Вычислительная инфраструктура в задачах биоинформатики

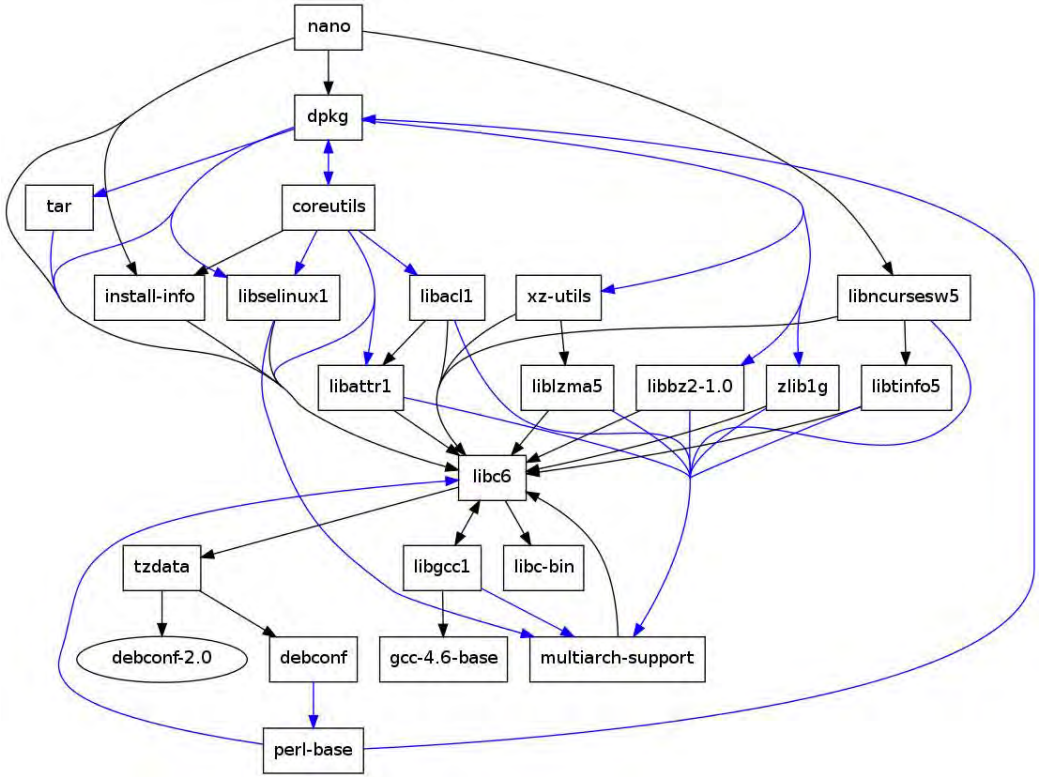
Михайлова Анастасия
aamikhaylova_9@miem.hse.ru
Tg: @Anst_M

08.11.2023/20.11.2024

Проблемы и решения

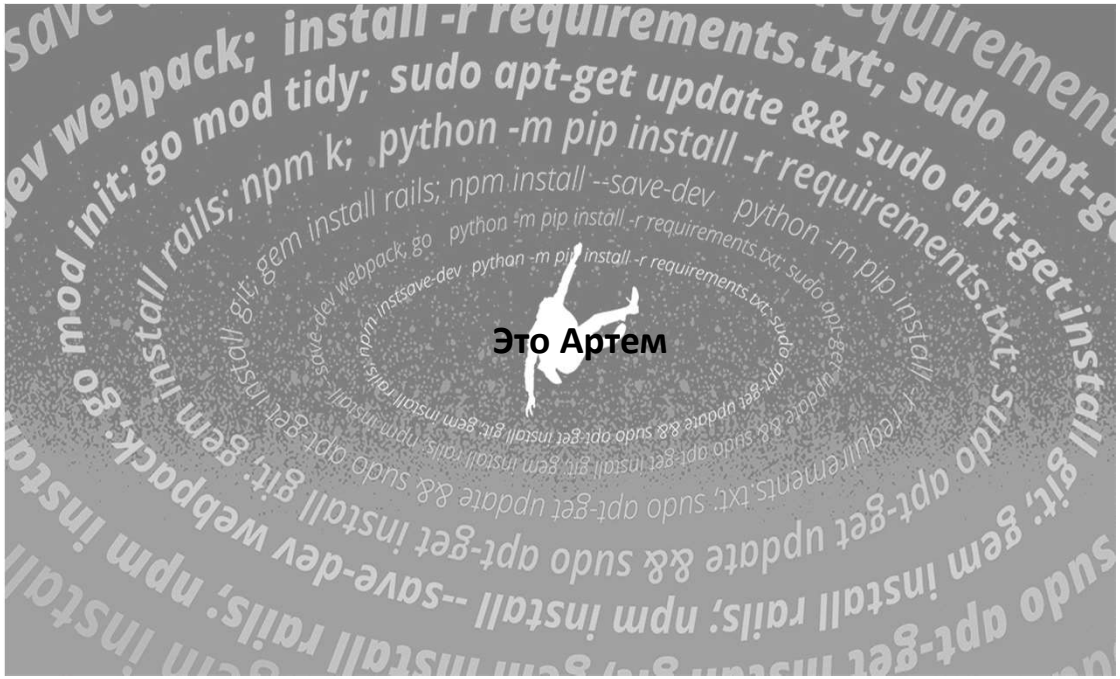
Менеджеры пакетов

- Имеем дело с пакетами (файл, скрипт и тд)
- Что есть в пакете:
  - Как собрать пакет (компилировать? Как и чем?)
  - Куда его положить
  - Какие зависимости нужны
- Примеры:
  - Conda
  - Snap
  - Apt
- Зависят от языка:
  - Pip, pip3 – Python
  - CRAN – R
  - Conan – C++, C

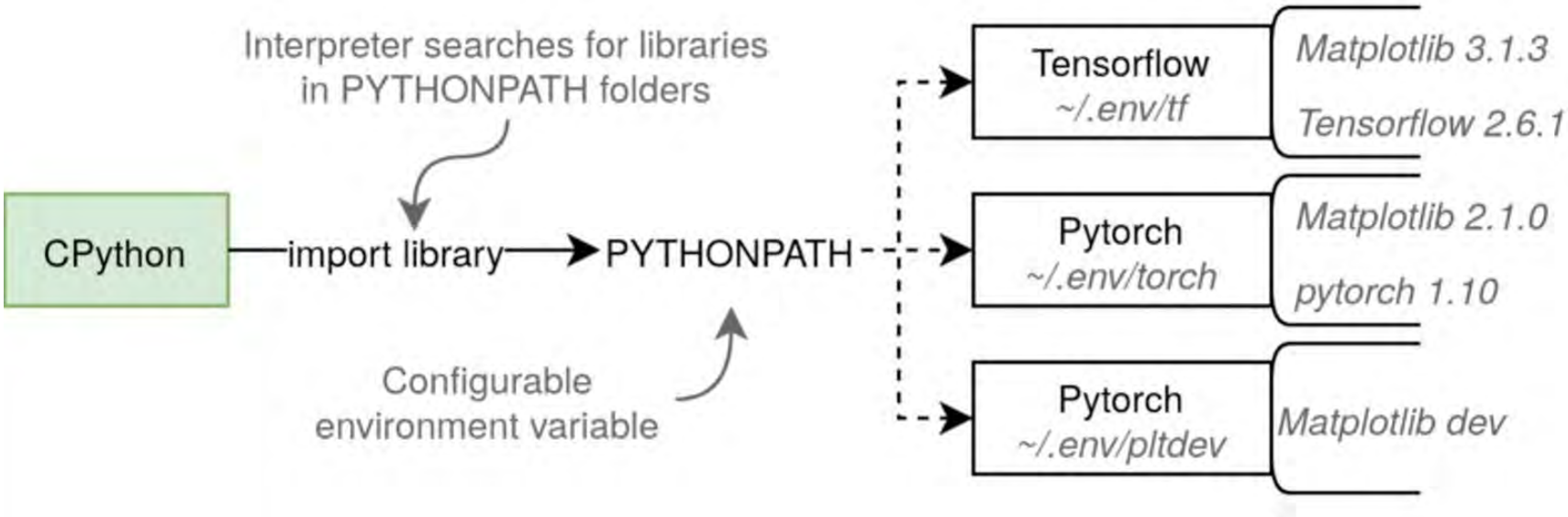


Ад зависимостей

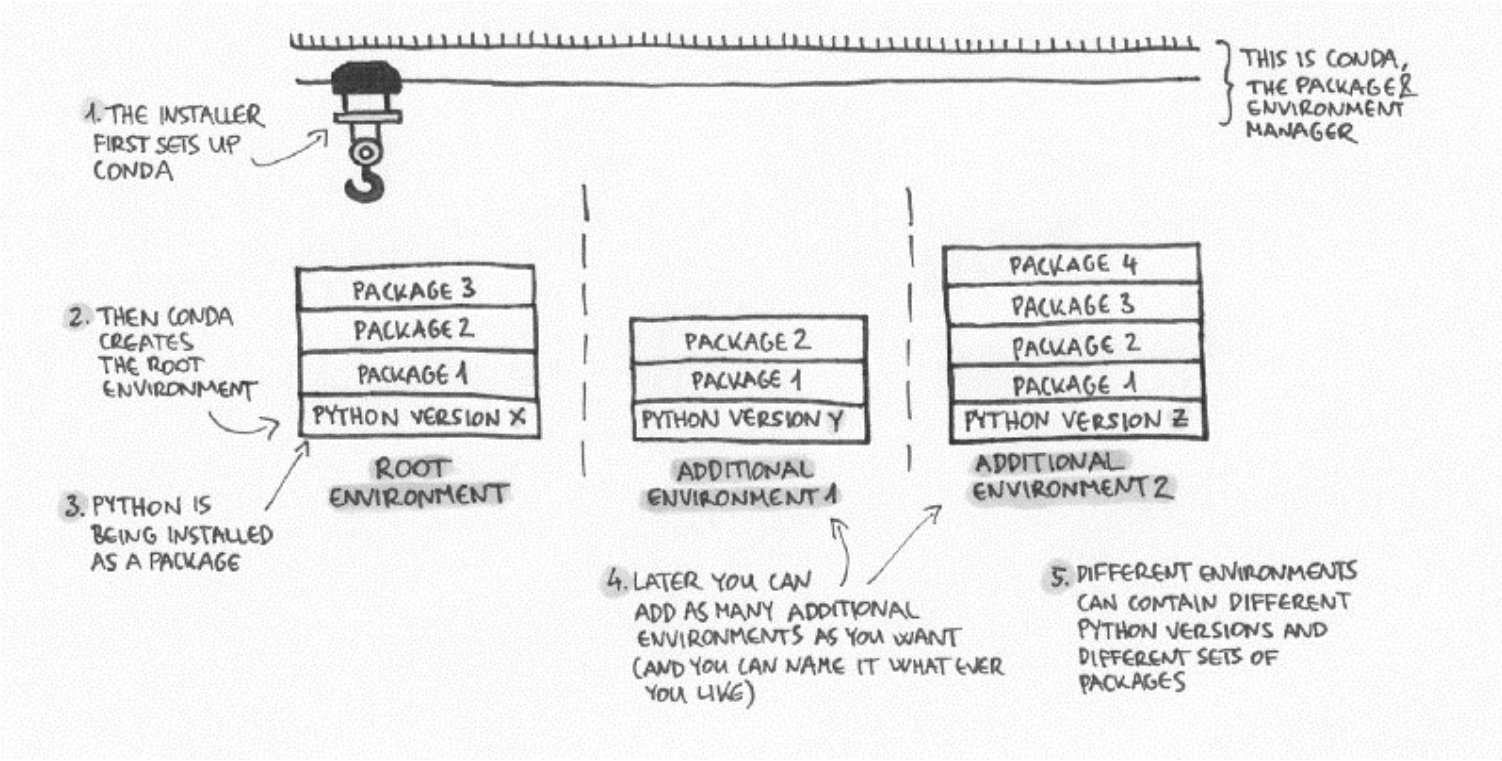
- При установке или обновлении одного пакета необходимо также устанавливать или обновлять другие пакеты, от которых зависит первый
- Когда вы устанавливаете новый пакет, пакетный менеджер должен удовлетворить все зависимости этого пакета, что может привести к автоматической установке или обновлению множества других пакетов. Это поддерживает целостность системы, обеспечивая, что все компоненты программного обеспечения будут совместимыми друг с другом.



Виртуальные окружения

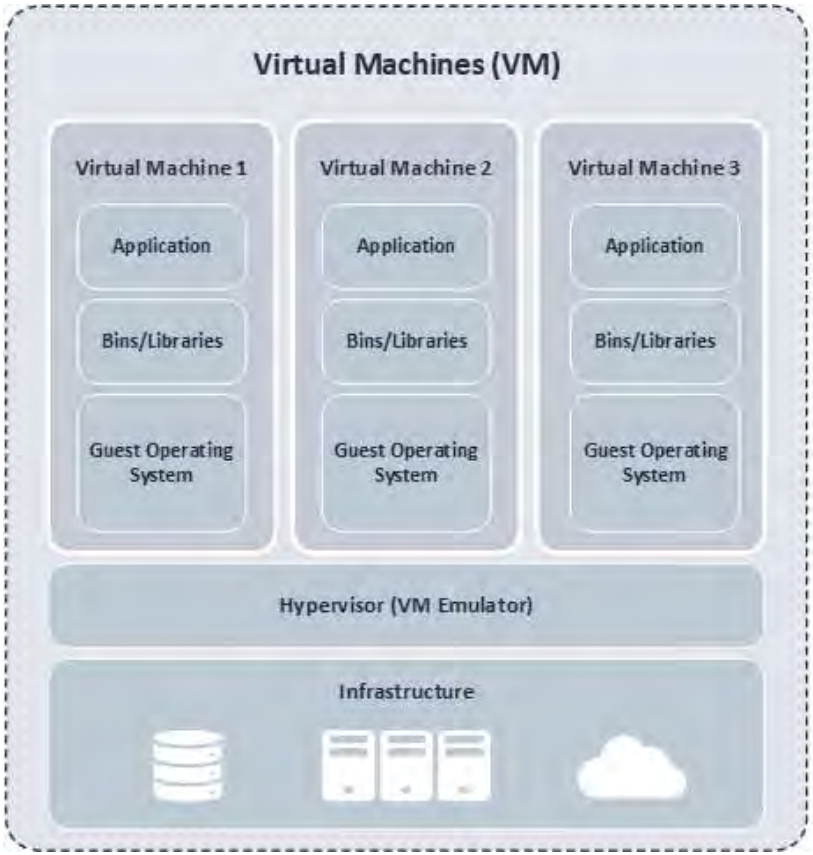


Конда

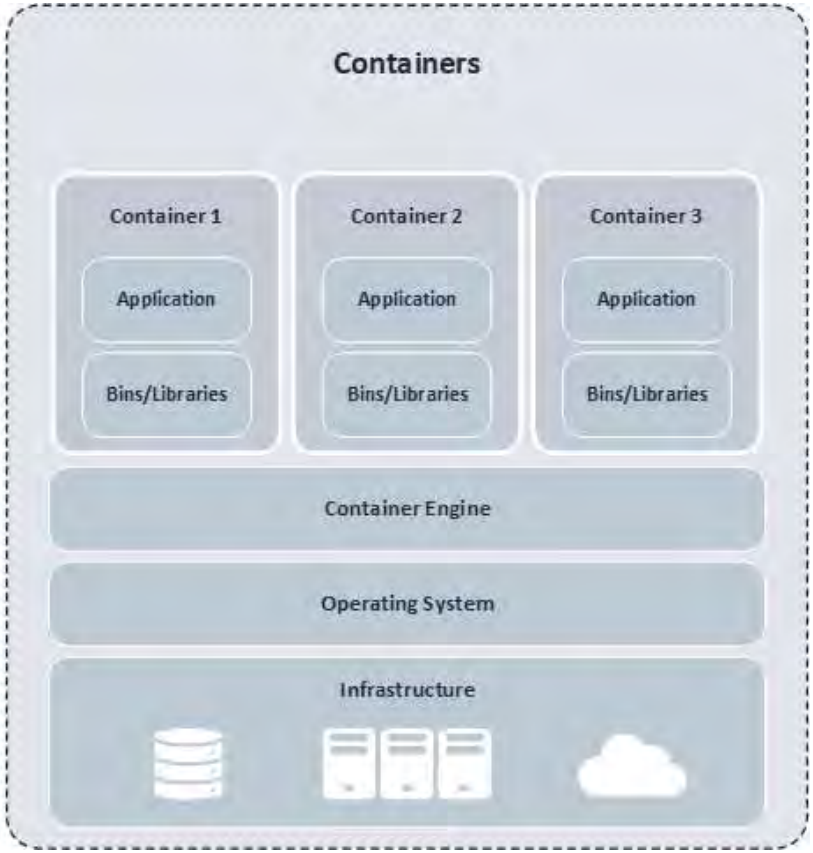

THIS IS CONDA, THE PACKAGE & ENVIRONMENT MANAGER
1. THE INSTALLER FIRST SETS UP CONDA
2. THEN CONDA CREATES THE ROOT ENVIRONMENT
3. PYTHON IS BEING INSTALLED AS A PACKAGE
PACKAGE 3
PACKAGE 2
PACKAGE 1
PYTHON VERSION X
ROOT ENVIRONMENT
PACKAGE 2
PACKAGE 1
PYTHON VERSION Y
ADDITIONAL ENVIRONMENT 1
PACKAGE 4
PACKAGE 3
PACKAGE 2
PACKAGE 1
PYTHON VERSION Z
ADDITIONAL ENVIRONMENT 2
4. LATER YOU CAN ADD AS MANY ADDITIONAL ENVIRONMENTS AS YOU WANT (AND YOU CAN NAME IT WHAT EVER YOU LIKE)
5. DIFFERENT ENVIRONMENTS CAN CONTAIN DIFFERENT PYTHON VERSIONS AND DIFFERENT SETS OF PACKAGES

Мамба

Больше изоляции



Контейнеры


Containers
Container 1
Application
Bins/Libraries
Container 2
Application
Bins/Libraries
Container 3
Application
Bins/Libraries
Container Engine
Operating System
Infrastructure

Основы

Контейнер, docker контейнер

- В нём можно что-то хранить
- Его можно переносить.
- В контейнер удобно что-то класть и удобно что-то из него вынимать.
- Если вам нужен контейнер, его можно купить

Виртуальная машина

Контейнер

Преимущества Docker

- **Минимальное потребление ресурсов** — контейнеры не виртуализируют всю операционную систему (ОС), а используют ядро хоста и изолируют программу на уровне процесса. Последний потребляет намного меньше ресурсов локального компьютера, чем виртуальная машина.
- **Скоростное развертывание** — вспомогательные компоненты можно не устанавливать, а использовать уже готовые docker-образы (шаблоны). Например, не имеет смысла постоянно устанавливать и настраивать Linux Ubuntu. Достаточно 1 раз ее инсталлировать, создать образ и постоянно использовать, лишь обновляя версию при необходимости.
- **Удобное скрытие процессов** — для каждого контейнера можно использовать разные методы обработки данных, скрывая фоновые процессы.
- **Работа с небезопасным кодом** — технология изоляции контейнеров позволяет запускать любой код без вреда для ОС.
- **Простое масштабирование** — любой проект можно расширить, внедрив новые контейнеры.
- **Удобный запуск** — приложение, находящееся внутри контейнера, можно запустить на любом docker-хосте.
- **Оптимизация файловой системы** — образ состоит из слоев, которые позволяют очень эффективно использовать файловую систему.

Термины и концепции

Компоненты Docker

сервер контейнеров, входящий в состав программных средств Docker.

файл, включающий зависимости, сведения, конфигурацию для дальнейшего развертывания и инициализации контейнера.

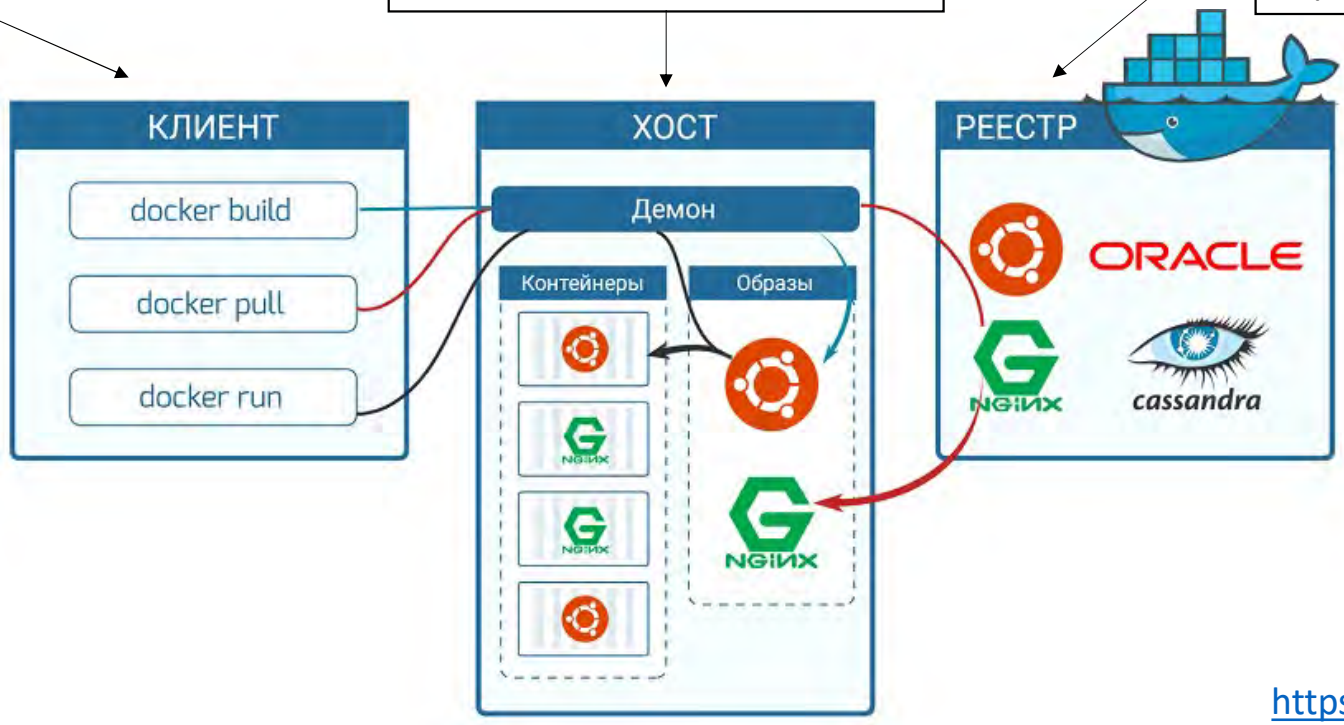
это инструкция для запуска приложения: код, среду выполнения, системные инструменты, системные библиотеки и настройки.

Компоненты Docker

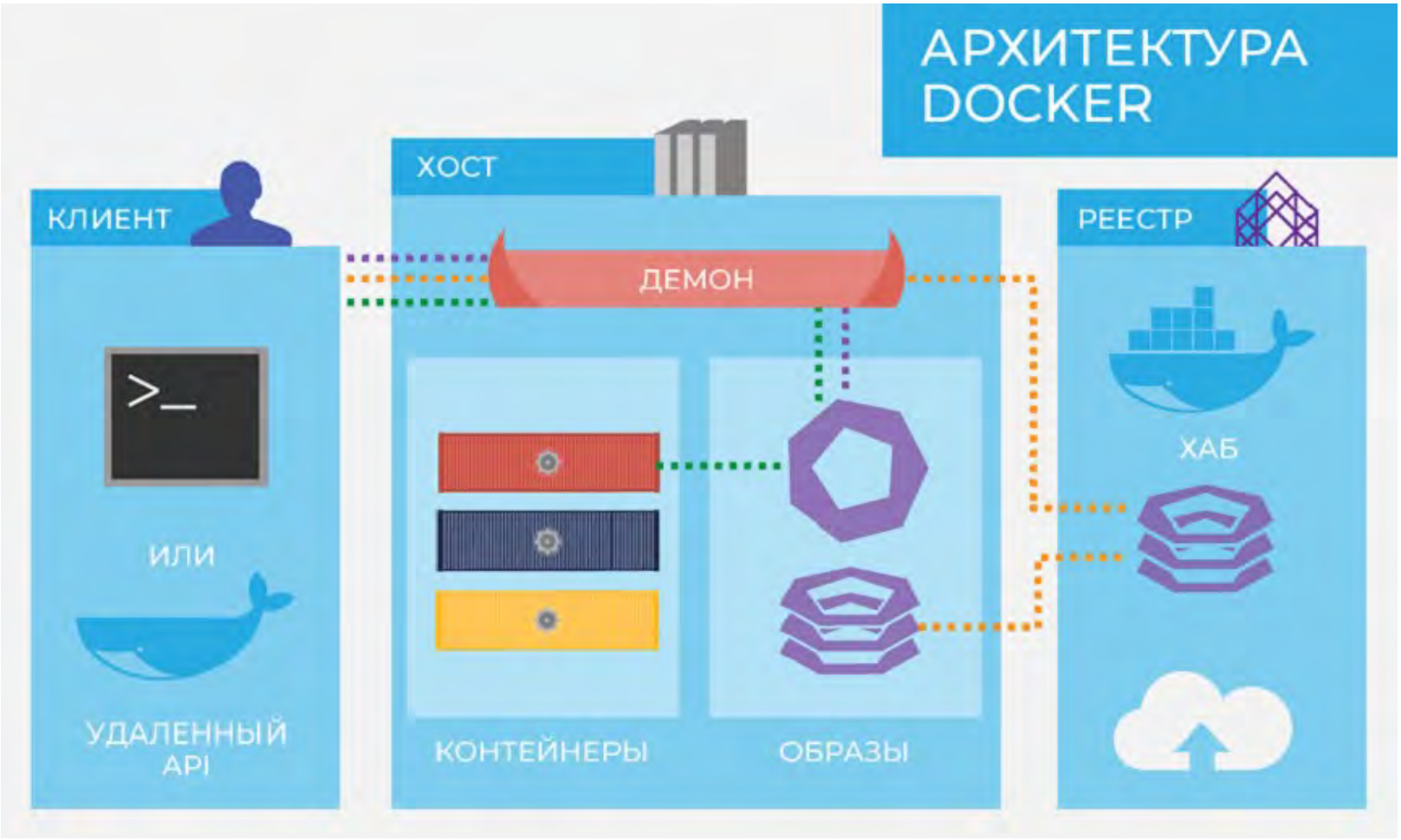
интерфейс взаимодействия пользователя с Docker-демоном

машинная среда для запуска контейнеров с программным обеспечением.

зарезервированный сервер, используемый для хранения docker-образов



https://hub.docker.com/


АРХИТЕКТУРА DOCKER
ХОСТ
КЛИЕНТ
РЕЕСТР
ДЕМОН
>_
ИЛИ
ХАБ
УДАЛЕННЫЙ API
КОНТЕЙНЕРЫ
ОБРАЗЫ

Что нужно сделать, чтобы собрать свой образ

Dockerfile

Шаг 1. Написание инструкций

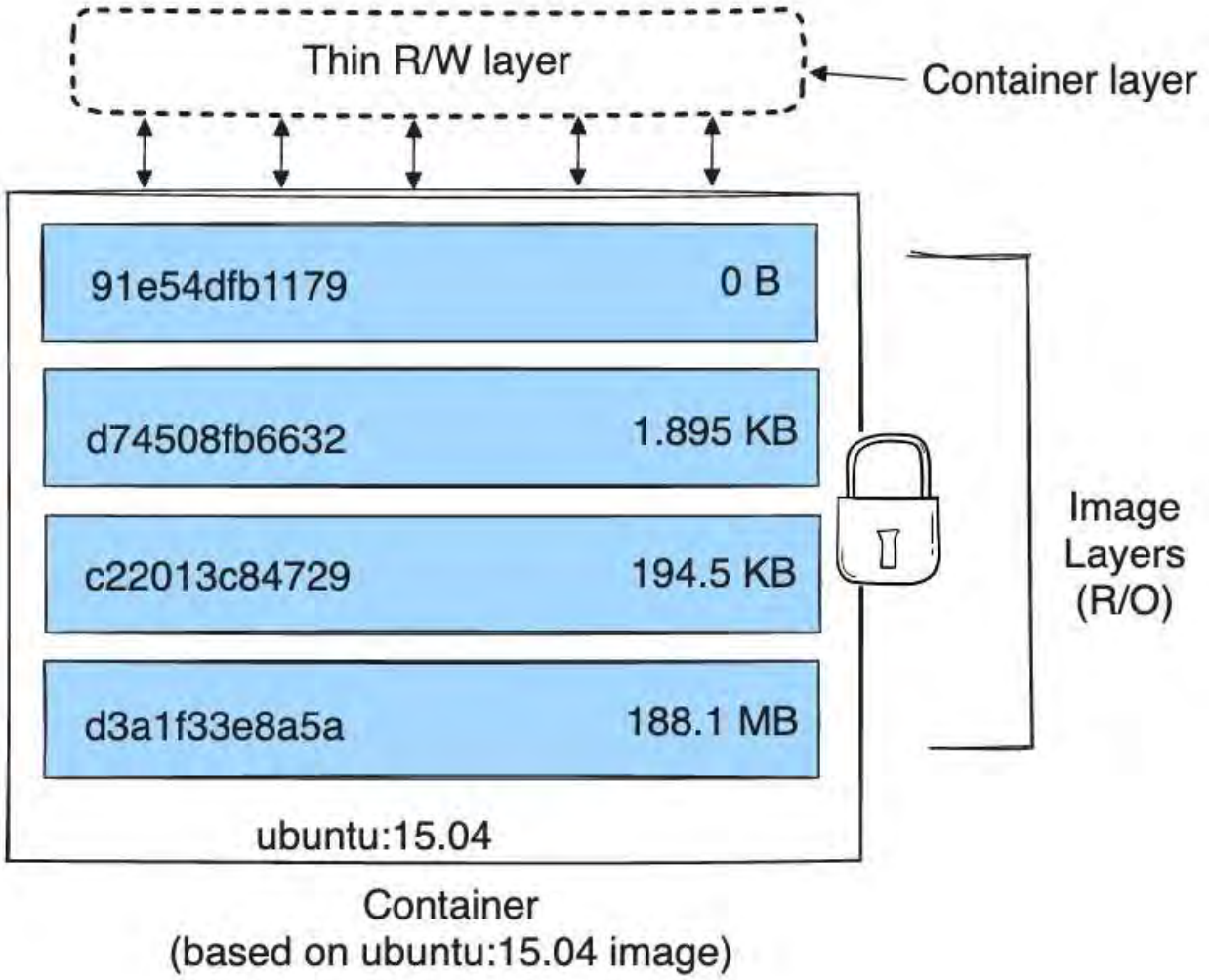
Начало начал или что-то попроще

```
FROM ubuntu:18.04
COPY . /app
```

Задаем базовый образ

Слои в итоговом образе создают только инструкции **FROM**, **RUN**, **COPY**, и **ADD**.

Другие инструкции что-то настраивают, описывают метаданные, или сообщают Docker о том, что во время выполнения контейнера нужно что-то сделать, например — открыть какой-то порт или выполнить какую-то команду.


Thin R/W layer
Container layer
91e54dfb1179
0 B
d74508fb6632
1.895 KB
c22013c84729
194.5 KB
d3a1f33e8a5a
188.1 MB
Image
Layers
(R/O)
ubuntu:15.04
Container
(based on ubuntu:15.04 image)

Основные инструкции

- FROM — задаёт базовый (родительский) образ.
- LABEL — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ.
- ENV — устанавливает постоянные переменные среды.
- RUN — выполняет команду и создаёт слой образа. Используется для установки в контейнер пакетов.
- COPY — копирует в контейнер файлы и папки.
- ADD — копирует файлы и папки в контейнер, может распаковывать локальные .tar-файлы.
- 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.
- WORKDIR — задаёт рабочую директорию для следующей инструкции.
- ARG — задаёт переменные для передачи Docker во время сборки образа.
- ENTRYPOINT — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются.

Что-то сложнее

```
FROM python:3.7.2-alpine3.8
```

Задаем образ

Данный базовый образ включает в себя Linux, Python. Образы ОС Alpine весьма популярны в мире Docker. Дело в том, что они отличаются маленькими размерами, высокой скоростью работы и безопасностью. Однако образы Alpine не отличаются широкими возможностями, характерными для обычных операционных систем. Поэтому для того, чтобы собрать на основе такого образа что-то полезное, создателю образа нужно установить в него необходимые ему пакеты.

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
```

Добавляем мета-данные

Инструкция LABEL (метка) позволяет добавлять в образ метаданные. Объявление меток не замедляет процесс сборки образа и не увеличивает его размер. Они лишь содержат в себе полезную информацию об образе Docker, поэтому их рекомендуется включать в файл.

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"
```

Задаем константы для контейнера

Инструкция ENV позволяет задавать постоянные переменные среды, которые будут доступны в контейнере во время его выполнения. В предыдущем примере после создания контейнера можно пользоваться переменной ADMIN.

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"
RUN apk update && apk upgrade && apk add bash
```

Обновляем систему и ставим bash

Инструкция RUN позволяет создать слой во время сборки образа. После её выполнения в образ добавляется новый слой, его состояние фиксируется. Инструкция RUN часто используется для установки в образы дополнительных пакетов.

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"
RUN apk update && apk upgrade && apk add bash
COPY . ./app
```

Копирование данных в образ

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"
RUN apk update && apk upgrade && apk add bash
COPY . ./app
ADD https://raw.githubusercontent.com/discdiver/pachy-vid/master/sample_vids/vid1.mp4 \
/my_app_directory
```


Скачивание файла в образ

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"
RUN apk update && apk upgrade && apk add bash
COPY . ./app
ADD https://raw.githubusercontent.com/discdiver/pachy-vid/master/sample_vids/vid1.mp4 \
/my_app_directory
RUN ["mkdir", "/a_directory"]
```

Создаем директорию

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"
RUN apk update && apk upgrade && apk add bash
COPY . ./app
ADD https://raw.githubusercontent.com/discdiver/pachy-vid/master/sample_vids/vid1.mp4 \
/my_app_directory
RUN ["mkdir", "/a_directory"]
CMD ["python", "./my_script.py"]
```

Выполняем скрипт

Еще один пример

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
# Устанавливаем зависимости
RUN apk add --update git
# Задаём текущую рабочую директорию
WORKDIR /usr/src/my_app_directory
# Копируем код из локального контекста в рабочую директорию образа
COPY . .
# Задаём значение по умолчанию для переменной
ARG my_var=my_default_value
# Настраиваем команду, которая должна быть запущена в контейнере во время его выполнения
ENTRYPOINT ["python", "./app/my_script.py", "my_var"]
```

Команды

Шаг 2. Манипуляции с образами и контейнерами

Начало начал

```
docker
```

```
docker version
```

```
docker system prune
```

Команды

Образы

Список основных команд

```
docker image my_command
```

- build — сборка образа.
- push — отправка образа в удалённый реестр.
- ls — вывод списка образов.
- history — вывод сведений о слоях образа.
- inspect — вывод подробной информации об образе, в том числе — сведений о слоях.
- rm — удаление образа.

Создание образов

```
docker image build -t my_repo/my_image:my_tag .
```

```
docker login
```

```
docker image push my_repo/my_image:my_tag
```

Исследование образов

```
docker image ls
```

```
docker image history my_image
```

```
docker image inspect my_image
```

Удаление образов

```
docker image rm my_image
```

```
docker image rm $(docker images -a -q)
```

Команды

Контейнеры

Список основных команд

```
docker container my_command
```

- create — создание контейнера из образа.
- start — запуск существующего контейнера.
- run — создание контейнера и его запуск.
- ls — вывод списка работающих контейнеров.
- inspect — вывод подробной информации о контейнере.
- logs — вывод логов.
- stop — остановка работающего контейнера с отправкой главному процессу контейнера сигнала SIGTERM, и, через некоторое время, SIGKILL.
- kill — остановка работающего контейнера с отправкой главному процессу контейнера сигнала SIGKILL.
- rm — удаление остановленного контейнера.

Создание контейнеров

```
docker container create my_repo/my_image:my_tag
```

```
docker container start my_container
```

```
docker container run my_image
```

```
docker container run -it my_image my_command
```

Проверка состояние контейнера

```
docker container ls
```

```
docker container ls -a -s
```

```
docker container inspect my_container
```

Остановка и удаление контейнера

```
docker container stop my_container
```

```
docker container kill my_container
```

```
docker container kill $(docker ps -q)
```

Уменьшение размерности и ускорение сборки

Кэширование

- Кэширование можно отключить, передав ключ --no-cache=True команде docker build.
- Если вы собираетесь вносить изменения в инструкции Dockerfile, тогда каждый слой, созданный инструкциями, идущими после изменённых, будет достаточно часто собираться повторно, без использования кэша. Для того чтобы воспользоваться преимуществами кэширования, помещайте инструкции, вероятность изменения которых высока, как можно ближе к концу Dockerfile.
- Объединяйте команды RUN apt-get update и apt-get install в цепочки для того, чтобы исключить проблемы, связанные с неправильным использованием кэша.
- Если вы используете менеджеры пакетов, наподобие pip, с файлом requirements.txt, тогда придерживайтесь нижеприведённой схемы работы для того, чтобы исключить использование устаревших промежуточных образов из кэша, содержащих набор пакетов, перечисленных в старой версии файла requirements.txt

```
COPY requirements.txt /tmp/
RUN pip install -r /tmp/requirements.txt
COPY . /tmp/
```

Подбор базового образа

Имя образа	Вес
ubuntu	500 MB
python	600 MB
openjdk	700 MB
alpine	200 MB
nginx	300 MB

Файл .dockerignore

Применение файлов .dockerignore позволяет:

- Исключать из состава образа файлы, содержащие секретные сведения наподобие логинов и паролей.
- Уменьшить размер образа. Чем меньше в образе файлов — тем меньше будет его размер и тем быстрее с ним можно будет работать.
- Уменьшить число поводов для признания недействительным кэша при сборке похожих образов. Например, если при повторной сборке образа меняются некие служебные файлы проекта, наподобие файлов с журналами, из-за чего данные, хранящиеся в кэше, по сути, необоснованно признаются недействительными, это замедляет сборку образов.

Основные рекомендации

- Используйте всегда, когда это возможно, официальные образы в качестве базовых образов. Официальные образы регулярно обновляются, они безопаснее неофициальных образов.
- Для того чтобы собирать как можно более компактные образы, пользуйтесь базовыми образами, основанными на Alpine Linux.
- Если вы пользуетесь apt, комбинируйте в одной инструкции RUN команды apt-get update и apt-get install. Кроме того, объединяйте в одну инструкцию команды установки пакетов. Перечисляйте пакеты в алфавитном порядке на нескольких строках, разделяя список символами \
- Включайте конструкцию вида && rm -rf /var/lib/apt/lists/* в конец инструкции RUN, используемой для установки пакетов. Это позволит очистить кэш apt и приведёт к тому, что он не будет сохраняться в слое, сформированном командой RUN. Подробности об этом можно почитать в документации.
- Разумно пользуйтесь возможностями кэширования, размещая в Dockerfile команды, вероятность изменения которых высока, ближе к концу файла.
- Пользуйтесь файлом .dockerignore.
- Не устанавливайте в образы пакеты, без которых можно обойтись.

Пример для пункта 3

```
RUN apt-get update && apt-get install -y \
    package-one \
    package-two \
    package-three
 && rm -rf /var/lib/apt/lists/*
```