# 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

### Dockerfile

1. Dockerfile как скрипт описывающий операции, которые необходимо совершить для процесса, имеет определенную структуру – операции выполняются построчно сверху вниз.

Поэтому самые необходимые операции (вроде установки ОС) необходимо записывать в самом начале скрипта.

#### Общая структура

1. Типичная 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.

#### 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.

### 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).

### Docker compose

Makefile - файл,содержащий как бы hotkeys для прописанных внутри него команд

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

### Терминал

#### Запуск контейнеров

docker build -t go-gin-app .

docker run -p 9090:8080 go-gin-app

docker system prune - удалить все контейнеры

docker container ls -a - показать ID всех контейнеров

docker container rm cc3f2ff51cab - Удалить конкретный контейнер по ID

#### Http запросы

curl -X GET "http://localhost:9200/"

#### Зайти внутреннюю консоль контейнера

docker ps - найти имя контейнера

docker exec -it strange\_dirac bash - зайти в терминал контейнера

docker exec -it strange\_dirac sh - если bash не установлен

???

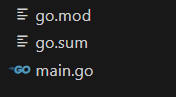
docker exec -it elasticsearch curl -X GET "localhost:9200/\_search?q=rocky+mountain"

exit - выход из терминала контейнера

### Начало

#### Запуск простого контейнера

##### 1) написать логику



main.go

package main

import (

"net/http"

"github.com/gin-gonic/gin"

)

type album struct {

ID string `json:"id"`

Title string `json:"title"`

Artist string `json:"artist"`

Price float64 `json:"price"`

}

var albums = []album{

{ID: "1", Title: "Blue Train", Artist: "John Coltrane", Price: 56.99},

{ID: "2", Title: "Jeru", Artist: "Gerry Mulligan", Price: 17.99},

{ID: "3", Title: "Sarah Vaughan and Clifford Brown", Artist: "Sarah Vaughan", Price: 39.99},

}

func getAlbums(c \*gin.Context) {

c.IndentedJSON(http.StatusOK, albums)

}

// postAlbums adds an album from JSON received in the request body.

func postAlbums(c \*gin.Context) {

var newAlbum album

// Call BindJSON to bind the received JSON to

// newAlbum.

if err := c.BindJSON(&newAlbum); err != nil {

return

}

// Add the new album to the slice.

albums = append(albums, newAlbum)

c.IndentedJSON(http.StatusCreated, newAlbum)

}

// getAlbumByID locates the album whose ID value matches the id

// parameter sent by the client, then returns that album as a response.

func getAlbumByID(c \*gin.Context) {

id := c.Param("id")

// Loop over the list of albums, looking for

// an album whose ID value matches the parameter.

for \_, a := range albums {

if a.ID == id {

c.IndentedJSON(http.StatusOK, a)

return

}

}

c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})

}

func main() {

router := gin.Default()

router.GET("/albums", getAlbums)

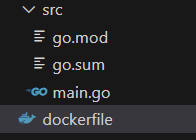
router.GET("/albums/:id", getAlbumByID)

router.POST("/albums", postAlbums)

router.Run("localhost:8080")

}

##### 2) написать dockerfile для логики



dockerfile:

FROM golang:1.24 AS builder

WORKDIR /app

# Копируем go.mod и go.sum из src

COPY src/go.mod src/go.sum ./

RUN go mod tidy

# Копируем весь src в рабочую директорию

COPY src/ .

# Сборка main.go

RUN GOOS=linux GOARCH=amd64 go build -o main main.go

# Финальный минимальный образ

FROM gcr.io/distroless/base

WORKDIR /root/

COPY --from=builder /app/main .

EXPOSE 8080

CMD ["./main"]

##### 3) Создание образа

в директории докерфайла прописать

docker build -t go-gin-app . - желтый это название образа

##### 4) Настройка портов

**Контейнер**

Внутри контейнера создаётся NAT, который слушает внутренний порт хоста (компьютера или сервера):

func main() {

router := gin.Default()

router.GET("/albums", getAlbums)

router.GET("/albums/:id", getAlbumByID)

router.POST("/albums", postAlbums)

router.Run("localhost:8080") // здесь мы заставляем слушать внутренний порт

}

Так контейнер слушает запросы приходящие на внутренний порт 8080 ТОЛЬКО локального хоста и больше НИКОГО  
Если сделать так 0.0.0.0:8080 - то тогда программа будет слушать ВСЕ запросы от всех устройств в сети.

**хост**

docker run -p 9090:8080 go-gin-app

Так мы говорим, чтобы хост слушал внешний порт 9090 и передавал его на внутренний 8080.

**dockerfile**

EXPOSE 8080

Это вообще не имеет отношение к портам, ни внутренним ни внешним

По сути ничего не делает, взуальная информация  
Но возможно в docker-compose может быть полезен

##### 5) запуск контейнера

docker run -p 9090:8080 go-gin-app

создание образа пользовательского контейнера

docker-compose.yaml автоматически запускает несколько таких контейнеров

#### Docker-compose up

##### 1) Создать файл инструкций по построению образов

На готовый dockerfile в туже директорию добавляем файл docker-compose.yaml

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, если не существует)

networks:

app-network: # Определение сети

driver: bridge # Используем мостовую сеть по умолчанию

##### 2) запустить через docker-compose

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

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

##### 3) Опустить все контейнеры

docker-compose down

### Работа

#### Простейший API

package main

import (

"encoding/json"

"log"

"net/http"

)

type Message struct {

Text string `json:"text"`

}

func helloHandler(w http.ResponseWriter, r \*http.Request) {

msg := Message{Text: "Привет из бэкенда на Go!"}

w.Header().Set("Content-Type", "application/json")

json.NewEncoder(w).Encode(msg)

}

func main() {

http.HandleFunc("/api/hello", helloHandler)

log.Fatal(http.ListenAndServe(":8080", nil))

}

#### HTTP запросы

##### Принять запрос

package main

import (

"fmt"

"io"

"log"

"net/http"

)

func queryHandler(w http.ResponseWriter, r \*http.Request) {

fmt.Println("Request Method:", r.Method)

fmt.Println("Request URL:", r.URL.String())

body, \_ := io.ReadAll(r.Body)

fmt.Println("Request Body:", string(body))

}

func main() {

http.HandleFunc("/api/process\_query", queryHandler)

log.Fatal(http.ListenAndServe(":8080", nil))

}

##### Отправка ответа

<https://labex.io/tutorials/go-how-to-write-http-response-body-450894>

Для ответа на запрос используют w http.ResponseWriter

Так например, можно использовать функцию вывода текста, для того, чтобы отправить ответ на запрос. Эта функция вернёт ответ

fmt.Println(w , “hello”)

Когда ты вызываешь:

json.NewEncoder(w).Encode(response)

происходит следующее:

Создаётся JSON-энкодер, который пишет не в файл, не в память, а прямо в HTTP-ответ (в поток, который ассоциирован с клиентским соединением).

В Go:

Ты не создаёшь ответ целиком заранее, как в некоторых фреймворках.

Ты записываешь в поток w по ходу работы обработчика.

Всё, что ты запишешь в w — окажется в body HTTP-ответа, когда обработчик завершится.

Метод Encode превращает структуру response в JSON-строку (с автоматическим добавлением \n на конце) и пишет этот JSON прямо в w — в тело HTTP-ответа.

Когда функция queryHandler завершает работу, сервер автоматически отправляет всё, что было записано в w, в качестве тела ответа клиенту.

А заголовки (вроде Content-Type) тоже уже готовы к этому моменту

##### Совершение своего запроса

# 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.