Лабораторная работа 11

Docker

Оглавление

Теоретическая часть

— Что общего у бомжа и девопса?
— Уверенность в завтрашнем дне?
— И те и другие роются в контейнерах. Этот вопрос мне задали в нескольких компаниях.
(*из недр IT-форумов*)

1. **Контейнеризация**

1.1. Понятие контейнера

Развитие технологий виртуализации началось ещё в 1960-х годах, когда на мэйнфреймах IBM появились первые механизмы разделения вычислительных ресурсов между несколькими пользователями. Позднее, с распространением персональных компьютеров и серверов, широкое распространение получили виртуальные машины (Virtual Machines, VM) — программные среды, полностью эмулирующие работу отдельного компьютера. Каждая виртуальная машина имеет собственное ядро операционной системы, виртуальные устройства, файловую систему и изолирована от других ВМ. Это обеспечивало высокий уровень безопасности и гибкости, но имело значительный недостаток — большие накладные расходы на ресурсы и длительное время запуска.

Следующим шагом в развитии технологий изоляции стали контейнеры. В отличие от виртуальных машин, контейнеры не эмулируют отдельную аппаратную платформу и не требуют установки собственной операционной системы. Вместо этого они используют ядро хостовой системы, создавая изолированные пользовательские пространства процессов. Это делает контейнеры гораздо более лёгкими и быстрыми в работе.

Контейнер можно рассматривать как упакованную среду выполнения приложения, которая включает в себя всё необходимое: код программы, библиотеки, зависимости, системные утилиты и конфигурационные файлы. Благодаря этому контейнер гарантирует одинаковое поведение приложения на любой системе, где установлен движок контейнеризации (например, Docker).

Ключевые принципы контейнеризации:

- **Изоляция процессов** — каждый контейнер работает в своём пространстве процессов, не мешая другим;
- **Портативность** — контейнер можно перенести и запустить на любой машине с поддержкой Docker или другой системы контейнеризации;
- **Лёгкость и скорость** — запуск контейнера занимает секунды, так как он использует уже загруженное ядро;
- **Воспроизводимость среды** — приложение работает одинаково в разработке, тестировании и продакшене.

Контейнеризация основана на возможностях ядра Linux, которые позволяют создавать такие изолированные среды выполнения без необходимости использовать полноценную виртуализацию. Главными механизмами, обеспечивающими работу контейнеров, являются **namespaces** и **cgroups**.

1.2. Namespaces

Одним из ключевых механизмов контейнеризации в Linux является **namespaces** — технология, обеспечивающая изоляцию ресурсов между группами процессов. Именно благодаря namespaces каждый контейнер видит только «свою» часть системы и не имеет доступа к пространствам других контейнеров или хостовой операционной системы.

Идея namespaces появилась в ядре Linux постепенно. Первые реализации (например, изоляция пространств монтирования) появились ещё в версии ядра 2.4, а к версии 3.x механизм стал достаточно зрелым, чтобы на его основе можно было реализовать полноценную контейнеризацию (что и сделали Docker и LXC).

Namespace создаёт **отдельное пространство имён** для определённого системного ресурса. Процесс, запущенный внутри такого пространства, «видит» только свои объекты — например, собственные процессы, сетевые интерфейсы или точки монтирования. Таким образом, namespaces позволяют создать иллюзию отдельной системы, хотя фактически все процессы работают под тем же ядром Linux.

В ядре Linux существует несколько типов пространств имён, используемых при создании контейнеров:

Тип	Назначение
UTS (Unix Timesharing System)	**Изолирует имя хоста и домен. Каждый контейнер может иметь собственное имя хоста.**
PID (Process ID)	**Изолирует идентификаторы процессов. Процессы в контейнере видят только свои PID, начиная с 1.**
Mount (mnt)	**Изолирует точки монтирования и файловую систему. Контейнер может иметь собственную структуру каталогов.**
Network (net)	**Изолирует сетевые интерфейсы, маршруты и IP-адреса. Каждый контейнер может иметь собственный сетевой стек.**
IPC (Inter-Process Communication)	**Изолирует объекты межпроцессного взаимодействия (очереди сообщений, семафоры, разделяемую память).**
User	**Изолирует идентификаторы пользователей и групп. Позволяет контейнеру иметь root-права внутри, не обладая ими на хосте.**
Cgroup	**Изолирует представление иерархий cgroups для контейнера.**

Пример

Для иллюстрации можно вручную создать изолированное пространство имён при помощи команды unshare:

```
sudo unshare --uts --ipc --pid --net --mount --fork mount-proc=/proc bash
```

После выполнения команды откроется новый экземпляр shell, где процессы, сеть и монтирование будут изолированы от основной системы. Если внутри такого окружения выполнить:

```
hostname container1
ps aux
```

то hostname изменится только в текущем пространстве, а список процессов будет содержать только процессы, запущенные внутри контейнера.

1.3. Cgroups

Если механизм **namespaces** отвечает за изоляцию процессов и ресурсов на логическом уровне, то **cgroups** (control groups) управляют тем, **сколько** ресурсов каждый контейнер может использовать. Cgroups позволяют ограничивать, распределять и отслеживать использование вычислительных ресурсов — процессорного времени, оперативной памяти, дисковых операций, сети и т. д.

Механизм cgroups был разработан в компании **Google** и впервые включён в ядро Linux в версии 2.6.24 (2008 год). Он стал важнейшей частью инфраструктуры контейнеризации, так как именно он обеспечивает контроль над ресурсами и предотвращает ситуации, когда одно приложение «захватывает» все вычислительные мощности системы.

Cgroups объединяют процессы в иерархические группы и позволяют ядру Linux применять к этим группам ограничения или правила распределения ресурсов. Каждая группа процессов может иметь собственные лимиты, приоритеты и параметры мониторинга.

Например:

- можно ограничить контейнер 512 МБ оперативной памяти;
- выделить не более 50% одного CPU;
- запретить использование определённого устройства ввода-вывода.

Каждый контейнер (или группа процессов) помещается в собственное дерево cgroups. При создании контейнера Docker автоматически формирует иерархию cgroups и применяет к ней параметры, заданные в команде запуска, например:

```
docker run --memory=512m --cpus=0.5 nginx
```

В этом примере контейнер ограничен 512 МБ оперативной памяти и половиной вычислительной мощности одного процессора. Ядро Linux следит за выполнением этих ограничений и, при необходимости, завершает процессы, которые выходят за установленные рамки (например, при превышении лимита памяти).

Cgroups — это фундамент, обеспечивающий **контроль и безопасность** в многоконтейнерной среде. Благодаря им контейнеры могут сосуществовать на одном хосте, не мешая друг другу, а системный администратор получает возможность точно регулировать распределение ресурсов.

Вместе с namespaces механизм cgroups создаёт основу контейнеров: первый отвечает за **изоляцию**, второй — за **ограничение и контроль**. Именно взаимодействие этих двух технологий делает возможной современную контейнеризацию, реализованную, в частности, в Docker.

2. Docker

2.1. Основы Docker

Docker is an open platform for developing, shipping and running applications.

(c) Документация

Docker — это открытая платформа для разработки, доставки и запуска приложений в контейнерах. Она позволяет разработчикам упаковывать приложения вместе со всеми зависимостями в единый **образ (image)**, а затем быстро и надёжно запускать его на любой машине, где установлен Docker. Благодаря Docker, одно и то же приложение будет работать **идентично** в средах разработки, тестирования и эксплуатации, что устраняет классическую проблему «на моей машине работает».

Проект Docker был создан в 2013 году компанией **dotCloud** (позже переименованной в Docker Inc.). Первоначально он основывался на технологии **LXC (Linux Containers)**, но затем разработчики создали собственный механизм управления — **libcontainer**, который позволил Docker напрямую взаимодействовать с функциями ядра Linux.

С этого момента Docker стал стандартом де-факто в области контейнеризации, а его идеи легли в основу многих современных инструментов оркестрации, таких как **Kubernetes** и **OpenShift**.

Плюсы использования Docker:

1. Легкое и быстрое развертывание
2. Экономичность потребления ресурсов (особенно в сравнении с ВМ)
3. Переносимость между разными системами
4. Легкость масштабирования
5. Хорошо подходит для микросервисной архитектуры
6. Доставка ПО в стандартизированной «коробке»
7. Изолированность от других процессов, что означает большую безопасность при одновременной работе множества процессов на хосте

Представим практическую ситуацию. Вы разрабатываете крупный программный проект, состоящий из нескольких модулей, написанных на языке **Python**. Каждый модуль использует сторонние библиотеки, например **NumPy**, но в разных версиях:

- первая программа — NumPy 1.24,
- вторая — NumPy 1.26,
- а в системе разработчика установлена последняя версия — NumPy 2.1.1.

В результате при выполнении одной из программ может возникнуть ошибка совместимости, так как она рассчитывает на старое поведение библиотеки, изменившееся в новой версии. Подобные конфликты версий зависимостей — частая проблема при разработке и сопровождении программных комплексов.

Использование **Docker** позволяет полностью избежать таких ситуаций. Приложение разворачивается в контейнере, где установлены только те версии библиотек и инструментов, которые ему действительно необходимы. Таким образом, Docker изолирует среду выполнения от системных настроек хоста и гарантирует, что программа будет работать одинаково на любом компьютере или сервере, независимо от установленного окружения.

2.2. Компоненты Docker

Платформа Docker состоит из нескольких взаимосвязанных компонентов, которые совместно обеспечивают полный цикл работы с контейнерами — от создания образа до его запуска и распространения. Архитектура Docker построена по модели клиент–сервер, где пользователь взаимодействует с системой через клиент, а основная работа выполняется серверным процессом — демоном Docker.

На рисунке 1 приведена схема работы Docker. Основным элементом является **Docker host** –компьютер или виртуальная машина на котором установлен сам Docker.

Docker daemon – это служба, которая управляет всеми контейнерами Docker, скачивает образы из репозитория, запускает контейнеры, обеспечивает для контейнеров сетевое подключение и постоянное хранилище (тома), собирает с них логи. Она постоянно работает в фоновом режиме на хосте.

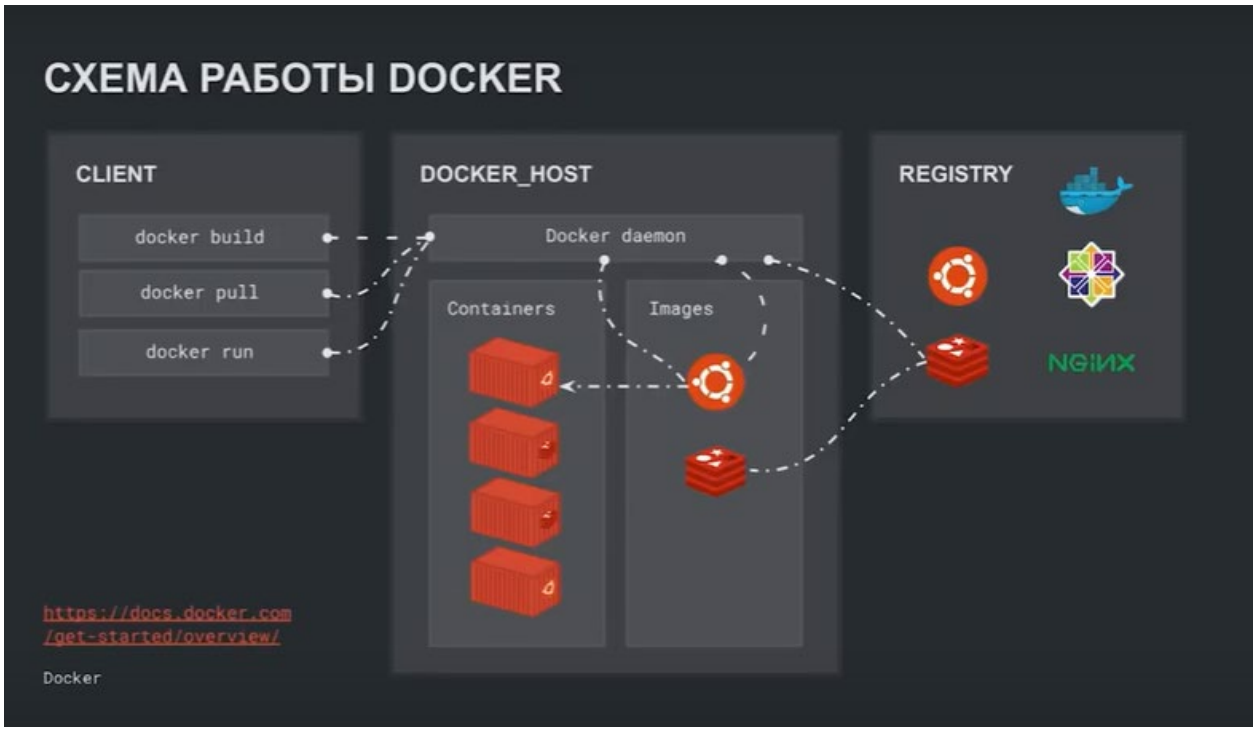
Client (Docker CLI) – это пользовательский интерфейс, с помощью которого возможно писать команды в интерфейсе командной строки и взаимодействовать с API Docker.

Docker Registry - хранилище для образов Docker. Наиболее известный публичный реестр — Docker Hub, также могут использоваться частные (private) реестры.

Image – неизменяемый файл, представляющий в большинстве случаев слепок файловой системы (содержит в себе файлы, директории, программы, с которыми взаимодействует основное приложение). Характерная особенность Image заключается в том, что они строятся слоями.

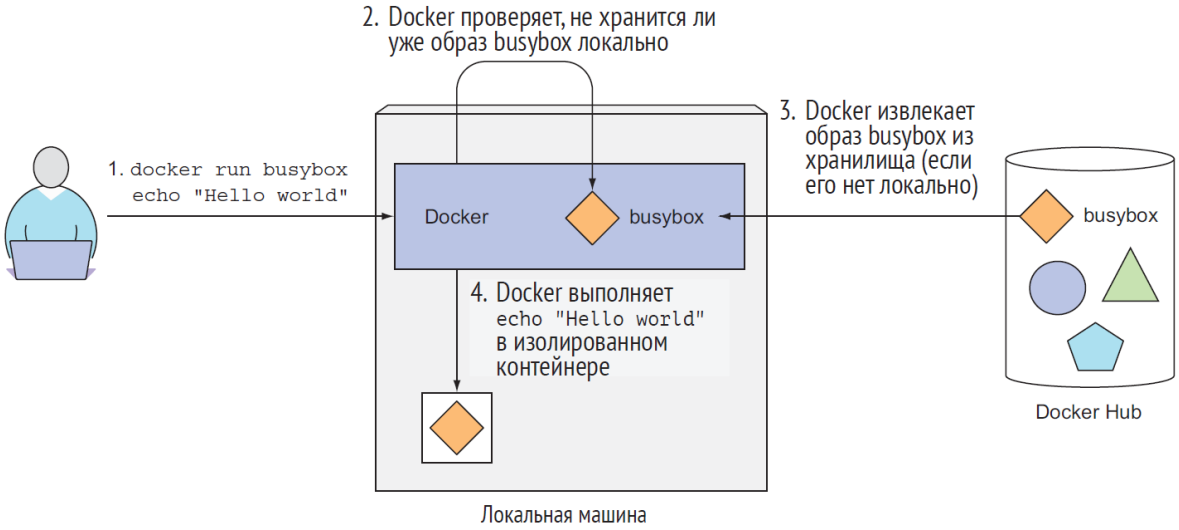
Docker Volumes - Механизм постоянного хранения данных за пределами контейнера.

Docker Networks - Средство организации сетевого взаимодействия между контейнерами.



Взаимодействие компонентов

1. Пользователь вводит команду через **Docker CLI** (например, docker run nginx).
2. Клиент отправляет запрос к **Docker Daemon** через REST API (по сокету /var/run/docker.sock или TCP).
3. Демон проверяет наличие требуемого **образа** локально. Если образ отсутствует, он загружает его из **Docker Registry** (например, Docker Hub).
4. На основе образа создаётся новый **контейнер**, которому выделяются файловая система, сеть и ресурсы согласно настройкам.
5. Контейнер запускается как изолированный процесс на хостовой системе.



2.3. Образы. Строение образов.

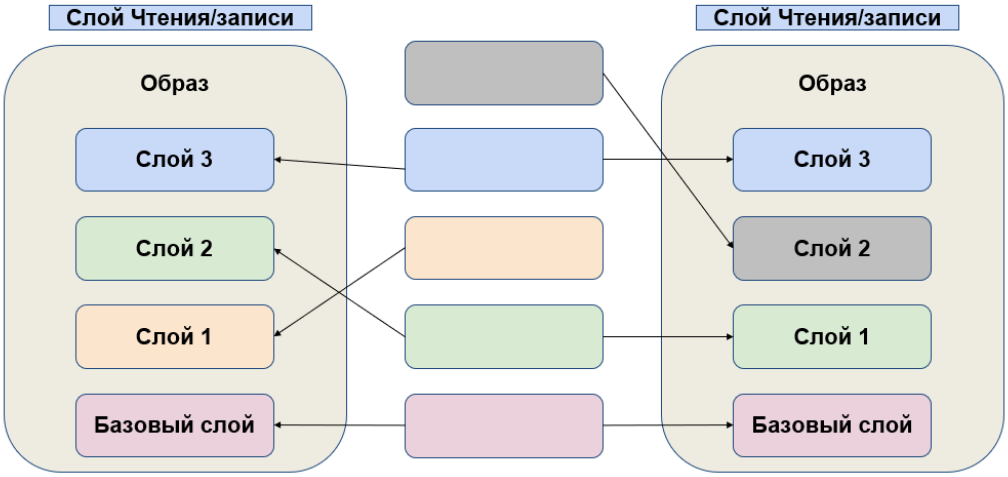
В основе работы Docker лежит понятие образа (image) — это шаблон, из которого создаются контейнеры. Образ можно сравнить с «снимком системы», содержащим всё необходимое для запуска приложения: операционную систему, библиотеки, зависимости и исполняемые файлы. При запуске контейнера Docker использует этот образ как основу для создания изолированной файловой среды.

Образ в Docker — это неизменяемый набор слоёв, каждый из которых добавляет определённые изменения к файловой системе. В отличие от контейнера, который является *запущенным экземпляром* образа, сам образ не выполняется — он служит только для создания новых контейнеров.

Сами образы хранятся в специальном реестре (Docker Registry), например:

- публичные (на Docker Hub)
- частные (локальные или корпоративные репозитории).

Каждый Docker-образ состоит из **нескольких слоёв (layers)**.



Каждый слой — это изменение, добавленное к предыдущему. Например:

- базовая операционная система (Ubuntu, Alpine, Debian и др.);
- установленные пакеты;
- добавленные файлы приложения;
- конфигурационные параметры.

Docker объединяет эти слои в единую файловую систему при помощи технологии, появившейся в Union File System (UnionFS), Docker использует её современную реализацию (OverlayFS, AUFS).
При этом подходе каждый слой доступен только для чтения, а при запуске контейнера поверх этих слоёв создаётся верхний (записываемый) слой, куда помещаются все изменения, происходящие во время работы контейнера.

Одно из ключевых преимуществ Docker заключается в том, что слои образов кэшируются и переиспользуются. Если два образа используют одинаковую базовую часть (например, ubuntu:22.04), Docker хранит этот слой только один раз.
Это позволяет значительно экономить дисковое пространство, ускорять загрузку и сборку образов и упрощать обновление и распространение.

2.4. Репозиторий. Dockerhub

Docker хранит образы локально на хосте (обычно в каталоге /var/lib/docker). Для обмена и распространения используются **реестры (registries)** — специализированные хранилища образов. Самый известный из них — **Docker Hub**, предоставляющий доступ к тысячам публичных образов популярных систем и приложений.

```
docker pull nginx       # загрузить образ из реестра
docker images           # вывести список локальных образов
docker rmi nginx        # удалить локальный образ
docker inspect nginx    # просмотреть подробную информацию об образе
```

Каждый образ имеет:

- **имя** (nginx, python, ubuntu);
- **тег (tag)**, обозначающий версию (nginx:1.25, python:3.12). Если не указывать его явно в команде docker pull , то по умолчанию будет скачиваться образ , имеющий tag

версии - latest. Тег **latest** не означает «самую новую стабильную версию», а лишь указывает на образ, помеченный так по умолчанию, — он может измениться без предупреждения. Поэтому использование latest делает сборку **непредсказуемой и нестабильной**, так как при следующем запуске контейнер может использовать уже другую версию образа.

- **уникальный идентификатор (ID)**, используемый Docker для внутреннего учёта.

Обратите внимание, что при загрузки новой версии образа старая автоматически не удаляется, поэтому необходимо делать это вручную.

2.5. Основные команды Docker

Работа с образами:

Посмотреть список образов

```
docker images
```

Удалить образы

```
docker rmi <образ> [образ…]
docker image rm <образ> [образ…]
```

Удаление неиспользуемых образов

```
docker image prune
```

Удаление всех неиспользуемых образов (все неиспользуемые образы, включая те, которые имеют теги)

```
docker image prune -a
```

Удаление неиспользуемых слоев

```
docker system prune
```

Удаление всех контейнеров, образов, сетевых драйверов и кэша

```
docker system prune -a
```

Работа с контейнерами:

Создание контейнера с именем

```
docker run --name <имя> <образ>
```

Удаление контейнера после завершения его работы

```
docker run --rm <образ>
```

Вывести список всех контейнеров

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

Вывести только ID контейнеров

```
docker ps -q
docker ps --quiet
```

Отслеживание запущенных контейнеров

```
docker ps
```

Запуск и подключение к контейнеру в интерактивном режиме

```
docker run -it <образ>
```

Удаление контейнера

```
docker rm <NAMES or CONTAINER ID>
```

Подключение к работающему контейнеру

```
docker exec -it <NAMES or CONTAINER ID> bash
```

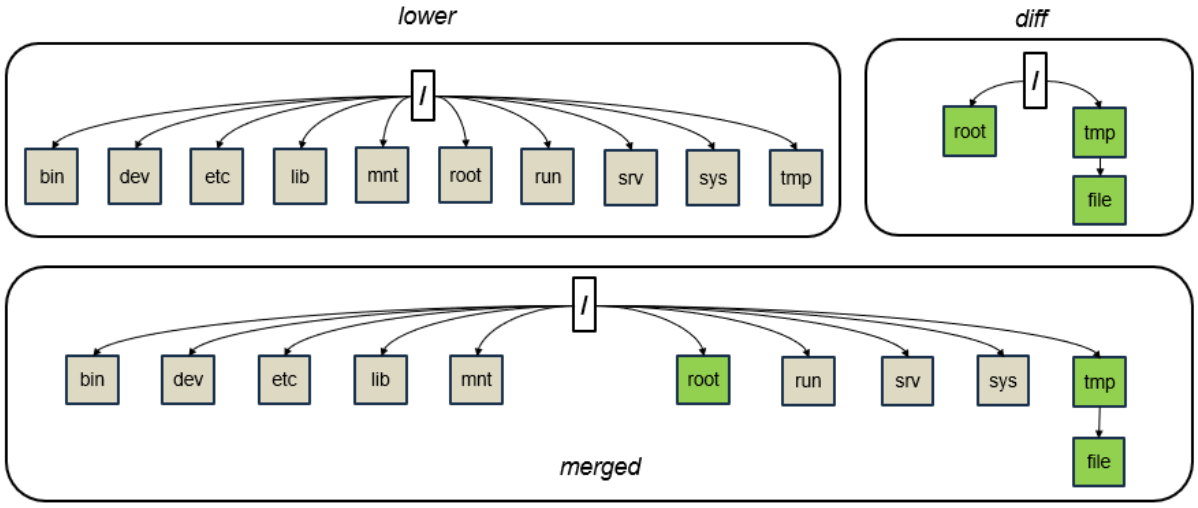
Запуск интерактивной оболочки в контейнере, не имеющем внутри себя Bash, например Alpine.

```
docker exec -it <NAMES or CONTAINER ID> sh
```

2.6. Файловая система контейнера

Каждый контейнер в Docker имеет собственную файловую систему, формируемую на основе слоёв образа. При запуске контейнера Docker добавляет поверх неизменяемых слоёв образа **верхний слой записи (container layer)**, в котором фиксируются все изменения, происходящие во время работы приложения.

Для реализации такой структуры Docker использует **драйвер файловой системы OverlayFS**, основанный на принципе наложения слоёв. На одном хосте, как правило, запускается несколько контейнеров, и у многих из них могут быть одинаковые базовые слои — например, основанные на образах **Alpine**, **Ubuntu** или специализированных сборках вроде **Golang**. Эти нижние слои монтируются в режиме **только для чтения (read-only)** и могут переиспользоваться между контейнерами, что значительно экономит дисковое пространство. Все изменения, происходящие во время работы контейнера, записываются в верхний (записываемый) слой, изолированный от остальных.



На рисунке показан принцип работы файловой системы контейнера с использованием драйвера OverlayFS. Базовый образ (в данном случае, например, Alpine Linux) представлен в виде нижнего слоя (lower), который является только для чтения. При запуске контейнера Docker добавляет поверх него слой записи (diff) — в нём фиксируются все изменения, происходящие в процессе работы контейнера.

Например, если внутри контейнера был создан файл /tmp/file, то он не вносится в базовый образ, а записывается в верхний слой.
Затем при объединении слоёв формируется сводный (merged) слой — это итоговая файловая система, с которой работает приложение внутри контейнера.

Таким образом, файловая система контейнера полностью изолирована от хостовой, что обеспечивает независимость среды выполнения. Однако все изменения внутри контейнера (например, логи, временные файлы или данные баз данных) **исчезают после его удаления**, если они не были вынесены во внешнее хранилище, например, в **том (volume)**.

Docker предоставляет два способа связывания файловой системы контейнера с файловой системой хоста: **bind mount** и **Volumes**. Оба позволяют сохранять данные между перезапусками контейнера, но имеют важные различия.

2.6.1. Bind mount

Bind mount позволяет подключить **любую директорию или файл на хосте** к контейнеру. Такой подход удобен для разработки: изменения на хосте сразу становятся доступными в контейнере.

Пример использования:

```
docker run -it --rm \
  -v /home/user/data:/app/data \
  alpine
```

- /home/user/data — папка на хосте.
- /app/data — путь внутри контейнера.

Однако использование bind mount имеет ряд ограничений. Во-первых, требуется указывать абсолютный путь, что делает контейнеры менее переносимыми между разными операционными системами. Во-вторых, это менее безопасно, так как контейнер получает доступ к любым файлам на хосте, что может привести к ошибкам или случайной порче данных. Кроме того, при большом количестве контейнеров и папок самостоятельно следить за структурой директорий становится неудобно.

2.6.2. Volumes

Volume — это специальная область данных, управляемая Docker, которая хранится **внутри Docker** (по умолчанию в /var/lib/docker/volumes/) и предназначена для сохранения данных контейнера. Они позволяют безопасно сохранять информацию между перезапусками контейнеров и удобно использовать её для обмена между несколькими контейнерами. В отличие от bind mount, volumes не зависят от структуры файловой системы хоста, что повышает переносимость контейнеров и упрощает управление. Кроме того, они считаются более безопасными, поскольку Docker контролирует доступ к данным и не позволяет случайно подключить произвольные файлы. Минусом может быть накопление большого числа томов, которые занимают место на диске, поэтому ненужные volumes следует удалять.

Примеры работы:

```
docker volume create data
docker run –it –-name c1 –v data:/shareddir --rm alpine
```

2.7.Работа с сетевыми интерфейсами

Контейнеры в Docker изолированы не только по файловой системе, но и по сети. Каждый контейнер получает собственный **сетевой стек**, что позволяет запускать несколько контейнеров на одной машине без конфликтов портов и IP-адресов. Docker предоставляет несколько типов сетей, которые можно использовать в зависимости от задач.

Основные типы сетей:

1. **bridge** (по умолчанию)

Используется для автономных контейнеров, которые должны взаимодействовать друг с другом. Контейнеры в одной подсети видят друг друга, но изолированы от других сетей. В примере ниже создается сетевой мост my_net, с помощью которого соединяются контейнеры c1 и c2.

```
docker network create my_net
docker run –it –-name c1 --network=new_net --rm alpine
docker run –it –-name c2 --network=new_net --rm alpine
```

Опция -p позволяет открыть определённый порт на своём компьютере и пробросить его на определённый порт внутри контейнера. В данном примере 8080 – внешний порт компьютера, а 80 – порт контейнера nginx.

Пример:

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

Здесь порт 80 контейнера доступен на порту 8080 хоста

2. **host**

Контейнер теряет сетевую изоляцию и может использовать интерфейсы основной машины. Подходит для высокопроизводительных приложений, но есть риск конфликта портов. Поддерживается только в Linux и Docker 17.06+.

```
docker run --network host -d nginx
```

3. **overlay/overlay2 (Оверлей)**

Соединяет контейнеры на разных хостах и демонах Docker. Используется в Docker Swarm для организации связи между сервисами или между отдельными контейнерами на разных узлах. Упрощает маршрутизацию, но на Windows не поддерживается шифрование.

4. **Macvlan/ipvlan**

Сетевой драйвер, который позволяют назначать MAC-адрес контейнеру, делая его отображаемым как *реальное физическое устройство* в вашей сети. Docker демон направляет трафик на контейнеры по их MAC-адресам. Использование macvlan драйвера иногда является лучшим выбором при работе с устаревшими приложениями, которые ожидают, что они будут напрямую подключены к физической сети. Поддерживается только в Linux.

5. **None**

Сетевой драйвер полностью отключает сетевое взаимодействие контейнера, изолируя его от любых сетей. Как правило, используется совместно с пользовательскими сетевыми драйверами или в тех случаях, когда требуется обеспечить максимальную безопасность и недоступность контейнера по сети.

6. Dockerfile

6.1. Создание собственных образов

Docker предоставляет пользователю возможность не только использовать готовые образы из репозиториев, но и создавать собственные. Это один из ключевых элементов экосистемы Docker, позволяющий автоматизировать развёртывание приложений, настраивать среду под конкретные задачи и делиться готовыми решениями с другими разработчиками.

Собственный образ — это шаблон, содержащий все необходимые компоненты для запуска приложения: операционную систему, зависимости, исполняемые файлы и конфигурации.
Создание образа можно рассматривать как процесс последовательного добавления слоёв — каждый слой представляет собой изменение по сравнению с предыдущим (например, установка пакета, копирование файлов или изменение переменной окружения).

Для сборки собственного образа используется специальный файл Dockerfile — текстовый сценарий, в котором по шагам описывается, как создать образ. Каждая инструкция в Dockerfile добавляет новый слой в итоговый образ. Это позволяет эффективно использовать кэширование: если один из шагов не изменился, Docker повторно использует существующий слой, ускоряя процесс сборки.

6.2. Основные инструкции Dockerfile

1. **FROM** — задаёт базовый (родительский) образ.
2. **LABEL** — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ.
3. **ENV** — устанавливает постоянные переменные среды.
4. **RUN** — выполняет команду и создаёт слой образа. Используется для установки в контейнер пакетов.
5. **COPY** — копирует в контейнер файлы и папки.
6. **ADD** — копирует файлы и папки в контейнер, может распаковывать локальные .tar-файлы и скачивать по URL.
7. **CMD** — описывает команду с аргументами, которую нужно выполнить, когда контейнер будет запущен. Аргументы могут быть переопределены при запуске контейнера. В файле может присутствовать лишь одна инструкция CMD.
8. **WORKDIR** — задаёт рабочую директорию для следующей инструкции.
9. **ARG** — задаёт переменные для передачи Docker во время сборки образа.
10. **ENTRYPOINT** — предоставляет команду для вызова во время выполнения контейнера. Не переопределяется, вариативные аргументы указываются потому отдельно в CMD. Итоговый вид команды, которая запустится при старте контейнера, будет складываться из соответствующих значений: ENTRYPOINT CMD.
11. **EXPOSE** — указывает на необходимость открыть порт (здесь чисто документирование, какой порт нужно не забыть пробросить при запуске контейнера, автоматом ничего не пробросится).
12. **VOLUME** — создаёт точку монтирования для работы с постоянным хранилищем.

Команды для сборки:

Создать образ на основе Dockerfile

```
docker build <путь, где лежит Dockerfile>
```

Создать образ с именем и тегом (ссылка)

```
docker build -t <имя_образа:тег> <путь>
```

Рассмотрим пример создания образа с простым веб-сервером на базе Ubuntu:

```
# Базовый образ
FROM ubuntu:22.04

# Установка зависимостей
RUN apt update && apt install -y nginx

# Копирование собственного index.html
COPY index.html /var/www/html/index.html

# Открываем порт 80
EXPOSE 80

# Указываем команду для запуска nginx
CMD ["nginx", "-g", "daemon off;"]
```

После создания Dockerfile можно собрать образ с помощью команды:

```
docker build -t mynginx .
```

Флаг -t задаёт имя образа, а точка указывает на текущую директорию, где расположен Dockerfile.

После сборки можно убедиться, что образ успешно создан:

```
docker images
```

Для запуска контейнера на его основе используется команда:

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

Теперь веб-сервер будет доступен по адресу http://localhost:8080.

6.3.Оптимизация образа

Одной из ключевых задач при работе с Docker является оптимизация создаваемых образов. От того, насколько грамотно построен Dockerfile, зависит размер образа, скорость его сборки, время загрузки в репозиторий и запуска контейнера. Кроме того, оптимизированные образы экономят место на диске и позволяют ускорить процесс доставки приложений в продакшн.

Оптимизация образов базируется на трёх основных принципах:

1. *Минимизация количества слоёв*

Каждый слой в Docker создаётся при выполнении инструкций RUN, COPY и ADD. Чем больше слоёв — тем больше размер итогового образа. Чтобы сократить их число, можно объединять команды в одной инструкции:

```
RUN apt update && \
    apt install -y nginx curl && \
    rm -rf /var/lib/apt/lists/*
```

Здесь три действия (обновление, установка и очистка) выполняются в одном слое.

2. *Использование лёгких базовых образов*

Вместо громоздких дистрибутивов, таких как ubuntu или debian, лучше использовать образы, основанные на Alpine Linux:

```
FROM alpine:3.19
```

Alpine весит около 5 МБ, что в десятки раз меньше стандартного Ubuntu. Однако стоит учитывать, что в Alpine используется библиотека musl вместо glibc, поэтому некоторые бинарные файлы могут потребовать пересборки.

3. *Удаление временных файлов и кэшей.*

После установки пакетов рекомендуется очищать системные кэши, временные директории и пакеты, чтобы не сохранять ненужные данные в слое образа.

```
RUN apt update && apt install -y build-essential && \
    make && make install && \
    apt remove -y build-essential && \
    rm -rf /var/lib/apt/lists/* /tmp/*
```

6.4. Многоступенчатая сборка (multi-stage build)

Одним из наиболее эффективных способов оптимизации Docker-образов является многоступенчатая сборка (*multi-stage build*). Этот механизм позволяет разделять процесс сборки приложения и создание финального образа, тем самым значительно сокращая его размер и повышая безопасность.

При сборке сложных приложений часто требуется установить дополнительные инструменты: компиляторы, зависимости, пакеты и библиотеки. Если выполнять сборку и запуск в одном образе, то все эти временные файлы попадают в итоговый результат, занимая место и потенциально увеличивая поверхность атаки.

Например, для сборки программы на Go или C++ нужно установить компилятор, библиотеки, утилиты — но при запуске готового бинарного файла они больше не нужны. Тем не менее, если не разделить сборку и запуск, все они останутся внутри контейнера.

Multi-stage build решает эту проблему. В таком подходе создаются **несколько этапов сборки** внутри одного Dockerfile. На первом этапе производится сборка приложения, а на втором (и последующих) — создаётся «чистый» финальный образ, куда копируется только результат сборки. В каждом этапе можно использовать собственный базовый образ, оптимизированный под конкретную задачу. Например, в первом — полноценный golang или ubuntu с инструментами, во втором — минимальный alpine или scratch.

```
# Этап 1 – сборка приложения
FROM golang:1.22 AS builder
WORKDIR /app
```

```
COPY . .
RUN go build -o myapp

# Этап 2 — финальный образ
FROM alpine:3.19
WORKDIR /app
COPY --from=builder /app/myapp .
CMD ["./myapp"]
```

Здесь:

- в первом этапе (builder) используется образ golang:1.22, где выполняется сборка программы;
- на втором этапе создаётся минимальный образ alpine, в который копируется только готовый бинарный файл myapp.

В результате итоговый контейнер весит всего несколько мегабайт, в отличие от десятков или сотен мегабайт, которые занимал бы полный образ с компилятором.

7. Docker compose

В процессе разработки и эксплуатации часто возникает необходимость запускать несколько контейнеров, которые должны взаимодействовать друг с другом. Например, веб-приложение, база данных и система кеширования могут быть оформлены как отдельные контейнеры, но для нормальной работы им нужно быть объединёнными в одну сеть и запускаться как единое приложение. Для этого используется Docker Compose — инструмент, который позволяет описывать и управлять многоконтейнерными приложениями с помощью простого YAML-файла.

Docker Compose по умолчанию использует конфигурационный файл docker-compose.yml, где задаются:

- используемые образы (готовые или создаваемые из Dockerfile);
- параметры запуска контейнеров (порты, переменные окружения, тома);
- зависимости между сервисами (порядок запуска);
- сетевые настройки.

Compose автоматически создаёт сеть, монтирует тома, запускает и связывает контейнеры между собой.
Это делает процесс развёртывания повторяемым и удобным — достаточно одной команды.

Основные функции Docker Compose:

- *Определение сервисов*

В файле docker-compose.yml можно определить несколько сервисов (контейнеров), которые составляют ваше приложение. Каждый сервис может иметь свои собственные настройки, такие как образ, порты, переменные окружения и зависимости.

- *Управление зависимостями*

Docker Compose позволяет указать, какие сервисы зависят от других. Это упрощает настройку и запуск приложений, состоящих из нескольких компонентов.

- *Упрощение команд*

Вместо того чтобы вручную запускать и управлять каждым контейнером, вы можете использовать одну команду (docker-compose up), чтобы запустить все сервисы, определенные в вашем файле.

- *Сетевые настройки*

Docker Compose автоматически создает сеть для ваших контейнеров, что позволяет им легко взаимодействовать друг с другом.

- *Масштабирование*

Вы можете легко масштабировать сервисы, увеличивая или уменьшая количество экземпляров контейнеров с помощью одной команды.

Пример конфигурации:

Простейший пример docker-compose.yml, в котором описаны два сервиса — **web** и **db**:

```
version: "3.9"

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    depends_on:
      - db

  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Здесь:

- сервис **web** использует готовый образ nginx и монтирует локальную папку html;
- сервис **db** поднимает контейнер с PostgreSQL, где данные сохраняются в том db_data;
- опция depends_on гарантирует, что база данных запустится раньше, чем веб-сервер;

- в конце файла объявлены именованные тома, управляемые Docker.

Основные команды:

```
docker compose ps – список контейнеров
docker compose up – поднять приложение
docker compose stop – остановить поднятые контейнеры
docker compose start – запустить остановленные контейнеры
docker compose down – остановить и удалить контейнеры
docker compose logs – вывести логи всех сервисов.
```

Обучающая часть

Задание 1. Установка docker

Установите docker на вашу машину. После установки Docker рекомендуется предоставить администратору право работать с контейнерами не используя sudo.

```
sudo apt install docker.io
sudo usermod -aG docker $USER
```

Установите docker compose

```
sudo apt install docker-compose-v2
```

Убедитесь, что программа установлена верно используя

```
docker info
```

Задание 2. Запуск контейнера

2.1. Поднимаем контейнер

```
docker run -it --name test-container ubuntu
```

- -it — интерактивный режим с терминалом
- --name test-container — даём контейнеру понятное имя

Внутри контейнера вы окажетесь в shell Ubuntu.

2.2. Исследуем контейнер

Проверяем текущий пользователь и директорию:

```
whoami
pwd
```

Смотрим процессы внутри контейнера:

```
ps aux
```

На хосте проверяем процессы Docker:

```
docker ps -a
ps aux | grep docker
```

Сравниваем PID’ы процессов внутри контейнера и на хосте

2.3. Установка ПО и Проверка портов

Установите внутри контейнера python

```
apt update
apt install python3
```

В контейнере попробуйте запустить сервер:

```
python3 -m http.server 8080
```

На хосте проверяем открытые порты:

```
netstat -tuln | grep 8080
```

Виден ли открытый порт на основной машине? Запустите сервер на ней и перезапустите команду netstat. Виден ли в этом случае?

2.4. Работа с файловой системой

Создаём директорию и файл внутри контейнера:

```
mkdir -p /home/student/lab1
echo "Hello Docker!" > /home/student/lab1/readme.txt
cat /home/student/lab1/readme.txt
```

Выходим из контейнера:

```
exit
```

Поднимаем контейнер снова:

```
docker start -ai test-container
```

Проверяем, есть ли файл:

```
ls /home/student/lab1
```

Очистите все остановленные контейнеры командой

```
docker container prune
```

Перезапустите контейнер и проверьте наличие файла.

Задание 3. Проверка ограничений контейнера

Запустите контейнер, содержащий командную оболочку, и подключитесь к нему

```
docker run -it bash
```

Внутри контейнера запустите скрипт, который нагружает систему бесконечной записью символа y в файл:

```
yes > file
```

В соседнем терминале запустите программу, осуществляющую мониторинг ресурсов:

```
docker stats
```

Обратите внимание на поля CPU и MEM USAGE. Какие значения они принимают? Прекратите выполнение скрипта и проверьте нагрузку. Остановите контейнер.

Запустите контейнер с параметрами, ограничивающими выделение ресурсов на контейнер

```
docker run -it --memory=512m --cpus=0.5 bash
```

Проверьте текущие значения мониторинга. Изменилось ли что-то в выводе команды docker stats?

Аналогично запустите нагружающий скрипт. Какое значение принял столбец с CPU?

Задание 4. Исследование сетевых драйверов

Исследование сетевого драйвера bridge

Создайте собственную сеть:

```
docker network create my-net
```

Запустить два контейнера в этой сети:

```
docker run -it --rm --name c1 --network my-net bash
docker run -it --rm --name c2 --network my-net bash
```

Проверить изнутри доступность контейнеров по имени и ip адресу:

```
ping c2
```

Контейнеры должны «видеть» друг друга по имени, так как Docker предоставляет внутренний DNS.

Исследование сетевого драйвера host

Запустите контейнер с общим сетевым пространством:

```
docker run -it --rm --network host bash
```

Проверьте сетевые интерфейсы:

```
ip a
```

Какие интерфейсы вы видите изнутри контейнера?

Исследование сетевого драйвера none

Запустить контейнер:

```
docker run -it --rm --network none bash
```

Проверьте сетевые интерфейсы:

```
ip a
```

Какие интерфейсы вы видите изнутри контейнера?

Проброс портов

Запускаем контейнер с Nginx:

```
docker run --rm --name web -p 8080:80 nginx
```

Открываем в браузере страницу:

```
http://localhost:8080
```

Задание 5. Исследование файловой системы

Проверка работы bind mount

Создаём папку на хосте:

```
mkdir bindtest
```

```
echo "Hello from host" > bindtest/test.txt
```

Запускаем контейнер и монтируем папку:

```
docker run -it --rm -v ./bindtest:/data alpine
```

Проверяем содержимое:

```
cat /data/test.txt
```

Создаём новый файл внутри контейнера:

```
echo "Created inside container" > /data/new.txt
```

После выхода из контейнера проверяем, что файл появился на хосте:

```
cat ./bindtest/new.txt
```

Проверка работы volume

Создаём именованный том:

```
docker volume create mydata
```

Запускаем контейнер, монтируя том:

```
docker run -it --rm -v mydata:/store alpine sh
```

Создаём файл:

```
echo "Persistent data" > /store/data.txt
exit
```

Запускаем другой контейнер и проверяем, сохранились ли данные:

```
docker run -it --rm -v mydata:/store alpine sh
cat /store/data.txt
```

Изучение файловой системы контейнера

Создаём контейнер на основе alpine, с интерактивным доступом:

```
docker run -it --name fs-test alpine sh
```

Внутри контейнера:

```
echo "Hello from container" > /home/somefile.txt
cat /home/somefile.txt
```

Выходим из контейнера, но не удаляем его

```
exit
```

Все слои и изменения контейнеров в Docker хранятся по пути: */var/lib/docker/overlay2/*

Проверим, к какому слою привязан наш контейнер:

```
docker inspect fs-test | grep UpperDir
```

Результат будет примерно такой:

```
"UpperDir":
"/var/lib/docker/overlay2/52de2b83d3f0e7a1f75ad5f9d67c91b1b7c8c56d8a8b51b
2f9f3e9e3a6f1kh7d/diff"
```

Теперь можно увидеть созданный файл прямо в файловой системе хоста:

```
sudo ls
/var/lib/docker/overlay2/52de2b83d3f0e7a1f75ad5f9d67c91b1b7c8c56d8a8b51b2
f9f3e9e3a6f1kh7d/diff/home
```

Посмотрим его содержимое:

```
sudo cat
/var/lib/docker/overlay2/52de2b83d3f0e7a1f75ad5f9d67c91b1b7c8c56d8a8b51b2
f9f3e9e3a6f1kh7d/diff/home/somefile.txt
```

Изучите содержимое каталогов diff, merged, lower. Перейдите в каталог diff. Какие директории/файлы вы там видите?

Изнутри контейнера зайдите в директорию bin и удалите ссылку на ping

```
rm /bin/ping
```

Что изменилось в директории diff? Какие атрибуты у появившегося файла?

Задание 6. Создание собственного контейнера

Базовый образ и рабочая директория

Создадим Dockerfile. В качестве базового образа выберем легковесную ОС – Alpine. Также зададим рабочую директорию при помощи инструкции WORKDIR. Это будет директория /home/student.

Dockerfile

```
FROM alpine:latest
WORKDIR /home/student
```

Соберем образ, запустим его и подключимся:

```
docker build -t myalpine:v1 .
docker run -it myalpine:v1 sh
```

Проверьте, в какой директории вы находитесь.

Копируем файл в контейнер

Добавим в контейнер какой-нибудь файл. Для этого создадим файл script.sh рядом с Dockerfile:

```
echo 'echo "Hello from inside the container!"' > script.sh
chmod +x script.sh
```

Теперь добавим копирование в Dockerfile:

```
FROM alpine:latest
WORKDIR /home/student
COPY script.sh ./script.sh
```

Собираем второй образ:

```
docker build -t myalpine:v2 .
```

Запускаем контейнер и проверяем, что файл действительно скопировался:

```
docker run -it myalpine:v2 sh
ls
./script.sh
```

Устанавливаем нужные программы

Добавим установку некоторых утилит, например curl, git и vim. Для этого используем инструкцию RUN, которая позволяет выполнить произвольную команду при сборке образа.

```
FROM alpine:latest
WORKDIR /home/student
COPY script.sh ./script.sh
RUN apk update && apk add --no-cache curl git vim
```

Собираем третий образ:

```
docker build -t myalpine:v3 .
```

После сборки можно проверить установленные пакеты:

```
docker run -it myalpine:v3 sh
git --version
vim --version
curl --version
```

Оптимизация слоёв

Измените скрипт и пересоберите образ. Какие действия закэшировались, а какие выполнились заново? Оптимизируйте скрипт, переставив порядок команд:

Исправленный вариант:

```
FROM alpine:latest
RUN apk update && apk add --no-cache curl git vim
WORKDIR /home/student
COPY script.sh ./script.sh
```

Соберите образ, запустите его, после измените скрипт и снова пересоберите. Какие слои закэшировались в этом случае?

Задание 7. Multi-staged сборка

Создайте директорию с вашим проектом.

```
project/
├── app/
│   └── main.c
├── Dockerfile.single
└── Dockerfile.multi
```

Создайте файлы со следующим содержимым:

main.c

```
#include <stdio.h>
int main() {
    printf("Hello, MPSU!\n");
    return 0;
}
```

Dockerfile.single

```
FROM gcc:13.2
WORKDIR /app
COPY app/main.c .
RUN gcc main.c -o main
CMD ["./main"]
```

Dockerfile.multi

```
FROM gcc:13.2 AS builder
WORKDIR /app
COPY app/main.c .
RUN gcc main.c -o main

FROM debian:bookworm-slim
WORKDIR /app
COPY --from=builder /app/main .
CMD ["./main"]
```

Соберите и запустите оба контейнера.

```
docker build -t c-single -f Dockerfile.single .
docker build -t c-multi -f Dockerfile.multi .
docker run --rm c-single
docker run --rm c-multi
```

Отличается ли вывод каждого из контейнеров? Используя docker images посмотрите размер контейнеров. Отличается ли он? Во сколько раз? Объясните почему.

Задание 8. Сравнение COPY/ADD

Копирование архивов

Создайте проект со следующей структурой:

```
project/
├── files.tar.gz
├── Dockerfile.copy
└── Dockerfile.add
```

Dockerfile.copy

```
FROM alpine:3.19
WORKDIR /app
COPY files.tar.gz .
CMD ["sh"]
```

Dockerfile.add

```
FROM alpine:3.19
WORKDIR /app
ADD files.tar.gz .
CMD ["sh"]
```

Для создания архива воспользуемся командой, приведенной ниже и заархивируем сами Dockerfile.

```
tar -czf files.tar.gz Dockerfile.copy Dockerfile.add
```

Соберем образы, запустим и подключимся к ним:

```
docker build -t add-test -f Dockerfile.add .
docker build -t copy-test -f Dockerfile.copy .
docker run --rm -it add-test
docker run --rm -it copy-test
```

Сравните содержимое директорий /app.

Загрузка по ссылке

Создайте Dockerfile со следующим содержимым:

```
FROM alpine:3.19
WORKDIR /app
ADD https://miet.ru miet.html
CMD ["sh"]
```

Соберите указанный образ. Подключитесь к нему и проверьте содержимое.

```
docker build -t copy-uri -f Dockerfile .
docker run --rm -it copy-uri
```

Создайте Dockerfile со следующим содержимым:

```
FROM alpine:3.19
WORKDIR /app
COPY https://miet.ru miet.html
CMD ["sh"]
```

Соберите указанный образ. Удалось ли вам это сделать? Почему?

Задание 9. Сравнение ENTRYPOINT/CMD

Dockerfile

```
FROM alpine:latest
CMD ["echo", "Hello from DOCKER"]
```

Соберите образ:

```
docker build -t cmd-example .
```

Запустите контейнер дважды – с дополнительным параметром и без:

```
docker run cmd-example
docker run cmd-example echo "Hi, custom message"
```

Сравните вывод команд.

Замените CMD на ENTRYPOINT. Соберите и запустите контейнер с аргументами и без. Сравните вывод со значениями, полученными с CMD.

Задание 10. Взаимодействие двух контейнеров

Создайте два файла – клиент и сервер на языке python.

server.py

```
import socket

HOST = "0.0.0.0"
PORT = 5000

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print(f"Server started on {HOST}:{PORT}")
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        data = conn.recv(1024)
        if data:
            print("Received:", data.decode())
            conn.sendall(b"Hello from server!")
```

client.py

```
import socket

HOST = "127.0.0.1"
PORT = 5000

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
   s.connect((HOST, PORT))
   s.sendall(b"Hello from client!")
   data = s.recv(1024)
   print("Received:", data.decode())
```

Убедитесь в их работоспособности.

```
Терминал 1 -> python3 server.py
Терминал 2 -> python3 client.py
```

Создадим для приложений два контейнера:

Dockerfile.server

```
FROM python:3.12-alpine
WORKDIR /app
COPY server.py .
CMD ["python", "server.py"]
```

Dockerfile.client

```
FROM python:3.12-alpine
WORKDIR /app
COPY client.py .
CMD ["python", "client.py"]
```

Собираем и запускаем контейнеры вручную

Сервер:

```
docker build -t python-server -f Dockerfile.server .
docker build -t python-client -f Dockerfile.client .
Терминал 1 -> docker run -d --name server -p 5000:5000 python-server
Терминал 2 -> docker run --rm --name client --network host python-client
```

Для упрощения сборки и запуска создадим Docker Compose

docker-compose.yml

```
version: "3.9"

services:
  server:
    build:
      context: .
      dockerfile: Dockerfile.server
    container_name: server
    ports:
      - "5000:5000"

  client:
    build:
      context: .
      dockerfile: Dockerfile.client
    container_name: client
    depends_on:
      - server
```

Внутри файла клиента исправим значение HOST на "server", чтобы обращение производилось по DNS имени.

Запустите сборку и проверьте работоспособность:

```
docker compose up --build
```

Исследовательская часть

Создание собственного контейнера на основе namespaces и cgroups

Шаг 1: Создание изолированного окружения

Используем команду unshare, чтобы вручную создать контейнероподобное окружение. Она запускает новый процесс, который не разделяет системные пространства имён с хостом.

```
sudo unshare --fork --pid --mount --uts --ipc --net --user --map-root-
user --mount-proc=/proc bash
```

Что здесь происходит:

- --pid — создаёт новый PID namespace (изолированные процессы);
- --uts — изолирует имя хоста;
- --ipc — изолирует системные очереди сообщений и сегменты памяти;
- --net — изолирует сетевой стек;
- --mount — создаёт новое mount-пространство;
- --user — создаёт новый user namespace (root внутри контейнера не равен root на хосте);
- --map-root-user — отображает UID 0 (root внутри) в UID текущего пользователя на хосте;
- --mount-proc=/proc — монтирует отдельный /proc внутри.
  Теперь внутри полученной “мини-системы” можно выполнить:

```
hostname test
```

и убедиться, что имя системы изменилось

Шаг 2: Монтирование файловых систем

Создадим набор директорий из Alpine Linux, к которым будем монтироваться изнутри контейнера

```
mkdir rootfs_alpine
cd rootfs_alpine
```

Скачиваем минимальный образ Alpine Linux (x86_64) и распаковываем в каталог rootfs

```
wget https://dl-cdn.alpinelinux.org/alpine/latest-
stable/releases/x86_64/alpine-netboot-3.22.1-x86_64.tar.gz
sudo tar -xzf alpine-netboot-3.22.1-x86_64.tar.gz -C .
```

Перезапускаем окружение и внутри контейнера монтируем каталоги:

```
mount -t proc proc rootfs_alpine/proc
mount -t sysfs sysfs rootfs_alpine/sys
```

Теперь запускаем оболочку с каталогами:

```
exec chroot rootfs_alpine /bin/sh
```

Проверяем:

```
hostname
```

```
ps aux
```

Ты уже внутри “контейнера” — процессы изолированы, файловая система ограничена, но пока нет сети и ограничений по ресурсам.

Шаг 3. Настройка сети через veth-пару

Теперь создаём связку veth-интерфейсов, чтобы контейнер имел сетевое соединение с хостом.

На **хосте**:

```
sudo ip link delete veth1 2>/dev/null || true
sudo ip link add veth2 netns $(ps aux | grep unshare | grep -v 'grep' |
awk 'NR>2{print $2}') type veth peer name veth1 netns 1
sudo ip addr add 172.12.0.1/24 dev veth1
sudo ip link set dev veth1 up
```

В **контейнере**:

```
ip addr add 172.12.0.2/24 dev veth2
ip link set dev veth2 up
ip link set dev lo up
```

Проверим соединение:

```
ping -c 2 172.12.0.1
```

Если пингуется — сеть между контейнером и хостом работает.

Шаг 4. Добавление ограничений по CPU (cgroups)

Изнутри контейнера запустим скрипт

```
while true; do :; done
```

На основной машине запустите top и проверьте, какой процент от CPU используется?

Создадим отдельную группу cgroups и ограничим процессорное время:

```
sudo mkdir /sys/fs/cgroup/mycontainer
echo "20000 100000" | sudo tee /sys/fs/cgroup/mycontainer/cpu.max
```

(означает: максимум 20% CPU, 20 000 мкс работы из 100 000 мкс периода)

Найдём PID процесса контейнера:

```
ps aux | grep unshare | grep -v 'grep' | awk 'NR>2{print $2}'
```

И добавим его в группу:

```
ps aux | grep '/bin/sh' | grep -v 'grep' | awk '{print $2}' | sudo tee
/sys/fs/cgroup/mycontainer/cgroup.procs
```

Теперь можно проверить ограничения, запустив CPU-нагрузку.

Шаг 5. Проверка ограничения CPU

Внутри контейнера:

```
while true; do :; done
```

Процесс теперь “вращается” в цикле, но за счёт настроек cgroup он будет использовать не более 20% CPU.

На хосте можно проверить:

```
top
```

Процесс контейнера не превышает ограничение.

Практико-ориентированная часть

Задание 1

Разверните контейнер, содержащий ваше приложение, собранное в предыдущих лабораторных работах. Для сборки образов используйте DockerFile. Проверьте работоспособность системы. Отправьте ваши изменения на git.

Задание 2.

Добавьте в CI/CD отправку вашего образа в репозиторий DockerHub. Выгрузите ваше приложение локально и убедитесь в его работоспособности.

Задание 3.

Используя docker-compose, объедините ваши контейнеры вместе с системой мониторинга из предыдущей лабораторной работы.