

# ЛЕКЦИЯ № 1

## Введение в Docker и контейнеризацию

Группа СА-24 · 4 курс

Дисциплина: системное администрирование

Полный текст занятия для преподавателя  
с формулировками для конспекта и демонстрационными командами

Актуализировано: 2 сентября 2026 года

## Как пользоваться этой лекцией

Этот документ рассчитан на первое занятие по Docker. Его можно читать почти последовательно как готовый сценарий. Обычный текст — это то, что преподаватель объясняет устно. Голубые блоки «Под запись» — короткие определения и тезисы, которые имеет смысл продиктовать студентам для конспекта. Серые блоки с командами — демонстрации в терминале. Жёлтые блоки — методические подсказки преподавателю.

При спокойном темпе, с вопросами аудитории и показом команд материал рассчитан примерно на одну стандартную пару — около 90 минут. Если группа активно отвечает и вы подробно разбираете демонстрации, лекция легко растягивается на два академических часа. Разделы о Dockerfile, сетях и данных можно использовать как резерв или перенести на начало следующего занятия.

### Цели занятия

- Понять, какую проблему решает контейнеризация и почему Docker стал стандартным инструментом администратора и разработчика.
- Различать виртуальную машину, образ и контейнер.
- Понимать общую архитектуру Docker Engine и роль Docker CLI, dockerd, containerd и runc.
- Понять, на каких механизмах ядра Linux основана изоляция контейнеров.
- Освоить базовый жизненный цикл контейнера и основные команды Docker CLI.
- Получить первое представление о сетях, томах, Dockerfile и безопасной эксплуатации контейнеров.

### Ориентировочный тайминг

| Блок                                  | Время     |
|---------------------------------------|-----------|
| Введение и проблема воспроизводимости | 10 мин    |
| Контейнеры и виртуальные машины       | 12 мин    |
| Архитектура Docker                    | 15 мин    |
| Image, container, registry            | 15 мин    |
| Демонстрация жизненного цикла         | 18 мин    |
| Сети и данные                         | 10 мин    |
| Dockerfile, безопасность, выводы      | 10–20 мин |

# 1. Вступление: зачем системному администратору Docker

Сегодня мы начинаем отдельный блок, посвящённый Docker. До этого момента вы уже сталкивались с операционными системами, службами, сетями, пакетными менеджерами, виртуальными машинами и серверным программным обеспечением. Docker объединяет сразу несколько этих тем, потому что контейнер — это одновременно процесс, файловая система, сетевое окружение, набор ограничений ресурсов и способ доставки приложения.

Начнём не с команды `docker run`, а с проблемы. Представьте, что разработчик приносит вам веб-приложение и говорит: «На моём компьютере всё работает». Вам нужно развернуть его на сервере. Вы выясняете, что приложению нужна конкретная версия Python, отдельная версия библиотеки OpenSSL, PostgreSQL определённого выпуска, Redis и ещё пять библиотек. На сервере уже есть другие приложения, которым нужны другие версии этих компонентов. Возникают конфликты зависимостей, различия конфигураций и вопрос: как гарантировать, что среда в тестировании и в эксплуатации действительно одинаковая?

Исторически эту проблему решали несколькими способами. Можно было подробно документировать установку: поставить ОС, затем пакеты, затем вручную изменить конфигурационные файлы. Позже появились системы управления конфигурацией — Ansible, Puppet, Chef. Они сделали развёртывание воспроизводимым, но само приложение всё ещё работает в общей операционной системе вместе с другими приложениями. Ещё один путь — для каждого сервиса создавать отдельную виртуальную машину. Это даёт сильную изоляцию, но требует заметно больше ресурсов и времени на запуск.

Контейнеризация предложила другой подход: упаковать приложение вместе с его пользовательскими зависимостями в стандартный переносимый образ, а запускать это приложение как изолированный процесс. В результате администратор получает не набор инструкций «установите двадцать пакетов и поправьте семь файлов», а объект, который можно одинаково запустить на ноутбуке, тестовом сервере и в инфраструктуре предприятия при наличии совместимой контейнерной платформы.

**Под запись: Контейнеризация — это способ запуска приложений в изолированных пользовательских пространствах, использующих ядро операционной системы хоста. Контейнер упаковывает приложение и необходимые ему зависимости, но обычно не содержит отдельного ядра ОС.**

Фраза «использует ядро хоста» очень важна. Именно она отличает контейнер от классической виртуальной машины. Контейнер не является «маленькой виртуальной машиной». Внутри контейнера может находиться файловая структура, похожая на Ubuntu или Alpine, но это не означает, что внутри работает собственное ядро Linux. Системные вызовы процессов контейнера в конечном итоге обслуживает ядро хостовой системы.

Для системного администратора Docker важен как минимум в четырёх сценариях. Первый — развёртывание приложений. Второй — создание одинаковых тестовых и производственных сред. Третий — изоляция сервисов и их зависимостей. Четвёртый — автоматизация CI/CD, когда образ строится один раз и затем проходит по цепочке тестирование — реестр — эксплуатация.

*Преподавателю: Спросите группу: «Что означает фраза “у меня работает, а на сервере нет”? Какие причины они уже встречали?» Обычно называют версии библиотек, переменные окружения, права, конфигурацию, порты. Это естественно подводит к идее воспроизводимой среды.*

## 2. Что такое Docker и что он не делает

Docker — это не один исполняемый файл и не синоним слова «контейнер». Docker — это платформа и набор инструментов для сборки образов, их хранения и распространения, а также создания и управления контейнерами. Центральной серверной частью является Docker Engine.

**Под запись: Docker Engine — клиент-серверная платформа контейнеризации. Она включает демон Docker, API и клиент командной строки docker. Демон управляет объектами Docker: образами, контейнерами, сетями и томами.**

Важно отделить технологию контейнеров от конкретного продукта Docker. Контейнеры в Linux существовали до Docker. Docker сделал их удобными, дал понятный формат образов, инструменты сборки, реестр и привычный интерфейс управления. Сегодня контейнерная экосистема опирается и на стандарты Open Container Initiative — OCI. Благодаря стандартам образ, созданный одним инструментом, во многих случаях может запускаться другим совместимым runtime.

То есть Docker не изобрёл саму идею изоляции процессов. Его историческая заслуга в том, что контейнеры стали удобным массовым способом упаковки и доставки приложений.

Также Docker не является полноценной системой оркестрации уровня Kubernetes. Docker умеет запускать контейнеры на отдельном хосте, а Docker Compose удобно описывает несколько связанных контейнеров. Но когда речь идёт о десятках серверов, автоматическом распределении нагрузки, масштабировании, самовосстановлении и управлении большим кластером, появляются оркестраторы. До них мы дойдём позже.

## 3. Контейнер и виртуальная машина: принципиальная разница

Сравним два подхода. В виртуализации у нас есть физический сервер, гипервизор и несколько виртуальных машин. Каждая виртуальная машина видит виртуальное оборудование и запускает собственную гостевую операционную систему со своим ядром. Внутри этой ОС уже работают приложения.

В контейнерной модели есть хостовая операционная система и её ядро. Контейнеры работают как изолированные процессы этого ядра. У каждого контейнера может быть собственное представление процессов, сети, точек монтирования и файловой системы, но ядро остаётся общим.

**Под запись: Виртуальная машина виртуализирует компьютер и запускает собственную гостевую ОС. Контейнер изолирует процессы на уровне операционной системы и разделяет ядро хоста с другими контейнерами.**

Отсюда следуют практические свойства контейнеров. Они обычно запускаются значительно быстрее виртуальных машин, потому что им не нужно загружать отдельную гостевую ОС. Образы контейнеров обычно меньше образов виртуальных дисков. На одном сервере можно разместить больше контейнеризированных сервисов при сопоставимой нагрузке. Однако изоляция контейнеров и виртуальных машин устроена по-разному, поэтому контейнер нельзя автоматически считать более безопасной заменой виртуальной машине во всех сценариях.

Есть и ограничение по ядру. Linux-контейнер рассчитывает на возможности ядра Linux. На обычном Windows-хосте Linux-контейнер не может просто взять и использовать ядро Windows как Linux-ядро. Поэтому Docker Desktop на Windows и macOS использует дополнительный слой

виртуализации, чтобы предоставить Linux-окружение для Linux-контейнеров. На Linux-сервере Docker Engine может работать непосредственно поверх Linux.

### Сравнение в одной таблице

| Характеристика       | Виртуальная машина                     | Контейнер                                           |
|----------------------|----------------------------------------|-----------------------------------------------------|
| Что изолируется      | Виртуальный компьютер                  | Процессы и их окружение                             |
| Ядро ОС              | Свое у каждой VM                       | Общее с хостом                                      |
| Старт                | Обычно дольше                          | Обычно секунды или доли секунды                     |
| Размер               | Часто гигабайты                        | Часто десятки/сотни МБ, иногда меньше               |
| Плотность размещения | Ниже                                   | Обычно выше                                         |
| Типичный сценарий    | Разные ОС, сильная аппаратная изоляция | Сервисы, микросервисы, CI/CD, воспроизводимые среды |

На практике эти технологии не конкурируют по принципу «должна остаться только одна». Очень часто Docker запускают внутри виртуальных машин. В облаке вы можете получить виртуальный сервер, установить на него Docker Engine и уже внутри виртуальной машины запускать десятки контейнеров. VM решает задачу инфраструктурной изоляции и границ сервера, а контейнеры — задачу упаковки и жизненного цикла приложений.

## 4. Что происходит внутри Linux: namespaces и cgroups

Чтобы понимать Docker как администратор, нужно знать два ключевых механизма ядра Linux: namespaces и cgroups. Docker использует не магию, а возможности ядра.

### 4.1. Namespaces: разные процессы видят разный мир

Namespace можно представить как механизм, который изменяет представление процесса о системе. Два процесса могут работать на одном ядре, но видеть разные наборы PID, разные сетевые интерфейсы или разные точки монтирования.

Основные пространства имён, с которыми полезно познакомиться:

- PID namespace — изоляция идентификаторов и дерева процессов. Процесс внутри контейнера может видеть себя как PID 1, хотя на хосте у него будет другой PID.
- Network namespace — собственные сетевые интерфейсы, IP-адреса, маршруты и таблицы сетевого стека.
- Mount namespace — собственное представление точек монтирования файловых систем.
- UTS namespace — отдельное имя хоста и доменное имя.
- IPC namespace — изоляция механизмов межпроцессного взаимодействия.
- User namespace — сопоставление пользователей и идентификаторов UID/GID между контейнером и хостом.

**Под запись: Namespaces обеспечивают изоляцию представления системных ресурсов: процессов, сети, файловых систем, имени хоста, IPC и пользователей. Контейнер видит собственное окружение, хотя процессы выполняются общим ядром Linux.**

Например, мы запускаем контейнер с `nginx`. Внутри контейнера команда `ps` может показать всего несколько процессов. На хосте эти же процессы существуют в общей таблице процессов, но контейнер из-за `PID namespace` не видит большинство процессов хоста.

## 4.2. `cgroups`: сколько ресурсов можно потребить

`Namespaces` отвечают в первую очередь на вопрос «что процесс видит?». `Control groups`, или `cgroups`, отвечают на другой вопрос: «сколько ресурсов ему можно использовать и как учитывать это использование?». С помощью `cgroups` ядро может группировать процессы и применять к ним ограничения по `CPU`, памяти, количеству процессов и другим ресурсам.

**Под запись: `cgroups` — механизм ядра Linux для группировки процессов, учёта и ограничения потребления ресурсов. В Docker через `cgroups` можно ограничивать `CPU`, память и другие ресурсы контейнера.**

Если никаких ограничений не задать, контейнер не получает автоматически «один процессор и один гигабайт». Он потенциально может конкурировать за ресурсы хоста с другими процессами. Поэтому на производственном сервере важно задавать разумные лимиты там, где это необходимо.

```
docker run --rm --memory 256m --cpus 0.5 nginx:alpine
```

В этой команде контейнер получает ограничение памяти 256 МБ и квоту примерно на половину одного `CPU`. Детали планирования сложнее, но на первом занятии важно понять принцип: контейнер лёгкий не потому, что ресурсы «берутся из воздуха», а потому что мы не запускаем отдельное ядро и можем управлять группами обычных процессов.

## 5. Архитектура Docker Engine

Теперь соберём картину уровней Docker. Когда вы вводите команду `docker ps`, клиент `docker` не ходит напрямую по процессам Linux. Клиент обращается к `Docker API`. На Linux обычно запрос принимает демон `dockerd`. Он управляет объектами Docker и координирует нижележащие компоненты.

Современная архитектура контейнеров разделена на уровни. Для понимания достаточно следующей упрощённой цепочки:

**Под запись: Упрощённая цепочка запуска: Docker CLI → Docker daemon (dockerd) → containerd → OCI runtime (обычно runc) → процесс контейнера в ядре Linux.**

Docker CLI — клиент. Он может находиться на том же хосте, что и демон, или обращаться к удалённому Docker Engine через API. `dockerd` принимает команды высокого уровня: скачать образ, создать контейнер, подключить сеть, смонтировать том. `containerd` занимается жизненным циклом контейнеров и управлением контейнерными объектами более низкого уровня. `runc` — низкоуровневый OCI-совместимый runtime, который непосредственно создаёт контейнерное окружение с нужными `namespaces`, `cgroups` и другими параметрами и запускает процесс.

На экзамене или собеседовании не нужно изображать Docker как монолит. Полезно понимать, что команда `docker run` запускает цепочку действий, а конечный результат — обычный процесс в Linux, которому ядро создало изолированное окружение.

## 5.1. Docker daemon как привилегированная точка управления

Docker daemon традиционно работает с высокими привилегиями, потому что ему нужно создавать сетевые интерфейсы, управлять пространствами имён, монтировать файловые системы и выполнять другие системные операции. Поэтому доступ к Docker API или к сокету `/var/run/docker.sock` фактически является очень серьёзным правом.

Если пользователь входит в группу `docker` и может свободно отдавать команды демону, его возможности на обычной конфигурации близки к `root`-доступу. Например, он способен запустить контейнер с монтированием файловой системы хоста. Поэтому добавление человека в группу `docker` — это не просто «разрешить одну удобную команду», а важное решение по безопасности.

**Под запись:** Доступ к Docker daemon необходимо защищать. На обычной `rootful`-конфигурации членство в группе `docker` следует рассматривать как привилегированный доступ к хосту.

Существует `rootless`-режим Docker, при котором и демон, и контейнеры работают без `root`-привилегий с использованием `user namespace`. Он уменьшает последствия некоторых уязвимостей, хотя имеет свои ограничения и особенности.

## 6. Главные объекты Docker: image, container, registry

### 6.1. Image — образ

Образ — это шаблон, из которого создаётся контейнер. В нём находятся файлы приложения, бинарные файлы, библиотеки и конфигурация, необходимые для запуска. Образ не нужно воспринимать как «архив работающего сервера». Это неизменяемый набор слоёв плюс метаданные о том, как запускать процесс.

**Под запись:** Docker image — неизменяемый шаблон, содержащий файловую систему и метаданные, необходимые для создания контейнера. Образы состоят из слоёв.

Слово «неизменяемый» означает, что после сборки мы не должны редактировать образ как обычную установленную систему. Если нужен новый пакет или новый файл, строится новый образ. Docker при этом может переиспользовать неизменившиеся слои.

Слои — один из ключей к эффективности Docker. Если несколько образов основаны на одном базовом слое, хранить одинаковые данные многократно обычно не требуется. Во время сборки отдельные инструкции `Dockerfile` формируют слои или влияют на конфигурацию образа. На уровне стандартов OCI образ содержит манифест, конфигурацию и упорядоченный набор слоёв файловой системы.

### 6.2. Container — контейнер

Контейнер — это запущенный или остановленный экземпляр образа с собственной конфигурацией и изменяемым слоем. Из одного образа можно создать много контейнеров. Это похоже на отношение «класс — экземпляр» только как учебная аналогия: образ задаёт основу, контейнер — конкретный экземпляр выполнения.

**Под запись:** Container — экземпляр образа с собственным состоянием выполнения, сетевой конфигурацией и записываемым слоем файловой системы. Один образ может быть основой для множества контейнеров.

Важно различать остановку и удаление. `docker stop` останавливает процесс, но объект контейнера остаётся, и его можно снова запустить командой `docker start`. `docker rm` удаляет контейнер как объект. Если данные были записаны только в его writable layer, после удаления контейнера рассчитывать на них нельзя.

### 6.3. Registry и repository

Образы нужно где-то хранить и передавать. Для этого используют container registry — реестр контейнерных образов. Docker Hub — самый известный публичный реестр, но существуют GitHub Container Registry, GitLab Container Registry, Harbor, облачные реестры и частные корпоративные решения.

**Под запись: Registry — сервис хранения и распространения контейнерных образов. Repository — логическая коллекция версий одного образа внутри registry. Tag — удобная метка версии или варианта образа.**

Например, в имени `nginx:1.28` `nginx` — имя репозитория, а `1.28` — tag. Если registry явно не указан, Docker обычно использует настроенный реестр по умолчанию, часто Docker Hub. Метка `latest` — это просто обычный тег с особым соглашением по умолчанию, а не гарантия того, что перед вами «самая новая и лучшая» версия.

Теги изменяемы: владелец репозитория может перепривязать один и тот же tag к другому содержимому. Для полной воспроизводимости в критичных сценариях используют более жёсткую фиксацию версии, вплоть до `digest` — криптографического идентификатора содержимого.

## 7. Первый жизненный цикл контейнера: от pull до rm

*Преподавателю: Здесь лучше открыть терминал на Linux-хосте. Даже если студенты пока не повторяют команды, показывайте вывод после каждого шага. Используйте `nginx:alpine` или `nginx:stable-alpine` — компактный образ и понятный сетевой сервис.*

### 7.1. Проверяем клиент и сервер

```
docker version
docker info
```

`docker version` показывает версии клиента и сервера, если связь с демоном установлена. `docker info` выводит общую информацию о Docker Engine: количество контейнеров и образов, storage driver, cgroup driver, доступные runtimes, параметры хоста и другие сведения.

Если команда `docker` есть, но демон недоступен, клиент может вывести ошибку соединения. Это хороший пример клиент-серверной архитектуры: наличие CLI не означает, что Docker Engine запущен и доступен.

### 7.2. Скачиваем образ

```
docker pull nginx:alpine
```

Команда `pull` обращается к `registry`, получает манифест образа и необходимые слои. Если часть слоёв уже хранится локально, повторно скачивать их не нужно. После завершения можно посмотреть список локальных образов.

```
docker image ls
# короткая форма той же идеи:
docker images
```

В выводе обычно видны `repository`, `tag`, `image ID`, время создания и размер. `Image ID` — идентификатор образа. При этом полный современный образ внутри реестра описывается несколькими `content-addressed` объектами, поэтому не надо сводить всю модель к одной строке `ID`.

### 7.3. Создаём и запускаем контейнер

```
docker run --name web1 -d nginx:alpine
```

Разберём команду по частям. `docker run` означает «создать новый контейнер из образа и запустить его». `--name web1` задаёт удобное имя. `-d` означает `detached mode` — контейнер работает в фоне. `nginx:alpine` — выбранный образ.

Если локального образа нет, `docker run` по умолчанию может сначала выполнить `pull`. То есть одна команда скрывает несколько шагов: найти образ, при необходимости скачать, создать контейнерную конфигурацию, подготовить файловую систему и сеть, создать изоляцию и запустить главный процесс.

**Под запись: `docker run` создаёт новый контейнер и запускает его. `docker start` запускает уже существующий остановленный контейнер. Это разные операции.**

### 7.4. Смотрим состояние

```
docker ps
docker ps -a
```

`docker ps` показывает работающие контейнеры. Ключ `-a` добавляет остановленные. Обратите внимание на `CONTAINER ID`, `IMAGE`, `COMMAND`, `CREATED`, `STATUS`, `PORTS` и `NAMES`. Для администратора колонка `STATUS` особенно важна: контейнер может быть создан, запущен, перезапущаться или завершён с кодом ошибки.

Контейнер живёт пока живёт его главный процесс. Если `PID 1` внутри контейнера завершился, контейнер считается остановленным. Это принципиально отличается от привычной идеи «виртуальная машина работает сама по себе». Контейнер — это способ запуска процесса, а не мини-компьютер, который обязан постоянно быть включён.

### 7.5. Логи

```
docker logs web1
docker logs -f web1
```

Команда `logs` получает журнал стандартного вывода и стандартного потока ошибок контейнера в зависимости от используемого `logging driver`. Ключ `-f` продолжает следить за новыми строками.

Приложения в контейнерах желательно проектировать так, чтобы рабочие логи удобно попадали в stdout и stderr, а уже контейнерная платформа собирала и перенаправляла их.

## 7.6. Выполняем команду в работающем контейнере

```
docker exec -it web1 sh
```

docker exec запускает дополнительный процесс внутри уже работающего контейнера. -i оставляет стандартный ввод открытым, -t выделяет псевдотерминал. В Alpine-образах часто используется sh вместо bash.

Внутри можно показать:

```
hostname
cat /etc/os-release
ps
ip addr 2>/dev/null || true
exit
```

Здесь полезно задать студентам вопрос: если cat /etc/os-release показывает Alpine Linux, означает ли это, что контейнер загрузил своё ядро Alpine? Нет. Файл /etc/os-release находится в файловой системе образа и описывает пользовательское пространство дистрибутива. Ядро можно проверить по uname: оно будет связано с ядром хоста.

```
docker exec web1 uname -a
uname -a
```

Результаты не обязаны быть посимвольно одинаковыми из-за настроек и окружения, но смысл один: системные вызовы контейнера обслуживает ядро хоста.

## 7.7. Подробная конфигурация

```
docker inspect web1
```

docker inspect выводит JSON с подробным описанием контейнера: идентификаторы, образ, состояние, параметры запуска, mounts, сетевые настройки, IP-адрес, переменные окружения и многое другое. Это одна из самых полезных диагностических команд администратора.

При необходимости можно извлечь конкретное поле форматированием:

```
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' web1
```

## 7.8. Останавливаем, запускаем, удаляем

```
docker stop web1
docker ps -a
docker start web1
docker stop web1
docker rm web1
```

docker stop отправляет процессу сигнал корректного завершения и ждёт установленный интервал, после чего при необходимости завершает жёстче. docker start не создаёт новый контейнер, а запускает существующий. docker rm удаляет остановленный контейнер.

Для одноразовых контейнеров полезен ключ `--rm`: объект автоматически удалится после завершения процесса.

```
docker run --rm alpine:latest echo "Hello from container"
```

**Под запись:** Базовый жизненный цикл: pull образа → run контейнера → `ps/logs/inspect/exec` для управления и диагностики → `stop` → `start` при необходимости → `rm` для удаления контейнера.

## 8. Порты и сеть контейнеров

Запустим `nginx` ещё раз, но теперь захотим открыть его из браузера. Если просто выполнить `docker run -d nginx`, `nginx` слушает порт 80 внутри сетевого namespace контейнера. Это не означает, что порт 80 автоматически опубликован на внешнем интерфейсе хоста.

**Под запись:** EXPOSE в образе и открытый порт внутри контейнера не равны публикации порта на хосте. Для доступа извне обычно используется `docker run -p HOST_PORT:CONTAINER_PORT`.

```
docker run --name web1 -d -p 8080:80 nginx:alpine
```

Теперь обращение к порту 8080 хоста перенаправляется на TCP-порт 80 контейнера. На Linux Docker создаёт необходимые правила маршрутизации и фильтрации. В стандартной bridge-сети используется NAT/port mapping.

Очень важно читать запись справа налево: сервис `nginx` слушает 80 внутри контейнера, а мы публикуем его как 8080 на хосте. Если записать `-p 80:8080`, то Docker будет ждать, что приложение внутри контейнера слушает 8080.

```
curl http://127.0.0.1:8080
```

По умолчанию публикация `-p 8080:80` без конкретного адреса может сделать порт доступным на всех интерфейсах хоста. Это потенциально означает доступ из внешней сети. Если сервис должен быть доступен только локально, можно привязать публикацию к `loorback`-интерфейсу:

```
docker run --rm -d -p 127.0.0.1:8081:80 nginx:alpine
```

**Под запись:** Публикация порта — это изменение сетевой доступности сервиса. Перед использованием `-p` администратор должен понимать, на каком адресе хоста будет слушаться порт и кому он станет доступен.

### 8.1. Bridge-сеть и DNS между контейнерами

Docker умеет создавать пользовательские bridge-сети. Контейнеры в одной такой сети получают сетевые интерфейсы и могут обращаться друг к другу. В пользовательской сети Docker

предоставляет встроенное разрешение имён контейнеров, поэтому зависимостям не нужно жёстко прописывать динамические IP-адреса.

```
docker network create demo-net
docker run -d --name web --network demo-net nginx:alpine
docker run --rm --network demo-net alpine:latest sh -c "wget -qO- http://web | head"
```

Второй контейнер обращается к имени web, а не к IP. Это базовая идея, на которой позже удобно строить многоконтейнерные приложения в Docker Compose.

```
docker rm -f web
docker network rm demo-net
```

## 9. Данные: почему нельзя хранить всё внутри контейнера

По умолчанию изменения файлов внутри контейнера записываются в его writable layer, расположенный поверх неизменяемых слоёв образа. Этот слой принадлежит конкретному контейнеру. Контейнер можно остановить и снова запустить — изменения останутся. Но если контейнер удалить и создать новый из исходного образа, старого writable layer уже нет.

Для stateless-приложения это удобно: контейнер должен быть легко заменяемым. Но база данных, загруженные пользователями файлы, сертификаты или иные постоянные данные не должны зависеть от существования конкретного контейнера.

**Под запись: Данные, которые должны пережить удаление и пересоздание контейнера, следует хранить вне writable layer контейнера — например, в Docker volume или bind mount.**

### 9.1. Docker volume

Volume — управляемое Docker постоянное хранилище. Оно имеет отдельный жизненный цикл и может быть подключено к контейнеру. Удаление контейнера само по себе не обязано удалять named volume.

```
docker volume create webdata
docker volume ls
docker run --rm -v webdata:/data alpine sh -c "date > /data/created.txt"
docker run --rm -v webdata:/data alpine cat /data/created.txt
```

Первый контейнер создал файл и был удалён, второй контейнер увидел тот же файл, потому что данные находились в volume.

### 9.2. Bind mount

Bind mount напрямую связывает конкретный путь хоста с путём внутри контейнера. Это удобно для конфигурационных файлов, исходного кода в разработке и ситуаций, когда файлы должны быть доступны и обычным процессам хоста.

```
mkdir -p ~/docker-demo-html
echo "<h1>Hello SA-24</h1>" > ~/docker-demo-html/index.html
docker run --rm -d --name webbind -p 8082:80 \
-v ~/docker-demo-html:/usr/share/nginx/html:ro nginx:alpine
```

Ключ `:ro` монтирует каталог только для чтения из контейнера. Это хороший принцип минимальных привилегий: если сервису не нужно изменять данные, не давайте ему запись.

Volumes и bind mounts решают разные задачи. Volume лучше отделяет данные от структуры каталогов хоста и управляется Docker. Bind mount нужен, когда контейнер должен работать с конкретным файлом или каталогом хоста.

## 10. Dockerfile: как появляется собственный образ

До этого мы использовали готовые образы. Но реальная ценность Docker начинается, когда мы создаём образ своего приложения. Описание сборки обычно хранится в текстовом файле Dockerfile.

**Под запись: Dockerfile — декларативное описание шагов сборки контейнерного образа. Команда `docker build` читает Dockerfile и контекст сборки и создаёт новый image.**

Рассмотрим максимально простой пример статического сайта на nginx.

```
mkdir docker-first-image
cd docker-first-image
echo "<h1>SA-24 Docker</h1>" > index.html
```

Создадим Dockerfile:

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
```

Первая инструкция FROM задаёт базовый образ. COPY добавляет наш файл. Собираем:

```
docker build -t sa24-web:1.0 .
```

Точка в конце команды — это путь к build context. Docker получает доступ к файлам этого контекста, и COPY может использовать их при сборке. После сборки запустим новый образ:

```
docker run --rm -d --name sa24-web -p 8083:80 sa24-web:1.0
```

Теперь важная мысль: если мы хотим изменить сайт, правильная модель — изменить исходный файл, пересобрать образ с новой версией и заменить контейнер. Плохая модель — зайти в работающий контейнер, вручную отредактировать файлы и считать эту ручную операцию способом развёртывания.

**Под запись: Контейнеры следует рассматривать как заменяемые экземпляры. Изменения приложения должны фиксироваться в исходниках и Dockerfile, после чего строится новый образ, а не вручную «допиливается» работающий контейнер.**

### 10.1. Несколько важных инструкций Dockerfile

- FROM — базовый образ.
- WORKDIR — рабочий каталог для последующих инструкций и процесса.
- COPY — копирование файлов из build context в образ.

- RUN — выполнение команды во время сборки образа.
- ENV — переменная окружения в образе.
- EXPOSE — документирование ожидаемого порта приложения; само по себе порт на хосте не публикует.
- USER — пользователь, от имени которого запускаются последующие инструкции/процесс.
- CMD — команда по умолчанию при запуске контейнера.
- ENTRYPOINT — основной исполняемый файл/точка входа, которую сложнее полностью заменить аргументами запуска.

На следующих занятиях мы отдельно разберём кеш сборки, порядок инструкций, `.dockerignore`, multi-stage builds и уменьшение размеров образов.

## 11. Контейнер как процесс: PID 1, сигналы и завершение

У контейнера есть главный процесс. В его PID namespace этот процесс обычно получает PID 1. Для Linux PID 1 имеет особое значение: он участвует в обработке сигналов и должен корректно работать с дочерними процессами. Поэтому образ должен запускать приложение так, чтобы основной процесс получал сигналы от Docker и корректно завершался.

Когда мы выполняем `docker stop`, Docker сначала пытается остановить контейнер корректно, посылая сигнал основному процессу. Если приложение игнорирует завершение, платформа спустя тайм-аут может принудительно остановить его. Для баз данных и сервисов это критично: корректное завершение позволяет дописать данные и освободить ресурсы.

Отсюда появляется инженерный принцип: один контейнер — одна основная ответственность. Это не строгий физический закон: внутри контейнера могут появляться дочерние процессы. Но не следует превращать контейнер в полноценную виртуальную машину, где одновременно запускаются `sshd`, `cron`, база, веб-сервер и ещё пять независимых сервисов только потому, что «так привычнее».

**Под запись: Типичная модель контейнера: один основной сервис или одна ответственность на контейнер. Независимые компоненты приложения обычно разделяют на отдельные контейнеры и связывают сетью.**

## 12. Безопасность Docker: что нужно усвоить с первого занятия

Контейнеризация даёт изоляцию, но контейнер не является абсолютной границей безопасности. Все Linux-контейнеры на хосте в конечном итоге взаимодействуют с одним ядром, а конфигурация контейнера может сильно расширить его привилегии.

### 12.1. Не использовать `--privileged` без необходимости

Ключ `--privileged` резко расширяет права контейнера и открывает доступ к устройствам хоста. В учебных примерах в интернете его иногда добавляют, чтобы «всё заработало». Для администратора это сигнал остановиться и разобраться, какое именно разрешение действительно требуется.

### 12.2. Не монтировать `docker.sock` в недоверенные контейнеры

Монтирование `/var/run/docker.sock` внутрь контейнера фактически даёт программе возможность управлять Docker-демоном хоста. Если приложение с таким доступом скомпрометировано, последствия могут затронуть весь Docker-хост.

### 12.3. По возможности не запускать приложение как root

Если сервис способен работать непривилегированным пользователем, образ следует настроить через USER или использовать другой безопасный способ. Это уменьшает последствия компрометации процесса. Root внутри контейнера не всегда полностью эквивалентен root на хосте из-за namespaces и ограничений, но игнорировать этот риск нельзя.

### 12.4. Минимизировать образ и доверять источнику

Чем больше пакетов и инструментов внутри образа, тем больше потенциальная поверхность атаки и сложнее обновление. Базовые образы следует выбирать осознанно, регулярно пересобирать, обновлять и проверять. Не стоит запускать неизвестный образ только потому, что он найден по подходящему имени.

### 12.5. Публиковать только необходимые порты

Команда -p может открыть сервис во внешнюю сеть. Порт базы данных не нужно публиковать на всех интерфейсах, если к базе обращается только соседний контейнер. Разделение внутренних и внешних сетевых связей — часть безопасной архитектуры.

### 12.6. Ограничивать ресурсы

Контейнер без лимитов способен израсходовать память или CPU и повлиять на другие сервисы. Ограничения ресурсов — не только оптимизация, но и элемент устойчивости и безопасности.

**Под запись: Контейнер — не «песочница, которой можно доверять всё». Безопасная эксплуатация требует минимальных привилегий, контроля образов, сетевой доступности, мониторингов, ресурсов и доступа к Docker daemon.**

## 13. Типичные ошибки новичков

### Ошибка 1. «Контейнер — это облегчённая виртуальная машина»

Такое объяснение допустимо как очень грубая бытовая аналогия, но технически ведёт к неправильной модели. Контейнер — изолированный процесс или группа процессов, использующих ядро хоста.

### Ошибка 2. «Если остановить контейнер, все данные исчезнут»

Не совсем. После stop writable layer контейнера остаётся. Данные исчезают как проблема именно при удалении и пересоздании, если они не вынесены в постоянное хранилище. Поэтому важно различать stop и rm.

### Ошибка 3. «latest означает последнюю версию»

latest — всего лишь тег, который может указывать на то содержимое, которое выбрал издатель. Для контролируемых развёртываний версии фиксируют осознанно.

### Ошибка 4. «EXPOSE 80 делает сайт доступным снаружи»

EXPOSE — метаданные и документация ожидаемого порта. Публикация на хост выполняется через -p/--publish или соответствующую конфигурацию Compose.

## Ошибка 5. «Нужно зайти в контейнер и настроить его как обычный сервер»

Вручную зайти через `docker exec` для диагностики можно. Но повторяемое изменение приложения должно попадать в `Dockerfile`, конфигурацию, секреты, `volume` или систему развёртывания. Иначе контейнер теряет главное преимущество — воспроизводимость.

## Ошибка 6. «Docker автоматически делает приложение безопасным»

Нет. Небезопасное приложение остаётся небезопасным. Слишком привилегированный контейнер может быть очень опасным. Docker даёт механизмы изоляции и контроля, но результат зависит от конфигурации.

## 14. Мини-демонстрация для закрепления: один контейнер от начала до конца

*Преподавателю: Если времени осталось 10–15 минут, выполните весь блок без длинных объяснений и попросите студентов вслух комментировать, что делает каждая команда.*

```
docker pull nginx:alpine
docker run -d --name lesson-web -p 8080:80 nginx:alpine
docker ps
docker logs lesson-web
docker exec lesson-web nginx -v
docker inspect lesson-web
curl http://127.0.0.1:8080
docker stop lesson-web
docker start lesson-web
docker rm -f lesson-web
```

Попросите студентов объяснить: где здесь `image`, где `container`, когда контейнер создаётся, чем `start` отличается от `run`, зачем нужен `-p`, почему `nginx` внутри слушает 80, а снаружи мы идём на 8080, что останется после `docker rm`, и откуда при первом запуске берётся образ.

## 15. Вопросы группе по ходу занятия и в конце

Эти вопросы можно задавать устно. Они проверяют не память команд, а правильную модель Docker.

1. Если внутри контейнера команда `cat /etc/os-release` показывает Ubuntu, обязательно ли ядро хоста тоже Ubuntu?
2. Чем `docker run` отличается от `docker start`?
3. Почему контейнер обычно запускается быстрее виртуальной машины?
4. Какой механизм Linux отвечает за изоляцию таблицы процессов и сетевого стека?
5. Какой механизм позволяет ограничить контейнер по памяти и CPU?

6. Что произойдёт с приложением, если его главный процесс завершится?
7. Чем image отличается от container?
8. Что такое registry и зачем он нужен?
9. Почему для базы данных опасно рассчитывать только на writable layer контейнера?
10. Что означает -p 8080:80?
11. Почему открытие Docker socket недоверенному контейнеру опасно?
12. Почему ручное изменение файлов через docker exec — плохой способ выпускать новую версию приложения?

## 16. Итог лекции: что студент должен унести с первого занятия

Сегодня нам важно не запомнить десятки ключей командной строки, а построить правильную картину. Docker — это способ упаковки и запуска приложений в контейнерах. Контейнеры используют механизмы ядра Linux и не несут отдельное ядро, как классическая виртуальная машина. Изоляцию обеспечивают namespaces, ограничения ресурсов — cgroups. Docker Engine предоставляет удобный клиент-серверный интерфейс управления образами, контейнерами, сетями и данными.

Образ — неизменяемая основа и набор слоёв. Контейнер — конкретный экземпляр запуска этого образа с writable layer и собственной конфигурацией. Образы передаются через registry. Сервис внутри контейнера не становится автоматически доступен извне: публикация портов управляется отдельно. Постоянные данные нужно отделять от жизненного цикла контейнера с помощью volumes или bind mounts. Собственные образы описываются Dockerfile и должны пересобираваться воспроизводимо.

И последнее: хороший системный администратор не относится к контейнеру как к чёрному ящику. Он должен уметь ответить, какой процесс запущен, какой образ используется, какие порты опубликованы, какие данные смонтированы, в какой сети находится контейнер, какие у него лимиты и привилегии, откуда взять логи и как безопасно заменить его новой версией.

**Под запись: Docker — инструмент управления жизненным циклом контейнеризированных приложений. Для администратора ключевые сущности: image, container, registry, network и volume; ключевые механизмы Linux: namespaces и cgroups; ключевой принцип эксплуатации: контейнеры должны быть воспроизводимыми и заменяемыми.**

## 17. Краткий конспект, который можно продиктовать в конце занятия

Если в конце пары вы хотите дать студентам единый компактный блок для тетради, продиктуйте следующие пункты.

1. Docker — платформа контейнеризации для сборки, доставки и запуска приложений в изолированных окружениях.
2. Контейнер — изолированный процесс или группа процессов, использующих ядро ОС хоста. В отличие от VM, контейнер обычно не содержит собственного ядра.
3. Изоляция в Linux строится главным образом на namespaces; учёт и ограничения ресурсов — на cgroups.

4. Docker Engine имеет клиент-серверную архитектуру: docker CLI обращается к Docker daemon. Для запуска контейнеров используются нижележащие компоненты containerd и OCI runtime, например runc.
5. Image — неизменяемый послойный шаблон. Container — экземпляр image с состоянием запуска и writable layer.
6. Registry хранит и распространяет образы. Repository группирует версии образа, tag обозначает вариант или версию.
7. docker run создаёт и запускает новый контейнер; docker start запускает существующий; docker stop останавливает; docker rm удаляет.
8. docker ps показывает контейнеры, docker logs — логи, docker exec — запускает дополнительный процесс внутри работающего контейнера, docker inspect — подробную конфигурацию.
9. -p HOST:CONTAINER публикует порт контейнера на хосте. Публикация порта влияет на сетевую доступность и должна контролироваться.
10. Постоянные данные выносятся в volumes или bind mounts. Writable layer контейнера не должен быть единственным местом хранения важных данных.
11. Dockerfile описывает воспроизводимую сборку собственного образа. Новая версия приложения должна приводить к новой сборке образа, а не к ручной правке контейнера.
12. Контейнеризация не отменяет безопасность: минимальные привилегии, доверенные образы, ограничения ресурсов, аккуратная публикация портов и защита Docker daemon обязательны.

## 18. Резервный материал: установка Docker Engine на Ubuntu Server

Этот раздел можно использовать, если Docker ещё не установлен на учебных виртуальных машинах, либо вынести в практическую работу. Команды ниже соответствуют официальному способу установки через репозиторий Docker для поддерживаемых версий Ubuntu на дату подготовки лекции.

Сначала удаляют конфликтующие пакеты, если они были установлены из других источников:

```
sudo apt remove docker.io docker-compose docker-compose-v2 docker-doc docker-buildx
podman-docker containerd runc
```

Добавляем официальный ключ и репозиторий Docker:

```
sudo apt update
sudo apt install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

sudo tee /etc/apt/sources.list.d/docker.sources > /dev/null <<'EOF'
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF

sudo apt update
```

Затем устанавливаем Docker Engine, CLI, containerd и современные плагины Buildx и Compose:

```
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
compose-plugin
```

Проверяем службу и тестовый контейнер:

```
sudo systemctl status docker
sudo docker run hello-world
```

В учебной среде часто хочется выполнять docker без sudo. Можно добавить пользователя в группу docker, но перед этим студентам нужно ещё раз проговорить: эта группа даёт очень высокий уровень доступа к хосту.

```
sudo usermod -aG docker $USER
# затем требуется новый вход в сеанс пользователя
```

Для серверов с повышенными требованиями безопасности следует отдельно изучить rootless mode и модель прав Docker daemon.

## 19. Резервный материал: что будет дальше

Следующее занятие логично посвятить более системной практике: сборке образов и Dockerfile, слоям и кешу, .dockerignore, переменным окружения, healthcheck, volumes, пользовательским bridge-сетям и Docker Compose. После этого можно переходить к эксплуатации нескольких сервисов: веб-приложение + база данных + reverse proxy, резервное копирование volumes, обновление образов и диагностика отказов.

Docker Compose вводит декларативное описание многоконтейнерного приложения в YAML-файле compose.yaml. Вместо длинного набора docker run мы описываем services, networks, volumes, environment и другие параметры, а затем приводим фактическое состояние к описанному командой docker compose up. Важно заранее различать два файла: Dockerfile отвечает за то, как построить image, а compose.yaml — за то, какие контейнеры и с какими параметрами нужно запустить.

## 20. Источники и литература для преподавателя

Содержание лекции сверено с актуальными материалами на 2 сентября 2026 года. В основном тексте ссылки намеренно не вставлены, чтобы лекция читалась как единый учебный материал.

- Docker Documentation. Get started / Docker concepts. <https://docs.docker.com/get-started/>
- Docker Documentation. Docker Engine. <https://docs.docker.com/engine/>
- Docker Documentation. What is a container?  
<https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/>
- Docker Documentation. What is an image?  
<https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/>
- Docker Documentation. Networking overview. <https://docs.docker.com/engine/network/>
- Docker Documentation. Port publishing and mapping. <https://docs.docker.com/engine/network/port-publishing/>
- Docker Documentation. Storage. <https://docs.docker.com/engine/storage/>
- Docker Documentation. Docker Engine security. <https://docs.docker.com/engine/security/>
- Docker Documentation. Rootless mode. <https://docs.docker.com/engine/security/rootless/>
- Docker Documentation. Building best practices. <https://docs.docker.com/build/building/best-practices/>

- Docker Documentation. Install Docker Engine on Ubuntu.  
<https://docs.docker.com/engine/install/ubuntu/>
- Open Container Initiative. Runtime Specification, current standards publication (2025).  
<https://specs.opencontainers.org/runtime-spec/>
- Open Container Initiative. Image Format Specification. <https://specs.opencontainers.org/image-spec/>
- Poulton, Nigel. Docker Deep Dive. Fourth Edition. Packt Publishing, 2025. 312 p. ISBN 9781837028344.

### Примечание об актуальности

Команды и концепции Docker меняются медленнее, чем интерфейсы Docker Desktop и номера пакетов, но установочные инструкции и поддерживаемые версии ОС со временем обновляются. Перед проведением практической установки на новом семестре проверяйте официальный раздел Install Docker Engine для вашей ОС. Для учебной теории придерживайтесь терминологии Docker Engine и OCI, а не случайных устаревших инструкций из блогов.