# Program environment

1. Любая программа всегда запускается в некотором программном окружении.

Окружением может быть некоторая ОС с установленными библиотеками или минимально нужные пакеты для работы программы

1. Культура DevOps предлагает запускать программу только в строго контролируемомом и воспроизводимом окружении.  
     
   Это позволяет гарантировать работоспособность программы и сторого ожидаемое поведение.
2. DevOps (человек) занимается настраиванием всех пайплайнов и определения стека, который будут использовать.

Однако, в практике программной разработки для тестирования программ предполагается, что созданием окружения занимается сам разработчик (или хотя бы составлением списка необходимых программ).

## Возможности для создания программного окружения

1. Для создания программного окружения используются возможности операционных систем.
2. Так ядро линукс предоставляет широкий спектр для создания изолированного и контролируемого окружения.

Стандартные инструменты для создания контролируемого программного окружения:

* Контейнеризация
* Виртуализация

Более полный список возможностей ядра Линукс представлен в Возможности ядра линукс и девопс в Материалы.

# Контейнеризация

1. Контейнер – это запущенный процесс, полностью изолированный от основной ОС

По сути контейнер это как минимальная ОС, полностью независимая от основной и в которой можно устанавливать свои пакеты и в этой среде запускать свои программы.

Контейнер - запущенная мини операционная система изолированная от нашей ОС и содержащую некоторую программу

1. Контейнеры можно легко создавать и удалять.

## Докер

1. Программа которая позволяет работать с механизмом контейнеризации (Фича линукс)
2. Докер создаёт изолированный процесс (контейнер) и проводит базовые операции с ним в соответствии со скриптом, написанным разработчиком – dockerfile.

Базовые операции такие как: скачинание и установка ОС, загрузка библиотки, перемещение файлов из основной ОС  
Разработчику важно самостоятельно указать требуемую ОС и программные пакеты нужные для работы его программы, а также перенос файлов в нужные папки. Именно так он создаёт контролируемое окружение.

1. После запуска такого процесса со скриптом – шаблон такого процесса можно сохранить для быстрого запуска.

Такой шаблон называется image

### Создание контейнера

#### Dockerfile

1. Dockerfile – скрипт, по инструкции которого нужно создать изолированный процесс
2. Этот докерфайл скрипт имеет определенную структуру – операции выполняются построчно сверху вниз, поэтому самые необходимые операции (вроде установки ОС) необходимо записывать в самом начале скрипта.
3. После создания изолированного процесса, можно заморожить исходное состояние этого процесса, чтобы в дальнейшем быстро с нуля снова запустить этот процесс – такой шаблон называется **image**

##### Общая структура и важные аспекты

1. Простая структура докер файла выглядит следующим образом:

# 1. Базовая ОС или базовый образ (ОБЯЗАТЕЛЬНО первый)

FROM python:3.9-slim

# 2. Метаданные (опционально)

LABEL maintainer="your.email@example.com"

LABEL version="1.0"

LABEL description="My Python application"

# 3. Переменные окружения

ENV PYTHONUNBUFFERED=1

ENV APP\_HOME=/app

# 4. Рабочая директория

WORKDIR $APP\_HOME

# 5. Копирование файлов зависимостей (для кэширования)

COPY requirements.txt .

# 6. Установка зависимостей

RUN pip install --no-cache-dir -r requirements.txt

# 7. Копирование остального кода приложения

COPY . .

# 8. Открытие портов (информационно)

EXPOSE 8000

# 9. Запуск приложения

CMD ["python", "app.py"]

Дополнительные операции могут добавляться или удаляться в заивисимости от нужд.

1. Список аспектов которые важно учитывать при создании образа:

Базовый образ: Выбран минимальный и безопасный (alpine, distroless).

Пользователь: Создан и используется непривилегированный пользователь (USER).

Кэширование: COPY и RUN инструкции оптимизированы для кэша.

Мультистадийная сборка: Используется для отделения артефактов сборки от рантайма.

Уязвимости: Образ просканирован на наличие CVE.

.dockerignore: Файл создан и заполнен.

Переменные окружения (ENV): Используются для конфигурации.

Томы (VOLUME): Объявлены для персистентных данных.

Порты (EXPOSE): Указаны для документации.

Команда запуска: Правильно выбран CMD или комбинация ENTRYPOINT + CMD.

Health Check (HEALTHCHECK): Добавлена проверка работоспособности.

Метки (LABEL): Добавлена метаинформация.

Сеть: Понятно, какие порты используются (настройка сети — этап runtime).

##### Базовая ОС контейнера

1. Разные программы требуют различного программного окружения.

* Большинство программ требуют наличие минималистичных операционных систем.
* Однако, существуют и программы для которых нужно настраивать окружение с чистого листа

###### Исходный образ с ОС

1. Обычно ОС устанавливаются не вручную в изолированный процесс, а копируется существующий образ (image) с уже установленной минималистичной ОС
2. Докер позволяет установить в изолированный поток огромный спектр базовых образов с минималистичными ОС, специализированными для конкретных языков или фреймоврков, вроде Golang linux, или python linux
3. Базовый образ с ОС – это самое базовое программное окружение, поэтому его требуется указать в самом начале dockerfile
4. Так если мне нужно запустить программу на python, то мне нужна минималистичная ОС, с установленным python и его интерпретатором:

FROM python:3.11-slim

1. Хранилище базовых образов с различными ОС и для различных целей

* DockerHub

###### FROM SCRATCH

##### Рабочая директория

1. Создаёт папку в руте и все остальные операции и команды будут выполняться из этой директории
2. Что происходит без WORKDIR:

Файлы копируются в корень (/)

Зависимости устанавливаются в корень

Приложение пытается запуститься из корня

Путаница с путями

Риск случайной перезаписи системных файлов

С WORKDIR:

FROM node:18

WORKDIR /app

COPY package.json .

RUN npm install # зависимости установятся в /app/node\_modules

Без WORKDIR:

FROM node:18

COPY package.json .

RUN npm install # зависимости установятся в /node\_modules (в корень!)

##### RUN / CMD

1. RUN выполняется во время сборки образа (docker build).
2. CMD Указание команды по умолчанию, которая будет выполняться при старте контейнера.

##### Сеть контейнера

1. Информация о том как работает сеть для контейнера представлена в
2. Docker networking overview ([docs.docker.com](http://docs.docker.com))

Networking with standalone containers

Книга: Docker Deep Dive (Nigel Poulton) — раздел о сети.

RFC 791 (IP) и RFC 793 (TCP) — базовая теория IP и портов.

#### Образ

##### Создание образа по dockerfile

1. <https://docs.docker.com/get-started/docker-concepts/building-images/>

#### Контейнер

##### Запуск контейнера

1. <https://docs.docker.com/reference/>

##### Подключение к контейнеру

#### Volumes (тома)

#### Контейнеры и порты

1. **Основы IP и портов**:

* IP-адрес (например, 192.168.1.100) идентифицирует устройство в сети, а порт (например, 80, 9090) указывает конкретное приложение или сервис на этом устройстве.
* Адрес в сети записывается как IP:порт (например, 192.168.1.100:80), где порт определяет, к какому процессу направлен запрос.
* Хост (компьютер или сервер) слушает определённые порты (например, 9090) для приёма внешних запросов от устройств.

1. **Роль Docker и контейнеров**:

* Docker-контейнеры — это изолированные среды, которые запускаются на хосте и имеют собственное сетевое, файловое и процессное пространство.
* Каждый контейнер получает уникальный внутренний IP-адрес в сети Docker (например, 172.17.0.2), который изолирован от IP хоста и других контейнеров.

1. **Изоляция контейнеров от хоста**:

* Контейнеры изолированы от хоста с помощью Linux namespaces и cgroups, что означает, что они не имеют прямого доступа к сетевым интерфейсам, файлам или процессам хоста, если это не настроено explicitly.
* Сетевой localhost (127.0.0.1) внутри контейнера относится только к самому контейнеру, а не к хосту. Например, curl localhost:8080 внутри контейнера обращается к сервису внутри того же контейнера, а не к хосту.
* Файловая система, процессы и ресурсы контейнера отделены от хоста, если только не используются общие тома (-v) или другие механизмы.

1. **Как контейнеры общаются с хостом и внешним миром**:

* **Портовый маппинг (-p)**: Хост может "публиковать" порты контейнера, связывая их с своими портами. Например, -p 9090:8080 перенаправляет запросы с порта 9090 хоста на порт 8080 внутри контейнера через NAT (Network Address Translation).
* **Мостовые сети (Bridge Networks)**: По умолчанию Docker использует мост (bridge network), который создаёт виртуальную сеть для контейнеров. Хост и контейнеры общаются через этот мост, но без прямого доступа. Мост позволяет контейнерам иметь собственные IP-адреса и порты, а хост может перенаправлять трафик через него.
* **Host Network Mode**: Если использовать --network host, контейнер "обходит" изоляцию и использует сетевые интерфейсы хоста напрямую, что позволяет ему видеть localhost хоста и все его порты, но снижает безопасность.

1. **Как контейнер может получать информацию от хоста**:

* **Через мосты (Bridge)**: В стандартной конфигурации (bridge network) контейнеры общаются с хостом через маппинг портов. Например, запрос от хоста на ХХХ.ХХ.ХХ:9090 перенаправляется в контейнер на его порт 8080.
* **Статический маршрут или DNS**: Контейнер может быть настроен так, чтобы знать IP хоста (например, 192.168.1.100) и обращаться к нему напрямую, если хост слушает определённые порты.
* **Общие ресурсы**: Хост может делиться данными с контейнером через монтируемые тома (-v /host/path:/container/path) или переменные окружения.
* **API или запросы**: Контейнер может отправлять HTTP-запросы или другие типы соединений на хост, если знает его IP и порт (например, curl 192.168.1.100:80).

1. **Настройка интерфейсов внутри контейнера**:

Приложение внутри контейнера (например, Gin с router.Run()) может выбирать, на каких интерфейсах слушать:

* 1. :8080 или 0.0.0.0:8080 — принимает запросы от всех (хост, другие контейнеры, сеть).
  2. localhost:8080 — принимает только от процессов внутри самого контейнера.
  3. Конкретный IP — принимает только от устройств с этим IP (если такой IP настроен).

### Управление множеством контейнеров

#### Описание

1. Докер имеет возможность управлять запуском и взаимодействием множества контейнеров
2. Можно указывать какие контейнеры нужно запускать, какой порядок запуска, нужна ли задержка при запуске
3. Это всё нужно, чтобы создать сложное приложение у которого много разнообразнных функций, каждая из которых представлена своим контейнером
4. Эти возможности реализованы в CLI утилите **docker-compose**.

Docker-compose ищёт в директории, откуда запущен терминал, специальный файл docker-compose.yaml – это файл скрипт, который определяет поведение и порядок запуска контейнеров.

1. docker-compose.yaml также имеет определённую структуру и синтаксис

#### Общая структура docker-compose.yaml и важные аспекты

1. Типичная структура docker-compose.yaml выглядит следующим образом:

version: '3.8' # Указываем версию Compose (можно использовать 3.x)

services:

go-app: # Имя сервиса (можно изменить)

build:

context: . # Контекст сборки — текущая директория

dockerfile: Dockerfile # Указываем имя Dockerfile (если оно другое, измените)

ports:

- "9090:8080" # Маппинг портов: "внешний:внутренний" (хост:контейнер)

networks:

- app-network # Имя сети (будет создано автоматически)

networks:

app-network: # Определение сети

driver: bridge # Используем мостовую сеть по умолчанию

1. Аспекты.  
     
   Глобальные секции:

version – версия формата yaml

**УСТАРЕЛА**

Можно просто не указывать эту секцию и всё  
  
services – описание представленных нам сервисов  
  
network – описание сети в которой взаимодействуют эти сервисы

#### Services

1. Секция services в свою очередь тоже состоит из нескольких подсекций:  
     
   подсекция исходных файлов сервиса (build/image) -   
   ports – прописывает какие порты прослушивает сервис  
   networks – прописывает в каких сетях

##### Исходные файлы сервиса:

1. Подсекция build внутри services – отвечает за создание контейнера с нуля из dockerfile и своего кода
2. Image вместо билд – указывает откуда взять уже готовый образ (обычно dockerhub)

###### Build

###### Image

#### Networks

1. networks:

app-network: # Имя сети, которое вы придумываете

driver: bridge # Драйвер, который управляет сетью

#### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### Утилита docker-compose

1. docker-compose up --build // со скачиванием или обновлением образов

docker-compose up // без обновления, а те которые сохранены

1. Опустить все контейнеры

docker-compose down

## Коммуникация между контейнерами

1. В разработке веб приложений важно настроить коммуникацию между контейнерами.  
     
   Подробнее это рассмотрено в Коммуникация между микросервисами  
   Прогр. Инженерия / 1 Основы разработки ИС / Фундаментальные архитектуры ИС / Микросервисные архитектуры / Коммуникация между микросервисами.word

# Виртуализация

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Материалы

## Возможности ядра линукс и девопс

1. Основные возможности ядра Linux:
2. Управление процессами: планирование (scheduling), приоритеты, fork/exec, сигналы.
3. Управление памятью: виртуальная память, paging, mmap, shared memory.
4. Управление устройствами: драйверы, виртуальная файловая система (VFS), udev.
5. Сетевой стек: TCP/IP, UDP, маршрутизация, firewall (Netfilter/iptables/nftables).
6. Безопасность: права доступа, namespaces, cgroups, SELinux/AppArmor, capabilities.
7. Файловые системы: ext4, XFS, Btrfs, NFS, overlayfs.
8. Изоляция и виртуализация: namespaces, cgroups, KVM, LXC.
9. Межпроцессное взаимодействие (IPC): очереди сообщений, семафоры, сокеты.
10. Модули ядра: загрузка и выгрузка драйверов/функционала без перезагрузки.

1. Что используют DevOps:
2. cgroups и namespaces → контейнеризация (Docker, Kubernetes).
3. overlayfs → слоёные файловые системы контейнеров.
4. Сетевой стек → настройка iptables/nftables, маршрутизация в Kubernetes.
5. SELinux/AppArmor, capabilities → безопасность контейнеров и сервисов.
6. Процессы и system calls → мониторинг (strace, perf, top, ps).
7. KVM → виртуальные машины (например, через libvirt, QEMU).
8. Файловые системы → NFS, ext4, XFS для стораджа в кластерах.
9. Модули ядра → загрузка сетевых/файловых драйверов при администрировании.