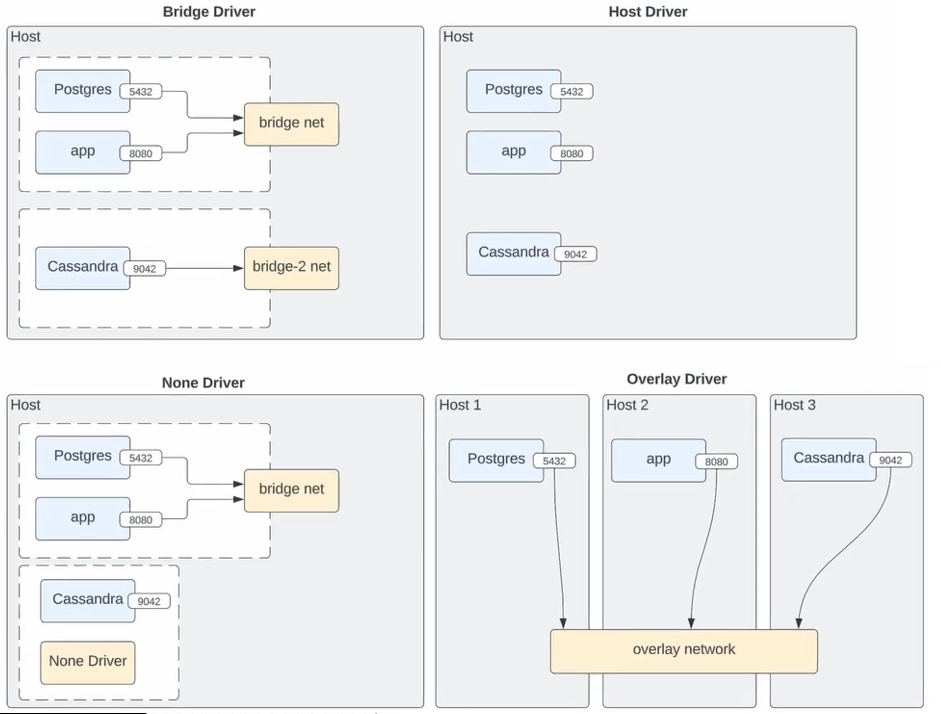
Плюсы/Минусы:

1. Быстрее storage в долговременной памяти
2. Невозможно шарить между контейнерами
3. не хранит состояние после отключения
4. поддерживается только на Linux, на остальных создается имитация и все равно хранится на жестком диске

**Networking**

- механизм, который отвечает за сетевое взаимодействие контейнеров. Позволяет обеспечивать сетевое взаимодействие между контейнерами или внешним миром, сохраняя изоляцию

За способ сетевого взаимодействия отвечают 4 вида драйверов:



**Bridge** – драйвер по умолчанию, создается подсеть, объединяющая группы контейнеров, которые имеют свободный доступ по сети между собой, а также способны обращаться к внешнему миру, но для обращения к ним необходим port forwarding.

*Port forwarding* – механизм, который позволяет направлять сетевой трафик с порта host машины на порт контейнера или наоборот (маппинг порта).

*Виды bridge драйверов:*

1. *Default-bridge* - создается автоматически самим Docker
   1. все контейнеры, у которых не указана сеть подключаются к ней
   2. Имеет ограниченные возможности конфигурации – нет встроенного DNS для обращения по имени контейнера (только по ip), нельзя управлять подсетями и правилами безопасности
2. *User-defined bridge network –* создается пользователем вручную
   1. Имеет DNS резолвер – можно использовать имя контейнера как его ip
   2. Есть возможность настройки подсетей и правил безопасности

**Host** – контейнер не получает собственной сети, а использует сеть host машины напрямую (порты контейнера полностью совпадают с портами host).

\* Не поддерживается напрямую в win/mac так как все равно идет проброс порта из VM на host.

1. Приводит к отсутствию изоляции по сети
2. Нет необходимости ручной конфиграции каждого порта
3. Снижение оверхэда на маппинг портов

**None** – полная сетевая изоляция: контейнер не получает сетевой интерфейс, не имеет доступа к внешнему миру и недоступен извне.

**\*** В основном используется для тестирования сервисов в изолированной среде

**Overlay** – создание виртуальной распределенной сети для взаимодействия контейнеров, расположенных на разных хотсах.

**Команды**

1. --help раскрывает доступные команды для этого уровня:   
   docker –help – список основных команд  
   docker network –help – [network] – основная команда, [help] - даст список подкоманд для network

Основные:

1. -- version - версия клиента и daemon
2. ps – список запущенных (-a) - остановленных контейнеров
3. Images – список локальных образов

Команды для переключения состояний контейнера:

1. Create
2. Start
3. Stop
4. kill
5. Restart
6. Pause
7. Unpause
8. Rm
9. Run – create + start

\*Описание команд выше

Флаги:

1. -I – запуск интерактивного ввода (контейнер слушает ввод)
2. -t – запуск виртуального терминала

\*запуск контейнера без этого флага – контейнер выполнит команду и завершит исполнение и далее Docker его закроет, запуск с флагами позволяет зайти внутрь контейнера

1. -d - запуск контейнера в фоновом режиме (то есть не заходим внутрь контейнера, но контейнер продолжит работать)
2. --name – задать собственное имя контейнеру
3. -p <внешний\_порт>:<внутренний\_порт> - публикация порта (проброс порта)

Команда docker run автоматически подключается к standard output – то есть залетает внутрь контейнера в этом терминале и если выйти, то процесс закроется, чтобы избежать -d

\*Exit – выйти из исполняемого процесса

Остальные:

1. Pull <registry>/<repository>/<name>:<tag> - подтянуть образ (обязательный только name, остальное подставляется по умолчанию)
2. Container prune – дропнуть все остановленные контейнеры
3. Inspect <id\_или\_имя контейнера> | grep или select-string <нужные\_метаданные> - вывод всех метаданных контейнера
4. Exec <id\_или\_имя\_контейнера> <название\_процесса> – создание нового процесса в запущенном контейнере, используется как инструмент для внедрения в уже запущенный контейнер. Его главная цель — выполнять команды внутри контейнера, не прерывая его работы и не пересоздавая процесс.
5. Cp<Контейнер>:<Путь\_на\_хосте> <Путь\_в\_контейнере> – копирование файлов между контейнером и хостом – работает и в обратную сторону
6. Diff – отображает изменения произошедшие в контейнере по сравнению с исходным образов (показывает writable layer)
7. Logs <имя\_контейнера> -подключение к потоку вывода контейнера (просмотр логов)
8. Stats – мониторинг ресурсов, потребляемых контейнерами
9. Attach <имя\_контейнера> – подключиться к работающему контейнеру

Network and Storage:

1. docker network create <имя> - создание кастомной bridge подсети
2. –network <имя\_драйвера> - подключение кастомного bridge к контейнеру
3. -p <публичный порт>:<порт\_контейнера> - указание port forwarding
4. docker run –mount type=volume, source=<имя\_Volume>, target=<директория в контейнере, которая синхронизируется с Volume >

**Image creation**

**Dockerfile –** файл с набором инструкций для создания image.

Создание image инициируется командой Docker build -f <Dockerfile\_name> <build context>

* -f указание пути к dockerfile
* -t задает имя image\_name:tag

(равноценно команде **docker tag** <iamge\_id> < image\_name:tag >)

* можно писать без указания Dockerfile, тогда возьмется файл с зарезервированным названием Dockerfile (без формата)

*Build context* **–** директория/удаленный ресурс содержимое которой Docker передает в Daemon при выполнении команды build. Из этой директории берутся: Dockerfile, файлы, которые указаны в командах COPY и ADD.

Механика:

1. вызывается команда build и передается какой-либо контекст
2. Docker читает все файлы внутри указанной директории
3. Создается временный архив и передается в Daemon
4. Daemon ищет Dockerfile и выполняет его инструкции, используя файлы из архива

Контекст может быть задан как:

1. Локальный – можно задать с помощью указания относительного, полного пути, текущей директории (с помощью .) или, если dockerfile лежит не в контексте или имеет имя отличное от стандартного, с помощью -f
2. Удаленный контекст – с помощью указания URL
3. Не использовать контекст, указывается с помощью “–“, в таком случае Dockerfile передается через standard input (передается в консоль)

*Структура:*

1. Команды для Dockerfile пишутся по стандартной схеме: INSTRUCTION arguments
2. Всегда начинается с FROM где указывается базовый образ на Linux систему, может начинаться с ARG, если для базового образа заданы аргументы

[Инструкции](https://docs.docker.com/reference/dockerfile/):

1. FROM – задает образы, на которых будет базироваться этот образ
2. ARG – локальная переменная, которую надо передать при build с помощью --build-arg
3. WORKDIR – задает рабочую директорию внутри контейнера, все последующие команды типа RUN, COPY и другие команды, работающие с файловой системой, будут выполнены в этой директории
4. RUN – выполнение команд внутри образа
5. COPY – копирование файлов и директорий из build context внутрь образа
6. ADD – возможности как и у Copy + возможность добавлять файлы по URL и распаковывает файлы автоматически
   1. \*Copy для локальных, для удаленных предпочтительнее использовать RUN так как она более гибкая, можно подчистить за собой скачанные файлы
7. ENTRYPOINT – используется для жёсткого указания процесса (executable) и ее параметров, которая будет всегда запускаться при старте контейнера.
   1. Жестко фиксирует переданные параметры – не рекомендуется, т.к. переопределение только с помощью –entrypoint
8. CMD – для указания главной команды (executable) и параметров для нее, который будут использованы «по умолчанию».
   1. Параметры по умолчанию дают возможность переопределить их при необходимости
   2. Рекомендуется комбинировать команды ENTRYPOINT и CMD, в первом задается процесс, который всегда запускается, во втором задаются параметры процесса по умолчанию, которые можно переопределить.
9. EXPOSE – для установки порта, который будет слушать контейнер.

**Оптимизация сборки Image**

*Кэширование*

Для ускорения сборки образов используется механизм кэширования – Dockerfile обрабатывается пошагово и производится проверка существования каждого слоя в кэш, если выполняются условия, то слой может быть переиспользован:

1. Слой существует в кэше
2. Слой не был изменен (не изменилась checksum)
3. Не была изменена инструкция для создания этого слоя

* Изменение в слое сбрасывает кэш для всех следующих слоев (при изменении вышестоящего слоя все нижестоящие кэши не смогут быть загружены из кэша, т.к. неизвестно, как эти изменения могли повлиять на совместимость, следовательно, необходимо соблюдать порядок вызова инструкций).
* Можно отключить использование закэшированных значений с помощью флага –no-cache

*Multi-stage build*

Подход, при котором в одном Dockerfile используется несколько этапов сборки (build stages), чтобы отделить процесс сборки приложения от финального образа.

1. Разделить процесс сборки и финальный образ приложения
2. Исключить из финального образа все временные файлы, исходники, инструменты сборки и тд, оставив только необходимое для runtime
3. Уменьшить размер образа
4. То есть все подготовительные этапы, такие как: скачивание зависимостей, копирование исходного кода, компиляция и другие действия, связанные со сборкой — должны выполняться отдельно, на ранних стадиях.
5. Всё, что не участвует непосредственно в запуске приложения (инструменты сборки, исходные файлы, временные артефакты, тестовые утилиты), не должно присутствовать в финальном этапе.
6. Финальный этап должен содержать только необходимое для запуска контейнера: минимальную базовую систему, собранные артефакты, зависимости для runtime и конфигурацию команды запуска.

* Каждый этап начинается с **From** и только слои последнего (финального) From попадут в image.
* Финальный этап использует все ресурсы подготовительных этапов с помощью COPY и обращению к каждому этапу с помощью --from= <stage\_index>
* Каждому подготовительному этапу можно задать алиас с помощью **AS**, вместо использования индексов слоев
* В результате уменьшается кол-во слоев (учитываются только слои финального stage), размер финального образа (не тянутся ресурсы, которые нужны для подготовки), что ускоряет процесс создания Docker image.

Лучшие практики:

1. использовать multi stage
2. грамотное использование кэширования – порядок исполнения инструкций, объединять инструкции в блоки (объединять зависимые инструкции), установка ПО всегда вначале, т.к. они тяжеловесные и занимают много времени, минимизировать кол-во слоев (если возможно объединять логически связанные команды в 1 RUN)
3. Уменьшать Build context – копировать только необходимые файлы, использовать .dockerignore
4. Одна entrypoint инструкция на image – один процесс один image
5. Не использовать latest, т.к. он динамический и будет подтягиваться новый с каждой новой версией

**Docker Compose**

Yaml файл, который описывает всю инфраструктуру, необходимую для запуска приложения многоконтейнерного приложения

1. Version – версия синтаксиса compose
2. services – описание запускаемых контейнеров
   1. build – инструкция для сборки образа из dockerfile
      1. context
      2. dockerfile
   2. image – указание метаинформации для image (имя, тэг)
   3. ports – маппинг портов сервиса
   4. environment/env\_file – декларация переменных окружения ключ=значение или файл
   5. volumes – подключение контейнера к тому
   6. restart – политика перезапуска контейнера при сбоях
   7. healthcheck – проверка состояния контейнера
   8. depends\_on – декларирует порядок запуска контейнера
3. volumes – глобальное определение томов
4. networks – глобальное определение сетей
5. logging – настройка логирования контейнера
6. profiles
7. secrets/config

**Как использовать переменные окружения**

1. Передать (pass) через командную строки (command line) с командой run с флагом -e
2. Передать .env файл через –env-file
3. Прописать прям в докерфайле (ENV)
4. Указать в docker-compose (тоже самое или перечислить или передать файл)

**Docker Swarm** – встроенный инструмент для оркестрации

* 1. Кластеризация – объединяет несколько docker-хостов в единый кластер
  2. Масштабирование сервисов – уменьшить или увеличить кол-во контейнеров сервиса
  3. Балансировка нагрузки
  4. High availability – нода (хост машина - сервер) или контейнер упали, поднимаются или перезапускаются новые