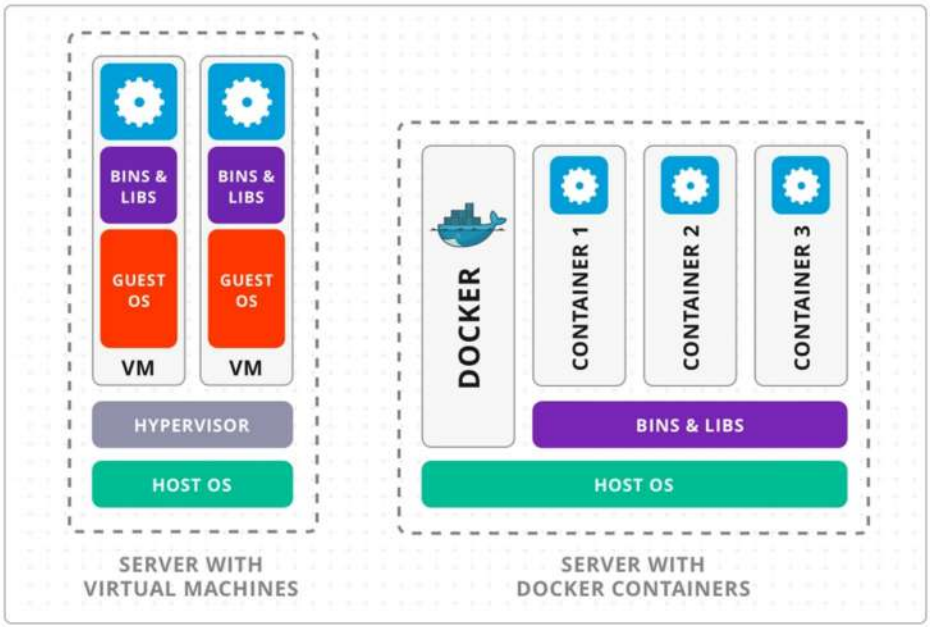
Основы работы с Docker

Docker – это платформа контейнеризации, которая упрощает развертывание и запуск приложений в изолированной среде. В Docker приложение вместе со всеми системными библиотеками и зависимостями упаковывается в контейнер [1]. Такой подход гарантирует одинаковое поведение приложения на разных машинах: локальный ПК разработчика, тестовый сервер или продакшн. Docker решает проблему «работает у меня» тем, что полностью воспроизводит окружение для приложения внутри контейнера [2].

До появления контейнеризации основным способом изоляции приложений были виртуальные машины (ВМ). Каждая ВМ эмулирует целый компьютер со своей ОС, драйверами и файловой системой. Для управления несколькими ВМ используется гипервизор, который распределяет ресурсы хоста между машинами [3]. Это надежно, но накладно: ВМ занимают гигабайты пространства и запускаются долго. Docker же работает по-другому – он использует контейнеризацию на уровне ОС [4]. Контейнеры разделяют ядро хостовой системы, но имеют собственное пространство процессов, файлов и сетей. Иными словами, **виртуальная машина** изолирует железо и поднимает полную ОС, а **контейнер** изолирует только процессы в рамках одной ОС [5]. Такое решение значительно легче: контейнеры занимают меньше места и стартуют почти мгновенно.



На схеме ниже показано отличие виртуальной машины и Docker-контейнера: виртуальная машина включает в себя гостевую ОС поверх гипервизора, а контейнер запускается «поверх» ядра хоста, используя общие ресурсы. Контейнер делит с хостом одно ядро, поэтому он более легковесен и быстро стартует по сравнению с полноценной ВМ [4].

Зачем нужен Docker

Docker получил широкое распространение благодаря ряду преимуществ, которые он даёт при разработке и эксплуатации приложений: - **Стабильное окружение:** контейнер гарантирует

одинаковую среду запуска везде. Всё, что нужно для работы, уже собрано в образе, поэтому приложение не зависит от настроек конкретной машины [6].
- **Лёгкость и скорость:** контейнеры используют общее ядро ОС и не требуют отдельной установки системы, поэтому занимают мало места и запускаются за секунды [7]. На одном сервере может работать сразу десяток контейнеров без потери производительности.
- **Масштабирование:** при возрастании нагрузки можно просто запустить дополнительные копии контейнера. Система быстро перераспределит ресурсы, не требуя изменений в коде [8].
- **Изоляция процессов:** каждый контейнер работает независимо. Сбой или утечка памяти в одном контейнере не затронет другие и не приведёт к падению всей системы [9].
- **Интеграция в CI/CD:** контейнеры являются стандартом в современных пайплайнах. Одно и то же приложение можно собрать, протестировать и запустить в одинаковой среде — от локальной машины до продакшена. Это снижает количество неожиданных багов, ускоряет и упрощает выпуск релизов [10].

Архитектура Docker

В основе Docker лежит **Docker Engine** – система, состоящая из трёх ключевых компонентов:
- **CLI (Command Line Interface)** – клиентский интерфейс, через который пользователь вводит команды.
- **Docker Daemon (dockerd)** – фоновый сервис, который выполняет команды пользователя: создаёт, запускает и удаляет контейнеры.
- **REST API** – веб-интерфейс, через который Docker CLI общается с демонном. В локальной установке CLI напрямую общается с `dockerd` через сокет, реализованный по REST-архитектуре [11].

Когда вы вводите команду типа `docker run`, клиент Docker передаёт запрос демону `dockerd`, который проверяет наличие нужного образа, готовит окружение, настраивает сеть и создаёт контейнер [12]. Внутри `dockerd` задействованы вспомогательные компоненты: **containerd** (отвечает за жизненный цикл контейнера – создание, запуск, остановку, удаление) и **runc** (собственно запускает контейнер, используя встроенные в Linux механизмы изоляции и контроля ресурсов: namespaces и cgroups) [13]. Оба следуют стандартам OCI (Open Container Initiative), благодаря чему Docker-образы совместимы с другими инструментами контейнеризации (например, Podman или CRI-O) [14] [15].

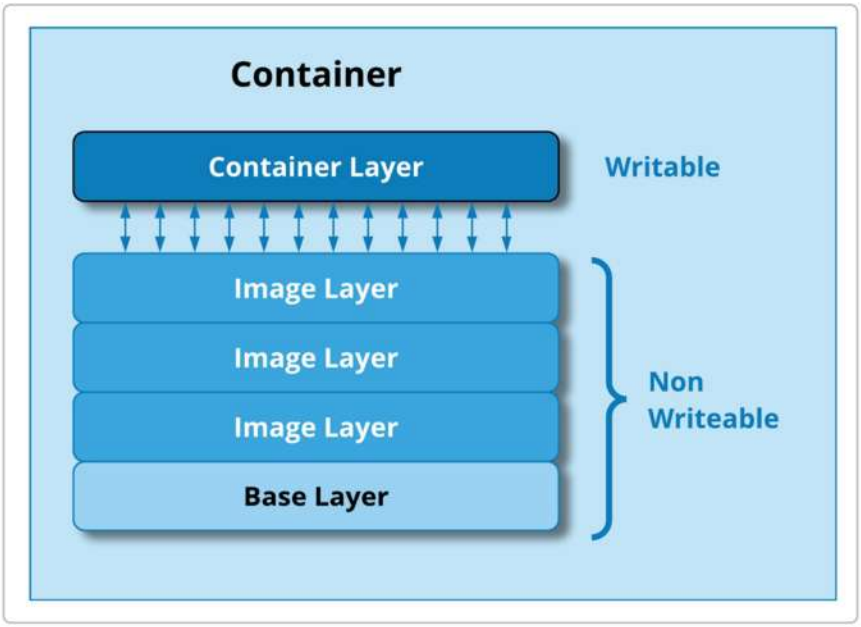
Вся цепочка выглядит так: CLI → Docker Daemon (`dockerd`) → containerd → runc → процессы внутри контейнера. При этом работающие контейнеры не зависят от `dockerd`: если демон перезапустить, контейнеры продолжат работу (этого добивается `containerd` как «прослойка») [16]. Таким образом, Docker Engine надёжно оркестрирует все части системы, обеспечивая изоляцию и управление контейнерами.

Docker Desktop

На Windows и macOS отсутствует встроенная контейнеризация ядра, поэтому Docker запускается через приложение **Docker Desktop**. Это настольное приложение, которое устанавливает внутри лёгкой виртуальной машины полноценный Linux с Docker Engine [17]. Пользователь получает привычный CLI и графический интерфейс, все настройки сети, проброс портов и управление томами настраиваются автоматически. Для новичков Docker Desktop — самый простой способ начать работу: достаточно установить приложение с официального сайта и сразу приступить к экспериментам [18].

Работа с образами

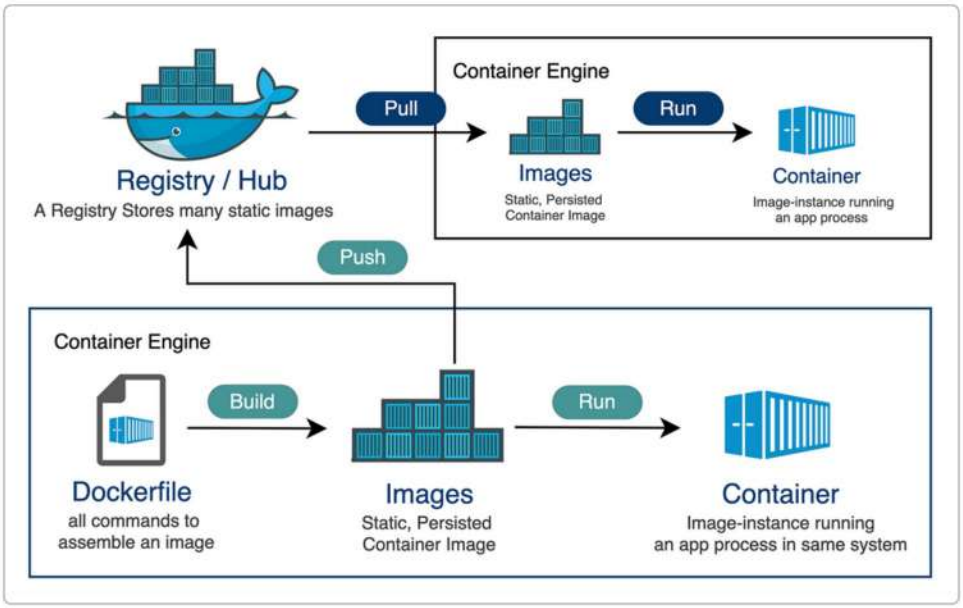
Контейнеры запускаются из **образов (image)** – неизменяемых шаблонов, в которых уже собраны все зависимости, библиотеки и настройки для приложения. Образ можно воспринимать как «снимок» файловой системы, готовый к запуску. Docker-образы состоят из **слоёв**: каждый слой соответствует одной инструкции сборки. При запуске контейнера Docker поверх образа создаёт дополнительный слой для изменений, оставляя исходный образ нетронутым [19] [20]. Это позволяет многократно использовать один образ для разных контейнеров без повторного скачивания и сборки общих частей. Docker хранит слои на диске по принципу наложения (overlay2-драйвер), поэтому слои одного образа могут переиспользоваться другими образами, экономя дисковое пространство [21] [22].



Образы Docker состоят из слоёв, которые накладываются друг на друга. При сборке `Dockerfile` каждый шаг создаёт новый слой, а Docker по возможности берёт слои из кэша, если исходный слой не изменился [23]. Такой механизм ускоряет сборку: правки, внесённые ниже по Dockerfile (например, обновлённый исходный код), не затрагивают верхние слои, и Docker повторно не пересоздаёт неизменившиеся слои.

Хранилища образов (Registry)

Docker-образы хранятся в **реестрах (registry)** – серверах, которые принимают и отдают образы по запросу. Самый известный публичный реестр – **Docker Hub**. В нём размещены миллионы готовых образов, включая официальные (например, nginx, Redis, Python, PostgreSQL) и пользовательские [24]. Docker Hub также поддерживает приватные репозитории для команд и компаний. Для корпоративных нужд можно развернуть собственный Docker Registry – сервер, совместимый по API с Docker Hub, но с контролем доступа и версионированием образов. Это удобно, когда образы содержат закрытый код или настройки, которые нельзя публиковать в общем доступе [25].



На схеме приведён основной жизненный цикл Docker-образа. Сначала образ **собирается** из Dockerfile и сохраняется локально, затем при необходимости его можно **отправить** (`docker push`) в реестр (Docker Hub или локальный Registry), а на другой машине **скачать** (`docker pull`) и **запустить** контейнер из него [26].

Каждый образ в реестре состоит из манифеста и набора слоёв [27]. Манифест – это файл-описание, в котором указывается, какие слои входят в образ. Само содержимое слоя хранится отдельно (как архив с уникальным SHA256). При загрузке или скачивании Docker проверяет контрольные суммы слоёв: если хотя бы один байт файла отличается, загрузка считается повреждённой и прерывается [28]. Это гарантирует целостность образа. Образы помечаются **тегами** (например, `latest`, `v1.0`, `dev`), которые помогают различать версии. Однако за каждым тегом всегда стоит неизменный **digest** (полный хэш образа) – он однозначно идентифицирует конкретную сборку. Даже если пользователь перезапишет тег другой версией, digest останется прежним [29].

Сеть контейнеров

При запуске контейнера Docker автоматически подключает его к виртуальной сети. По умолчанию создаётся **bridge-сеть**: каждый контейнер получает собственный IP и может общаться с другими контейнерами в этой сети [30]. Для контейнера такая сеть выглядит как отдельная машина с собственным сетевым интерфейсом и маршрутами. Docker на стороне хоста поднимает виртуальный интерфейс, назначает подсеть и настраивает NAT, чтобы контейнеры могли выходить в Интернет.

Если приложению нужно принимать запросы извне, при запуске указывается флаг `-p`, чтобы пробросить порт хоста внутрь контейнера. Например, опция `-p 8080:80` делает порт 80 контейнера доступным по адресу `localhost:8080` на хосте [31].

Кроме стандартной **bridge**-сети у Docker есть и другие режимы: - **host** – контейнер работает в сетевом пространстве хоста (использует его IP и интерфейсы). Такой режим не изолирован, зато даёт максимальную производительность и прямой доступ к сети [32].
- **none** – сеть полностью отключена. Контейнер не получает IP, не выходит в интернет и не принимает внешних соединений [33]. Подойдёт для задач, не требующих сети.

- **overlay** – объединяет контейнеры на разных хостах в одну виртуальную сеть. Используется для распределённых приложений, когда контейнеры должны взаимодействовать между несколькими серверами [34].
- **macvlan** – каждому контейнеру даётся собственный MAC-адрес, и в локальной сети он виден как отдельное устройство [35]. Это может пригодиться, если контейнер должен полноценно взаимодействовать с другими машинами в сети.

Сеть контейнеров управляется Docker Daemon'ом: он создаёт интерфейсы, подключает контейнеры и автоматически настраивает правила маршрутизации. Благодаря этому контейнеры изолированы друг от друга, но при необходимости могут обмениваться данными по чётким настройкам сети [36].

Логи и события

Все процессы внутри контейнера работают так же, как в обычной Linux-системе. То, что они выводят в стандартные потоки (`stdout` и `stderr`), Docker перенаправляет в систему логирования. По умолчанию используется драйвер **json-file**: логи каждого контейнера сохраняются на хосте в JSON-файле [37]. Такой формат удобен для локальной отладки и просмотра командой `docker logs`. В продакшен-средах часто подключают другие драйверы: например, **journald** (запись в системный журнал), **syslog** (отправка на внешний сервер) или **fluentd** (сбор и анализ в распределённых системах) [38]. Для каждого контейнера можно настроить параметры логирования: выбрать нужный драйвер, ограничить размер файлов и настроить ротацию (автоматическую архивацию и удаление старых записей) [39].

Docker также фиксирует **события** – всё, что происходит в системе контейнеров: запуск и остановка контейнеров, создание и удаление образов, подключение томов и т. д. Эти события можно отслеживать через API или команду `docker events`. В реальных сценариях события часто используют для мониторинга и интеграции: например, CI/CD-система может реагировать на появление нового контейнера или образование ошибки и автоматически запускать тесты или уведомлять команду [40]. Такой подход делает работу Docker прозрачно – можно в любой момент узнать, что и когда произошло с контейнерами, без непосредственного вмешательства.

Безопасность

Docker использует механизмы ядра Linux для изоляции контейнеров. В основе лежат **namespaces** и **cgroups**. Namespaces создают отдельные пространства для процессов, пользователей, сети и файлов внутри каждого контейнера [41]. Контейнер видит только свои процессы и каталоги, он не «видит» другие контейнеры или системные ресурсы хоста. Например, PID 1 внутри контейнера никак не связан с PID 1 хостовой системы [42]. Cgroups (control groups) ограничивают ресурсы: с их помощью задаются лимиты по CPU, памяти и операциям ввода/вывода для контейнера [43]. Если приложение внутри контейнера станет требовать слишком много ресурсов, cgroups ограничат его, не давая нарушить работу хоста или других контейнеров.

Помимо базовых механизмов изоляции, Docker поддерживает и дополнительные средства безопасности. **Capabilities** позволяют тонко регулировать права root-пользователя внутри контейнера (например, запретить изменение сетевых настроек) [44]. Инструменты контроля доступа AppArmor и SELinux задают, какие файлы и операции разрешены процессам контейнера [44]. Фильтр системных вызовов **seccomp** блокирует потенциально опасные обращения ядра (например, загрузку модулей ядра) [45]. Для повышения безопасности Docker можно запускать в **rootless**-режиме – тогда контейнеры работают не от имени администратора хоста, а под обычным пользователем. Это снижает риски для хоста, хотя функциональность контейнеров при

этом немного ограничена [46]. Все эти уровни вместе дают хорошую защиту: даже в случае уязвимости внутри контейнера злоумышленник не сможет получить больше прав, чем ему изначально разрешено.

На чём написан Docker

Docker написан на языке **Go** [47]. Именно на Go реализованы все основные компоненты Docker Engine: `dockerd` (демон), `containerd` (менеджер контейнеров), `runc` (низкоуровневый рантайм) и **BuildKit** (движок сборки образов) [47]. Язык Go выбран из-за простоты работы с параллелизмом и возможности создавать статически скомпилированные двоичные файлы без внешних зависимостей. Docker придерживается стандартов OCI (Open Container Initiative), поэтому его образы могут использоваться и в других системах контейнеризации [15]. Исходный код Docker распространяется под лицензией Apache 2.0 [48]: её условия разрешают использовать, изменять и распространять код Docker в любых проектах (кроме обязательного сохранения уведомлений об авторских правах и текста лицензии) [48]. Название и логотип Docker являются зарегистрированными товарными знаками и не могут использоваться без разрешения.

Основные понятия Docker

Ниже приведены ключевые термины и определения, чтобы было проще ориентироваться в Docker:

- **Образ (Image)**

  Образ – это шаблон (снимок), из которого запускаются контейнеры. Образ содержит всё необходимое для работы приложения: базовую файловую систему, системные библиотеки, зависимости, ваш код и настройки. Образ при запуске не меняется – Docker использует его как основу и добавляет сверху изменяемый слой контейнера. Благодаря этому один и тот же образ можно использовать многократно без изменения исходного кода [49].

- **Контейнер (Container)**

  Контейнер – это изолированная среда выполнения приложения. Контейнер запускается на основе образа и может работать, быть остановлен и удален независимо от других контейнеров. При удалении контейнера всё, что было в его слое изменений (данные, логи и т. д.), исчезает – поэтому для постоянного хранения данных используют **тома** [50]. Контейнер по сути – это процессы приложения со своей средой, изолированной от остальной системы.

- **Том (Volume)**

  Том – специальное хранилище для данных, которые должны сохраняться между перезапусками контейнеров. По сути том монтируется в контейнер как внешний диск: всё, что записано в том, остаётся на хосте и не теряется при удалении контейнера [51]. Через тома обычно хранят базы данных, логи, конфиги и другие важные файлы.

- **Сеть (Network)**

  Сеть Docker отвечает за связь между контейнерами и внешний мир. Docker по умолчанию создаёт bridge-сеть, где каждый контейнер получает свой IP-адрес [52]. Контейнеры в

одной сети могут обращаться друг к другу по имени (Docker поднимает встроенный DNS), а для связи с внешним миром используются порт-пробросы. Можно создавать дополнительные сети или изолировать контейнеры друг от друга при необходимости, задавая для них отдельные сетевые пространства [52].

- **Реестр (Registry)**

  Реестр – это место, где хранятся Docker-образы. Публичный реестр – Docker Hub, на котором располагаются миллионы готовых образов (официальных и пользовательских) [53]. Если образы должны оставаться внутри организации, разворачивают свой локальный Docker Registry – тот же сервер хранения, но с ограниченным доступом. Registry позволяет эффективно передавать образы между машинами и хранить различные версии образов.

Установка и примерные команды

Для работы с Docker сначала нужно установить Docker Engine. Он включает в себя демон `dockerd`, клиент CLI (`docker`) и инструменты для работы с образами и сетями [54]. На **macOS** и **Windows** установка производится через приложение Docker Desktop. Его можно скачать с официального сайта (docker.com) и установить как обычную программу [55]. На **Linux** Docker обычно ставится из официальных репозиториев дистрибутива. Например, в Debian/Ubuntu достаточно выполнить:

```
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io
```

(подробные инструкции есть в документации Docker [56]).

После установки важно убедиться, что Docker Engine запущен. Команда `docker version` выводит информацию о клиенте и сервере (демоне). В выводе должны присутствовать оба раздела – *Client* и *Server* – это означает, что `dockerd` работает корректно [57].

Для первоначальной проверки можно запустить тестовый контейнер с помощью образа `hello-world`. Достаточно выполнить:

```
docker run hello-world
```

Docker скачает тестовый образ с Docker Hub и запустит контейнер, который выведет в консоль «Hello from Docker!» [58]. Если вы видите это сообщение, значит установка прошла успешно и демон реагирует на команды.

Запуск первого контейнера

После установки можно попробовать запустить реальное приложение в контейнере. Например, один из самых простых тестов – запустить веб-сервер Nginx. Команда

```
docker run -d -p 8080:80 nginx
```

сделает следующее [59] [60]:
1. `docker run` создаёт новый контейнер из указанного образа (`nginx`). Если образа нет локально, Docker автоматически скачает его с Docker Hub.
2. Флаг `-d` запускает контейнер в фоновом режиме (detached mode).
3. Параметр `-p 8080:80` пробрасывает порт: порт 80 внутри контейнера связывается с портом 8080 на хост-машине.

После запуска Docker выведет идентификатор контейнера. Затем достаточно открыть в браузере страницу **http://localhost:8080**, чтобы увидеть стандартную стартовую страницу Nginx [61]. Это означает, что веб-сервер внутри контейнера работает и доступен снаружи.

Чтобы убедиться, что контейнер действительно запущен, можно использовать команду:

```
docker ps
```

Она покажет список активных контейнеров, их идентификаторы, образы, статус и проброшенные порты. В списке вы увидите контейнер с именем `nginx` и статусом `Up`, что подтверждает его успешный запуск [62]. (Если добавить флаг `-a`, `docker ps` также покажет остановленные контейнеры – это помогает обнаружить старые тестовые экземпляры или выяснить причины их остановки.)

Интерактивный запуск контейнера

Иногда нужно не просто запустить контейнер, а **зайти внутрь** него и выполнить команды. Для этого используют флаги `-i` и `-t`. Например:

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

Этот запуск создаст контейнер на основе образа `ubuntu` и запустит внутри него оболочку Bash [63]. Флаг `-i` оставляет открытым стандартный ввод контейнера, а `-t` выделяет псевдотерминал. После выполнения команды вы окажетесь в консоли контейнера (под индикатором приглашения, обычно `root@…`), где можно выполнять любые Linux-команды.

Для выхода из контейнера достаточно ввести `exit` [64]. Если нужно подключиться к уже запущенному контейнеру (например, зайти внутрь уже работающего Ubuntu), используется `docker exec`:

```
docker exec -it <id> bash
```

Эта команда запустит Bash внутри активного контейнера с указанным идентификатором. Таким образом можно «заглянуть» в контейнер после его запуска.

Как следить за контейнерами

Docker постоянно отслеживает состояние запущенных контейнеров. Удобный способ посмотреть текущий статус – та же команда `docker ps`, которая показывает активные контейнеры [65]. Если надо получить полную информацию о конкретном контейнере (сетевые настройки, переменные

окружения, точная команда запуска и пр.), можно выполнить:

```
docker inspect <id>
```

Команда выведет полное JSON-описание контейнера, включая все детали его конфигурации [66].

Чтобы быстро увидеть вывод приложения внутри контейнера без захода в него, можно использовать:

```
docker logs <id>
```

Она покажет последние строчки, которые контейнер «написал» в stdout/stderr [67].

Управление контейнерами

Контейнерами управлять так же просто, как процессами. Основные команды:
- `docker stop <id>` – аккуратно останавливает контейнер (отправляет сигнал SIGTERM, затем SIGKILL) [68].
- `docker start <id>` – запускает ранее остановленный контейнер снова [69].
- `docker rm <id>` – удаляет контейнер (его данных). Если контейнер запущен, Docker не позволит удалить его без флага `-f` (force) [70].
- `docker rm -f <id>` – принудительно останавливает и удаляет контейнер одной командой [71].

Помимо контейнеров, Docker хранит локальные образы. Основные команды для работы с образами:
- `docker images` – показывает список сохранённых образов [72].
- `docker rmi <image>` – удаляет ненужный локальный образ [73].
- `docker system prune` – убирает всё, что не используется (остановленные контейнеры, неиспользуемые образы, неактивные сети) [74]. Это освобождает место на диске и оставляет только то, что нужно в данный момент.

Создание образа: Dockerfile

Готовые образы подходят во многих случаях, но в реальных проектах часто требуется свой уникальный образ: с собственным кодом, библиотеками и настройками. Для этого используется **Dockerfile** – текстовый файл, в котором пошагово описано, как собрать образ. Рассмотрим пример `Dockerfile`:

```
FROM python:3.11-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Что делает каждая инструкция (приведено в примере выше) [75]:
- `FROM` – задаёт базовый образ (его имя и версию), от которого «унаследуется» ваш образ. В

примере базовый образ – `python:3.11-slim`.
- `WORKDIR` – задаёт рабочую директорию внутри контейнера (при необходимости создаст её). В примере рабочая папка – `/app`.
- `COPY . .` – копирует все файлы из текущей директории на хосте в текущую директорию образа (`/app`). Так проект попадает внутрь образа.
- `RUN` – выполняет команды при сборке образа. В примере `RUN pip install -r requirements.txt` устанавливает зависимости из файла `requirements.txt`. Каждый `RUN` создаёт новый слой в образе.
- `CMD` – указывает команду по умолчанию, которая будет выполнена при запуске контейнера. Здесь это `python app.py`.

Важный момент: в контейнере по умолчанию все процессы работают с правами `root`. В учёных целях это обычно допустимо, но в продакшене такой подход небезопасен. Чтобы снизить риски, в Dockerfile часто добавляют собственного пользователя и переключаются на него. Например:

```
RUN useradd -m appuser
USER appuser
```

Теперь приложение внутри контейнера будет запускаться от обычного пользователя `appuser`, а не от root [76]. Это усложняет злоумышленнику получение доступа к хосту через уязвимости контейнера.

Dockerfile поддерживает две инструкции для запуска процесса при старте контейнера: **CMD** и **ENTRYPOINT**. Разница в следующем [77]:
- `CMD` задаёт команду (или аргументы) по умолчанию. Если при запуске контейнера указать свои аргументы, они заменят `CMD`.
- `ENTRYPOINT` задаёт фиксированную часть команды, которую Docker всегда выполнит, а любые аргументы при запуске просто добавятся к ней.

Например:

```
ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8000"]
```

Здесь `ENTRYPOINT` устанавливает, что контейнер всегда запускает `python app.py`, а `CMD` по умолчанию добавляет к этой команде аргумент `--port 8000`. То есть по умолчанию при запуске выполнится:

```
python app.py --port 8000
```

Если же при старте контейнера указать свои параметры, Docker объединит их с `ENTRYPOINT`, заменив `CMD`. Например, команда `docker run myapp --port 9000` приведёт к выполнению:

```
python app.py --port 9000
```

Такой приём удобен, когда контейнер всегда должен запускать одно и то же приложение, а параметры можно менять по желанию [78] [79].

Практический пример

Рассмотрим полный пример создания и запуска своего первого контейнера. Допустим, у нас есть папка проекта `myapp` со следующими файлами:

```
myapp/
├── app.py
├── requirements.txt
└── Dockerfile
```

В файле `app.py` находится простейшее веб-приложение на Python, которое отвечает строкой `Hello from Docker!` на любые HTTP-запросы (используется встроенный модуль `http.server`). Пример содержания `app.py`:

```
from http.server import HTTPServer, BaseHTTPRequestHandler

class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello from Docker!")

if __name__ == "__main__":
    HTTPServer(("0.0.0.0", 8000), Handler).serve_forever()
```

Файл `requirements.txt` можно оставить пустым, если дополнительных библиотек нет.

Сборка образа

Перейдите в директорию `myapp` и запустите сборку образа командой:

```
docker build -t myapp .
```

Здесь `-t myapp` задаёт имя (тег) образа, а `.` указывает путь к контексту сборки (папке с Dockerfile). Docker прочитает `Dockerfile` и выполнит инструкции последовательно, в результате создав новый локальный образ `myapp` [80].

Запуск контейнера

После сборки запустите контейнер из своего образа:

```
docker run -d -p 8000:8000 myapp
```

Это создаст контейнер, внутри которого запустится ваш `app.py`. Благодаря проброску портов приложение будет доступно по адресу `http://localhost:8000` на вашей машине [81]. Если перейти по этому адресу в браузере, вы увидите текст **Hello from Docker!** – значит, контейнер успешно работает.

Вы только что создали собственный образ и убедились, что контейнер работает. В реальных проектах может быть несколько версий образа (например, для тестов, продакшена и т. д.). Чтобы различать их, Docker использует теги. Если при сборке образа не указывать тег, Docker присвоит ему тег `latest`. Но лучше явно указывать версии, например:

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

Так вы получите образ `myapp:1.0`. Обретя несколько тегов (например, `1.0`, `dev`, `latest`), можно удобно различать стабильные и экспериментальные варианты образа [82].

Передача и хранение образов

После сборки образ локально доступен только на вашей машине. Чтобы поделиться им с коллегами или использовать на другом компьютере, его нужно загрузить в реестр. Наиболее простой вариант – Docker Hub. Пошагово:
1. **Создайте репозиторий на Docker Hub.** Зайдите в свой аккаунт на hub.docker.com и создайте новый репозиторий (аналогично созданию проекта на GitHub) [83]. Имя репозитория должно совпадать с тем, что вы будете использовать при загрузке.
2. **Переименуйте (отметьте) локальный образ.** Docker-образ нужно «тагировать» в формате `username/название:тег`. Например:

```
docker tag myapp:1.0 username/myapp:1.0
```

Здесь `username` – ваш логин на Docker Hub, `repository` – имя репозитория, `tag` – версия образа. Docker сам по имени определит, куда отправлять образ [84].
3. **Авторизуйтесь в Docker Hub:**

```
docker login
```

Docker запросит ваш логин и пароль. После успешного входа вы увидите сообщение `Login Succeeded` [85].
4. **Загрузите образ в репозиторий:**

```
docker push username/myapp:1.0
```

Docker начнёт отправку образа на Docker Hub [86]. По завершении загрузки он появится в вашем аккаунте и станет доступен другим (если репозиторий публичный) [87].

Теперь образ доступен из любого места. На другой машине или сервере вы можете скачать его командой:

```
docker pull username/myapp:1.0
```

Docker скачает только те слои, которых нет локально, – повторно не потребуется загружать то, что уже сохранено [88]. Затем вы можете запустить контейнер из этого образа привычным способом:

```
docker run -d -p 8000:8000 username/myapp:1.0
```

Docker Compose: что это и зачем нужно

Когда приложение растёт, одного контейнера обычно недостаточно. Помимо основного сервиса могут потребоваться база данных, кеш, очередь сообщений и т. д. Запускать несколько контейнеров вручную бывает неудобно. **Docker Compose** упрощает эту задачу. С помощью файла `docker-compose.yml` можно описать набор всех требуемых контейнеров и их настройки, а затем запустить их одной командой [89].

Простой пример файла `docker-compose.yml`:

```
services:
  web:
    build: .
    ports:
      - "8000:8000"

  redis:
    image: redis:alpine
```

Здесь определены два сервиса: `web` (собирается из текущей директории и слушает порт 8000) и `redis` (просто запускается из официального образа `redis:alpine`) [90]. Чтобы собрать и запустить всё вместе, сохраняем файл и выполняем:

```
docker compose up
```

Docker Compose автоматически соберёт образы, создаст контейнеры, подключит их к сети и запустит все сервисы [91]. Чтобы остановить проект, используется:

```
docker compose down
```

Это закроет все контейнеры и удалит созданные сети, но сохранит образы для повторного запуска.