ДЕПАРТАМЕНТ ОБРАЗОВАНИЯ И НАУКИ ГОРОДА МОСКВЫ

Государственное бюджетное профессиональное образовательное   
учреждение города Москвы

«Колледж автоматизации и информационных технологий №20»

**ИНДИВИДУАЛЬНЫЙ ПРОЕКТ**

на тему

**«Docker контейнеризация»**

по дисциплине

«Индивидуальный проект»

Обучающийся:

Хаксар М.Н

УО ДАТАХАБ курс 1 группа ИСП131

Руководитель ИП:

Ломаченкова А.И.

**МОСКВА**

**2024**

**Структура проекта:**

**ВВЕДЕНИЕ 3-6**

1. Актуальность.

2. Цель проекта.

3. Задачи проекта.

4. Гипотеза. Продукт.

5.Целевая аудитория.

6.Объект исследования.

7.Предмет исследования.

**1 ГЛАВА 6-15**

1.1. Терминология. 6-8

1.2. Историческая основа. 8

1.3. Причины использования. 9-10

1.4. Области эксплуатации. 10-11

1.5. Теоретические основы. Компоненты. 11-15

1.6. Отличие контейнера от виртуальной машины. 15-16

2 ГЛАВА

2.1. Конструирование ботов, подготовка к деплою.

2.2. Создание и локальное тестирование контейнеров.

2.3. Развертывание приложения, проверка работоспособности.

**ЗАКЛЮЧЕНИЕ**

**СПИСОК ЛИТЕРАТУРЫ**

**ВВЕДЕНИЕ**

**Актуальность**

Docker – это платформа, позволяющая автоматизировать развертывание, масштабирование и управление приложениями с использованием контейнеров.

Контейнеры представляют собой легковесные, изолированные среды (операционные системы), в которых можно запускать приложения и их зависимости. Это значительно упрощает процесс разработки и развертывания, обеспечивая согласованность на разных стадиях жизненного цикла программного обеспечения.

Платформа позволяет быстро тестировать и выкладывать приложения, запускать на одной машине требуемое количество контейнеров, что делает её лидером среди малочисленных аналогов.

По данным опросов, таких как Stack Overflow Developer Survey, около 30-40% разработчиков используют контейнерные технологии в своих проектах, а Docker занимает около 83% на рынке контейнеризации в целом.

По статистике сайта TheirStack.com более 100 000 организаций уже активно используют Docker контейнеризацию в своих сервисах.Среди них: Yandex, eBay, Spotify, Netflix, PayPal. Число этих компаний растёт с каждым днём. Сумма дохода первых десяти организаций в подборке составила более 135 миллиардов долларов.

Сама же организация настолько разрослалсь, что к 2022 году смогла привлечь финансирование в размере более 100 миллионов долларов.

По отчетам: рынок контейнеризации ожидает роста на 25-30% в год; к 2025 году его объем может достигнуть более 10 миллиардов долларов.

Благодаря своей эффективности и отсутствию большой конкуренции на рынке, Docker-контейнеризация является ведущим инструментом на рынке цифровых технологий, что, несомненно, говорит о его высокой актуальности.

**Цель проекта**

Разработать практические материалы, где будет описываться процесс создания и контейнеризации Python-приложений с использованием Docker, а также разбор необходимой теории для воспроизведения результатов.(конкретизировать) старый

Разобрать на примере контейнеризации python ботов и деплоя их на площадку back4app с использованием инструментов git и нестандартных библиотек python возможности применения Docker контейнеризации и деплоя на сервер

**Задачи проекта**

1. **Изучить теоретические основы Docker**

2. Разработать файл с зависимостями, Dockerignore

3. Разработать два Python приложения, которые будут использованы для примера

3. **Разработать Dockerfile**

4. **Собрать Docker-образ:**

5. **Развернуть приложение на веб-сервере**

6. **Протестировать развернутое приложение**

7. **Вывод**

**Продукт.**

**Продукт:**

* 2 python-приложения.
* Dockerfile, dockerignore, 2 requirements (файлы с базовыми зависимостями)
* Документация, включающая всё, что нужно для ознакомления с базовыми понятиями и функциями Docker, в том числе: с git и github, некоторыми модулями python, площадкой back4app. Вопрос в том, что это далеко не все «базовые понятия» Стоит ли запариваться над этим и перефразировать ?

**Целевая аудитория**

* Студенты IT направлений
* Профессиональные разработчики
* Dev-Ops инженеры
* Работники узконаправленных специальностей в IT
* Организации и группы разработчиков

**Объект исследования**

Dev-Ops инженерия.

**Предмет исследования**

Контейнеризация и деплой python приложений.

**ГЛАВА 1**

**1.1. Терминология**

Общие термины

* **Развертывание (Деплой)**: Процесс переноса готового приложения или системы из среды разработки в среду, где оно будет доступно конечным пользователям (в основном серверы).
* **Автоматизация**: Использование технологий для выполнения задач без ручного вмешательства.
* **Масштабируемость**: Способность системы справляться с увеличением нагрузки путем добавления ресурсов (масштабирование "вверх") или дополнительных экземпляров системы (масштабирование по горизонтали).
* **Воспроизводимость**: Возможность многократно создавать одинаковое окружение и результаты, что важно для тестирования и развертывания.
* **Конфигурация** **(Configuration)**: Настройка параметров и окружения приложения для его правильной работы.
* **Окружение:** обозначение конкретного набора переменных, настроек, путей, зависимостей и других параметров, которые необходимы приложению для корректной работы в конкретной среде.
* **Репозиторий:** Хранилище, которое может быть удалённым или локальным (GitHub, DockerHub)
* **Среда (Environment):** конкретные условия и ресурсы, необходимые для запуска, тестирования и работы приложения.
* **Веб-сервер:** Программное обеспечение, которое принимает запросы от клиентов (браузеров) и отправляет им веб-страницы или другие данные.
* **Непрерывное развертывание:** автоматизированный подход, при котором каждый коммит автоматически развертывается в производственную среду.
* **Переменные окружения (Environment variables):** Настройки, которые хранятся в операционной системе и могут быть доступны приложениям во время выполнения. Часто используются для хранения конфиденциальных данных (паролей, ключей API и т.д.) и настроек приложения, которые могут меняться в разных средах.
* **Зависимости:** конфигурации, файлы, модули, которые нужны для работы приложения.
* **Продакшн (Production):** конечный продукт.
* **API:** программный интерфейс, позволяющий связывать между собой различные приложения.

Docker термины

* **Контейнер(Container):** Изолированная среда, в которой приложение может выполняться.
* **Docker**: Платформа для автоматизации развертывания приложений в контейнерах.
* **Dockerfile**: Текстовый файл, содержащий все команды, которые нужно выполнить для сборки образа Docker.
* **Образ (Image)**: Статический файл, содержащий все необходимые зависимости и конфигурации для запуска контейнера. Образ создается на основании Dockerfile.
* **Базовый образ**: Образ, который является основой для построения другого.
* **.dockerignore**: Файл, где перечисляются ненужные для работы контейнера файлы, папки и. т. д.
* **Docker host**: машина, на котором установлен Docker.
* **Docker daemon**: центральный системный компонент, который управляет всеми процессами Docker.

**Git термины**

* **GitHub**: Платформа для хостинга репозиториев git, которая позволяет управлять версиями кода и сотрудничать с другими разработчиками.
* **Коммит(commit):** Способ сохранения изменений в коде.
* **Push:** Команда, позволяющая отправить коммит в репозиторий.
* **.gitignore:** Файл, где перечисляются ненужные для репозитория файлы, папки и. т. д.

Python термины

* **pip**: Пакетный менеджер для Python, используемый для установки и управления библиотеками.
* **requirements.txt**: Файл, содержащий список зависимостей Python-проекта.

**1.2. История создания**

Виртуальная контейнеризация возникла как ответ на потребности в более эффективном использовании ресурсов вычислительных систем. Её основой стали технологии виртуализации, появившиеся в 1960-х годах. Первые серверы позволяли запускать несколько операционных систем на одном физическом сервере, что значительно увеличивало степень загрузки оборудования.

Позже возникла потребность в более легковесных решениях, и в 6 августа 2008 года на свет появились LXC (Linux Containers) и схожие аналоги.

Docker задумывался как способ работы с контейнерами, вдохновлённый LXC, но гораздо проще и удобнее, дружелюбнее к неподготовленному пользователю. В 2013 году Docker был запущен как open-source проект, что привело к значительному росту сообщества и появлению экосистемы инструментов.

**1.3. Причины использования**

Главная цель Docker — обеспечение универсальности и согласованности работы программного обеспечения, независимо от используемой платформы и специфики окружения. Это достигается посредством технологии контейнеризации, которая изолирует приложение и все его зависимости в единый, самодостаточный блок. Традиционно, разработчики сталкиваются с проблемой "работает на моей машине", когда приложение, успешно функционирующее в среде разработки, может вызывать сбои и конфликты при переносе в тестовую или, что критичнее, в рабочую среду. Эти проблемы обусловлены различиями в операционных системах, версиях установленных библиотек, конфигурациях сетевого окружения и других факторах, которые сложно отследить и воспроизвести в разных окружениях. Docker устраняет этот разрыв, предоставляя стандартизированный способ упаковки приложений, что значительно снижает вероятность возникновения таких проблем. Благодаря контейнеризации можно запускать свои программы практически на любом устройстве, где можно установить Docker Engine, будь то сервер, персональный компьютер, ноутбук или даже роутер. Это открывает новые возможности для развертывания приложений в самых разнообразных условиях, от локальной разработки до облачных вычислений.

Кроме того, использование Docker способствует более эффективному использованию ресурсов, так как контейнеры значительно легче виртуальных машин (ВМ). Контейнеры используют ядро хостовой операционной системы, в то время как ВМ эмулируют целую операционную систему. Это приводит к тому, что контейнеры занимают меньше дискового пространства, потребляют меньше памяти и быстрее запускаются. Контейнеры могут быть запущены на одном сервере в большом количестве, что позволяет повысить плотность размещения приложений и снизить затраты на инфраструктуру. В итоге, Docker становится ключевым инструментом для современных подходов к разработке и развёртыванию приложений, обеспечивая высокую гибкость, масштабируемость, эффективность использования ресурсов и ускоряя процессы разработки, тестирования и эксплуатации.

Из-за вышеперечисленных плюсов, Docker стал незаменимым инструментом в контексте микросервисной архитектуры, где приложение состоит из множества независимых сервисов. Каждый сервис может быть упакован в отдельный контейнер, что позволяет разрабатывать, тестировать и развертывать сервисы независимо друг от друга. Это обеспечивает большую гибкость и устойчивость системы в целом, позволяя командам быстро реагировать на изменения и сбои. Использование Docker Compose и других инструментов управления контейнерами позволяет эффективно управлять множеством взаимосвязанных сервисов и автоматизировать их развертывание и масштабирование.

**1.4. Сферы эксплуатации**

Docker нашел широкое применение в множестве сфер, включая веб-разработку, автоматизацию тестирования, DevOps, микросервисную архитектуру, научные вычисления и машинное обучение (машинное обучение и анализ данных требуют множества зависимостей и библиотек, что делает Docker идеальным вариантом).

В веб-разработке Docker особенно ценен за возможность создания сред, максимально приближенных к боевым условиям. Разработчики могут использовать Docker для создания контейнеров, в которых будут установлены те же самые версии операционной системы, библиотек и других зависимостей, что и на Продакшн-сервере. Это исключает конфликты версий и гарантирует, что приложение будет работать предсказуемо в любой среде. Docker также способствует созданию более согласованной среды разработки, когда все участники команды используют одинаковые конфигурации.

DevOps-инженеры (специалисты, которые объединяют разработку программного обеспечения и администрирование с целью повышения эффективности и качества разработки, развертывания и эксплуатации приложений) используют Docker для упрощения развертывания приложений и управления их жизненным циклом. Docker автоматизирует процесс упаковки, распространения и запуска приложений, что позволяет командам быстро и эффективно вносить изменения и релизы. Docker также упрощает процесс управления инфраструктурой, предоставляя стандартный способ определения и запуска приложений, что позволяет быстрее масштабировать ресурсы в зависимости от текущей нагрузки.

Тестировщики могут использовать Docker для создания воспроизводимых сред, в которых они могут проводить автоматическое и ручное тестирование, зная, что приложение будет работать в предсказуемых условиях, а разработчики могут использовать Docker для создания индивидуальных контейнеров для разработки новых функций, исправления ошибок или проведения экспериментов.

**1.5. Теоретические основы. Компоненты**

Экосистема Docker состоит из системных и переменных типов компонентов:

Системные компоненты включают в себя: Docker host, Docker daemon, Docker client и Docker-compose (менеджер запуска кластера контейнеров).

Основные переменные компоненты включают в себя: Dockerfile, Docker image, Docker container.

Также можно отметить git как один из важных и часто используемых инструментов для работы с контейнерами.

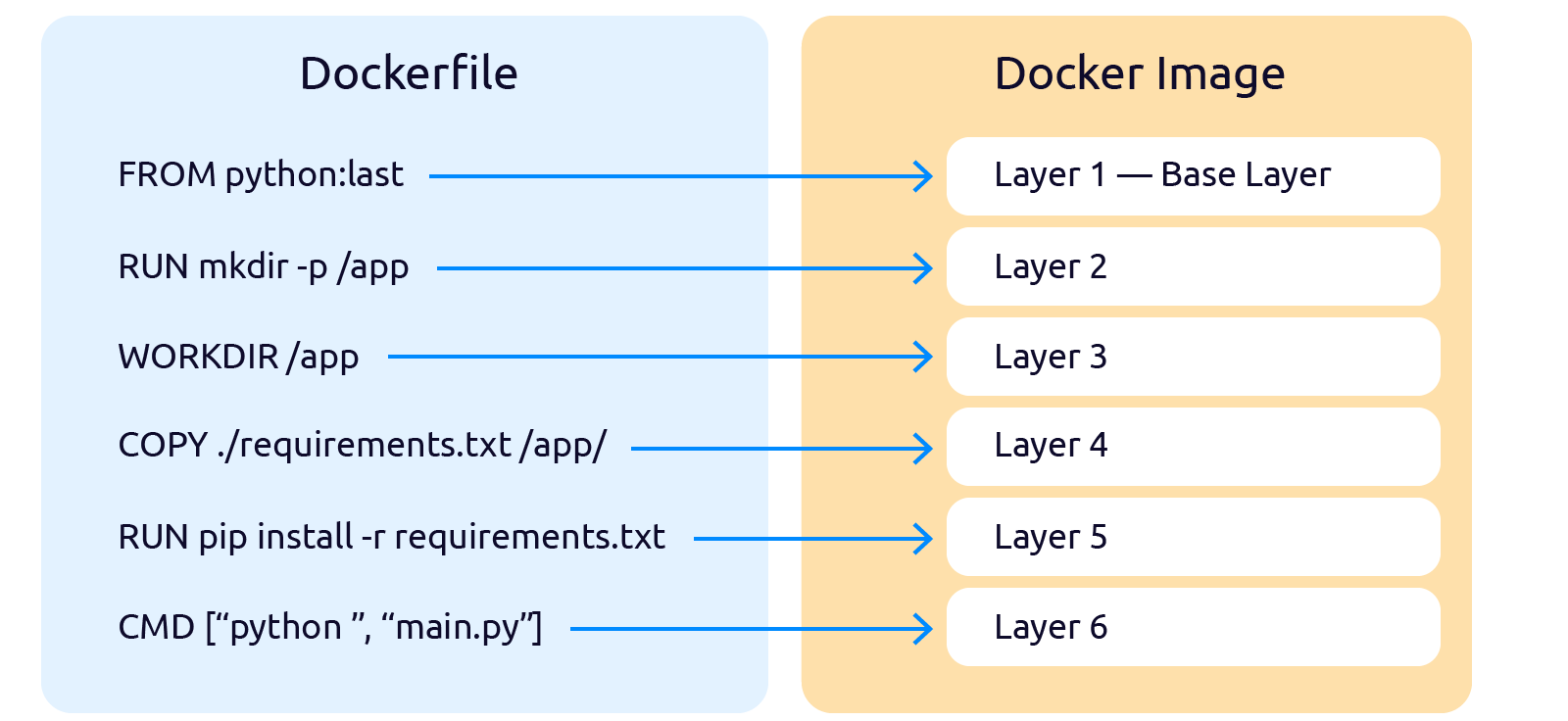
Файл Dockerfile состоит из серии инструкций, которые выполняются в определенном порядке. Некоторые из наиболее распространенных инструкций включают FROM (используется для указания базового образа), RUN (для выполнения команд в контейнере), COPY (для копирования файлов в контейнер), EXPOSE (для указания портов, которые должны быть открыты для внешнего доступа) и ENTRYPOINT/CMD (для определения команды, которая будет запущена при старте контейнера).

При создании нового образа Docker, Dockerfile используется для создания нового контейнера на основе базового образа, который затем может быть дополнительно настроен и оптимизирован для конкретного приложения или задачи. После создания образа Docker, он может быть сохранен в репозитории Docker Hub или другом частном репозитории для последующего использования.



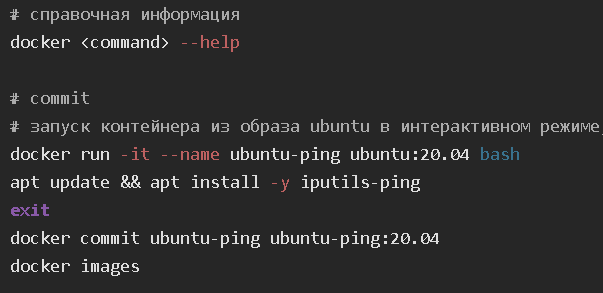
Пример Dockerfile

Образ Docker является важным элементом в последовательности создания контейнера, т.к включает в себя метаданные: название, версия, зависимые пакеты и команды для запуска приложения в контейнере. Когда создается новый образ Docker, он сначала загружается из репозитория DockerHub или другого приватного репозитория и потом запускается в контейнере. По сути образ – это шаблон для создания контейнера. Образы довольно мало весят, поэтому являются очень удобными в плане портативности.



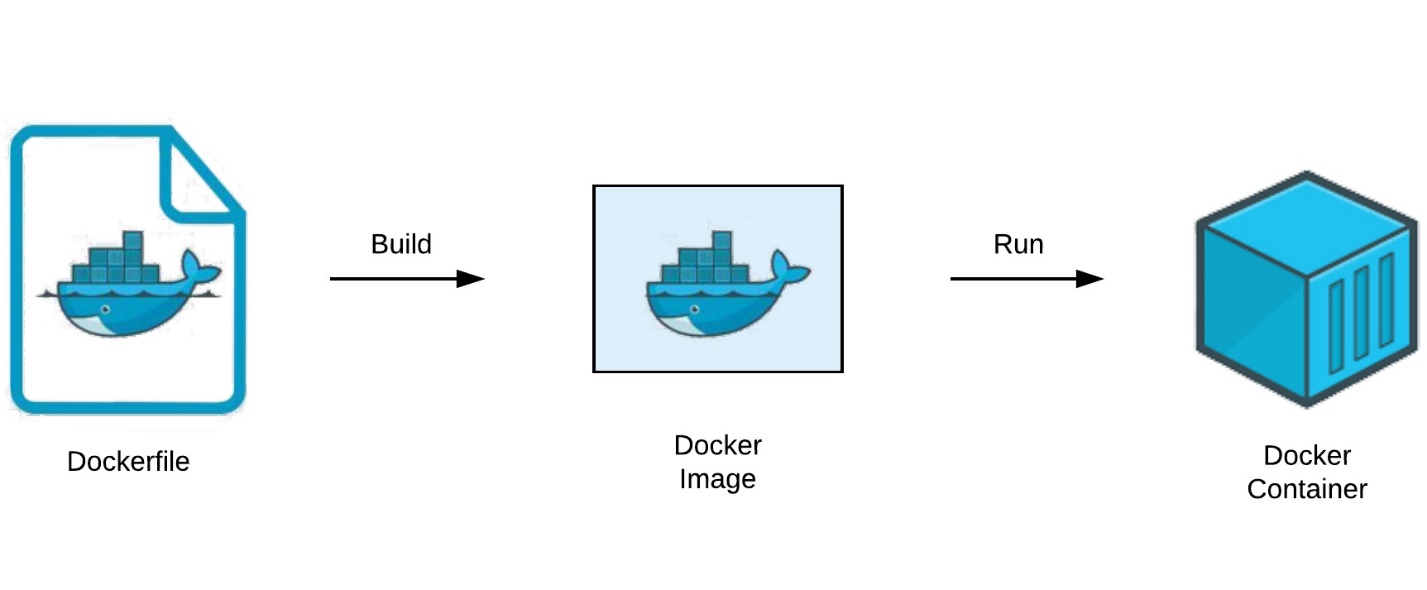
Структура образа

Стоит отметить, что контейнер всегда запускается на ядре linux, поэтому образ можно создать, запустив контейнер из базового образа в интерактивном режиме. На практике способ удобен для небольших быстрых доработок:



Контейнер является последней стадией в процессе контейнеризацции. Каждый контейнер содержит все необходимые зависимости и конфигурации для работы приложения, что делает его независимость от внешней среды, высокую портативность и надежность развертывания.

Контейнеры Docker можно легко клонировать, масштабировать и управлять ими, поэтому, чаще всего, вокруг одного сервиса образуется целая сеть из различных по функционалу контейнеров. Для Таких сетей не редко используются специальные “оркестраторы”, позволяющие оптимизировать процесс управления контейнерами, например: Docker Swarm, Kubernets. Кластерные технологии часто применяются в высоконагруженных сервисах, облачных приложениях.



Контейнеризация.

git - неотъемлемая часть рабочего процесса разработки с Docker. Он обеспечивает версионирование, воспроизводимость, совместную работу, автоматизацию и аудит, что делает процесс создания и развертывания контейнеризированных приложений более эффективным и надежным.

git не является заменой Docker, а скорее является важным дополнением, обеспечивающим контроль над кодовой базой и процессом сборки образов.

Некоторые из возможностей совмещения git и Docker:

* Триггеры сборки Docker: Многие системы используют git для отслеживания изменений в репозитории. При появлении новых коммитов можно автоматически запускать сборку Docker-образа и его развертывание
* Интеграция с реестрами Docker: git может быть интегрирован с реестрами Docker для автоматической отправки собранных образов после успешной сборки.
* Возможность отката: Если что-то пошло не так, вы всегда можете откатиться к предыдущей версии кода или Dockerfile с помощью git.

**1.6. Отличие контейнера от виртуальной машины**

Многие люди совершенно не видят разницы между контейнером и виртуальной машиной. Это глубокое заблуждение.

Виртуальные машины и контейнеры немного схожи, но представляют собой два различных метода создания изолированных сред для работы приложений. Оба подхода обеспечивают независимость запуска приложений, но реализуют это по-разному:

Виртуальная машина формирует окружение, эмулирующее физическое оборудование. Каждая такая машина обладает своей операционной системой, функционирующей на основе гипервизора, который управляет виртуальными машинами и распределяет ресурсы между ними.

Контейнер же, вместо эмуляции физического оборудования, задействует уже имеющуюся ОС и добавляет слой абстракции для независимого запуска приложений.

Ключевые отличия:

1. Производительность: виртуальные машины создают полные копии оборудования, что требует больше ресурсов и может замедлить работу. Контейнеры, используя общую ОС, работают быстрее, не требуя дополнительных ресурсов для изоляции.
2. Масштабируемость: виртуальные машины масштабируются тяжело и затратно. Контейнеры проще масштабируются горизонтально благодаря использованию общих ресурсов.
3. Управление: каждую виртуальную машину нужно управлять отдельно, тогда как контейнеры управляются общим менеджером на хосте. В целом, выбор между виртуальными машинами и контейнерами почти всегда смещается в сторону контейнеров.

**ГЛАВА 2**

В этой главе будет рассматриваться код ботов и способ деплоя их на веб-хостинг Back4App. По сути, докер в этой ситуации даёт возможность пользоваться тем фактом, что любая платформа для хостинга – это сервер, благодаря чему получится запустить там приложение, которое не предназначенное для этого хостинга. (почему я выбрал именно этот пример? Что делать с этим вопросом?) Это очень удобно для небольших приложений, по типу пет-проектов или ботов, потому что очень мало где можно найти бесплатный хостинг для, в теории, нагруженных приложений, а схожих хостингов с Back4App гораздо больше. Также деплой на сервер является одним из самых простых примеров работы с Docker

**2.1. Конструирование ботов, подготовка к деплою.**

Для начала нужно составить иерархическую структуру папок:

* bot.py (Код с ботом)
* requirements.txt (Файл с зависимостями для ботов)
* .env (Файл с токеном (ключом активатором) для бота )
* .dockerignore (То, что нужно игнорировать Docker)
* .gitignore (То, что нужно игнорировать git)
* Dockerfile (Файл, конструирующий контейнер)
* /Cogs/ (Папка с когами)

**Telegram bot**

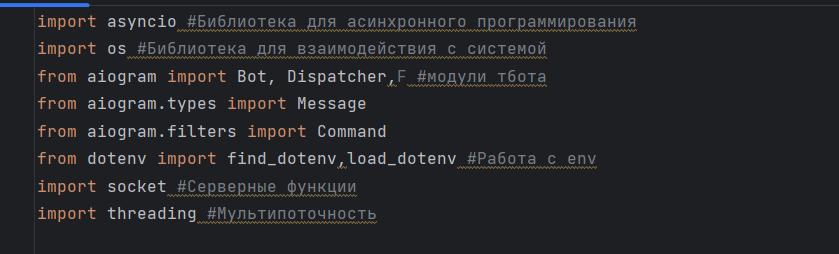


Рисунок 1

Для начала импортируем нужные нам библиотеки:

Asyncio нужно для того, чтобы части программы могли выполнятся параллельно

В данном случае библиотека os используется для извлечения токена(ключа-активатора бота)

Aiogramm нужен для взаимодействия с API телеграмма(взаимодействия с ботом)

Dotenv нужен для более удобной работы с env файлами

Socket используется для создания и управления сервером

Threading нужен для разделения работы разных программ/кусков кода, чтобы они не мешали друг другу

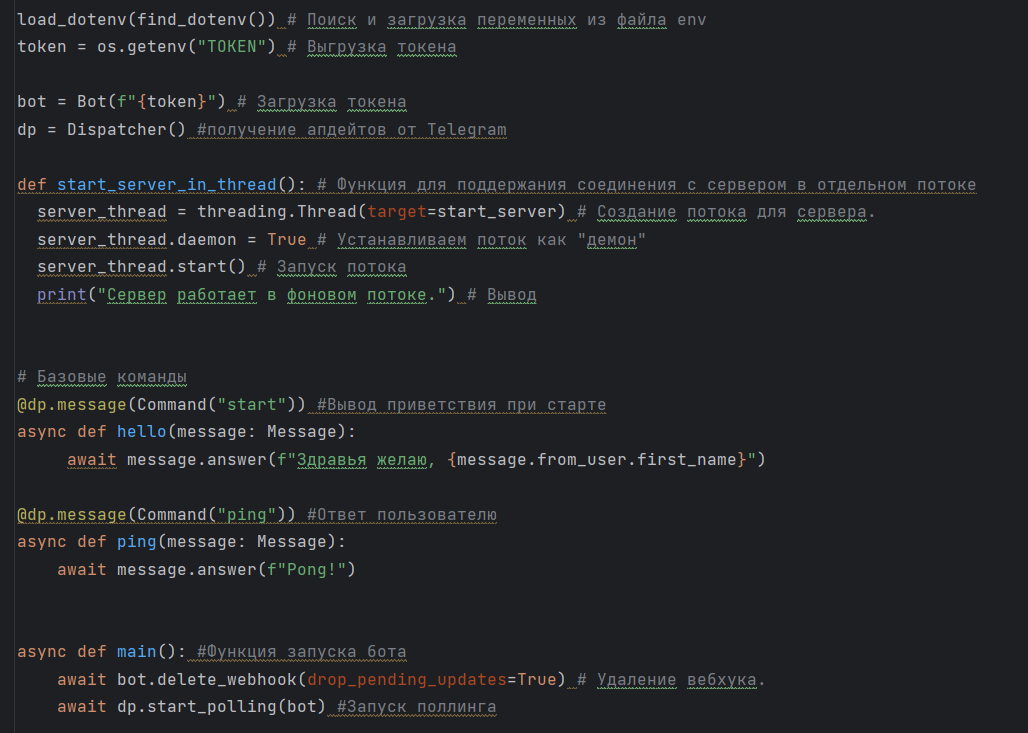


Рисунок 2

Далее идёт код, в котором реализуется: загрузка токена в переменную; функции запуска сервера в отдельном потоке и запуска бота; две базовые команды для проверки работоспособности бота

Важно подметить, что:

Переменные в файл .env записываются в формате: name = value. В этом случае переменная будет выглядить так: TOKEN = “1234567:tokenexample”

Сервер обязательно должен находится в фоне и существовать параллельно с самим ботом, так как иначе не получится подключиться к хостингу

Установка потока как демон означает, что он выполняет фоновые задачи и не блокирует завершение программы

Удаление вебхука нужно, чтобы под получал апдейты через поллинг(периодические запросы на сервер). В данном случае лучше всего использовать именно его, т.к. метод получения апдейтов через вебхук требует полноценный сервер и гибкой настройки, поэтому удобнее всего будет использовать поллинг.

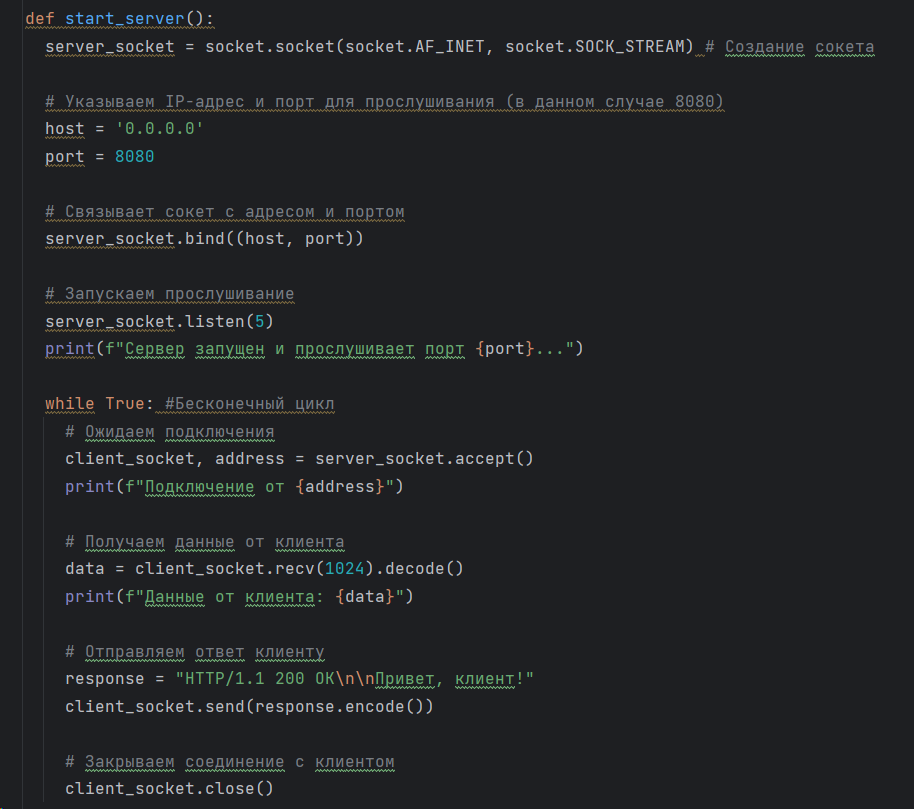


Рисунок 3

В этой части кода реализуются серверные функции.

В контексте проекта сервер нужен из-за требований хостинга, который используется в проекте. Так как площадка для деплоя предназначена для веб-хостинга, то контейнер станет уходить в спящее состояние, если не будет запросов к серверу. Код сделан так, чтобы запросы отправлялись автоматически на сервер, поддерживая контейнер в активном состоянии

Сокет нужен для установки соединения в сети. Сокет создается с использованием IPv4 (AF\_INET) и TCP-соединением (SOCK\_STREAM). Это самый частый способ создания сокета для сетевого взаимодействия.

Host 0.0.0.0 значит, что сервер принимает подключение со всех ip. Значение переменной port играет не сильное значение: главное, чтобы оно не мешало другим соединениям. Обычно порт 8080 берут для разработки, но можно использовать порты в пределах 1024—49151 (лучше брать выше 4096)

Сервер работает в бесконечном цикле, чтобы постоянно принимать новые подключения. Это стандартная практика для серверов, которые должны работать непрерывно.

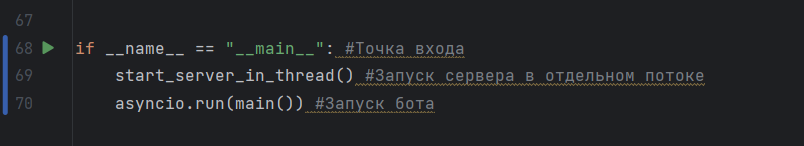


Рисунок 4

Завершается код конструкцией под названием точка входа.

Конструкция if \_\_name\_\_ == “\_\_main\_\_” является точкой входа. Простыми словами – это место, откуда начинается программа.



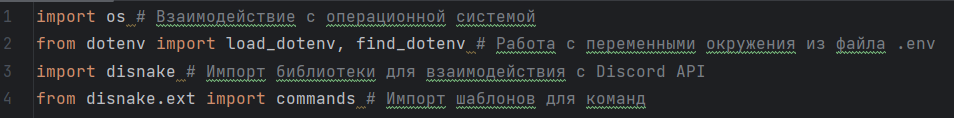
Рисунок 5

Перейдём к файлу requirements.txt

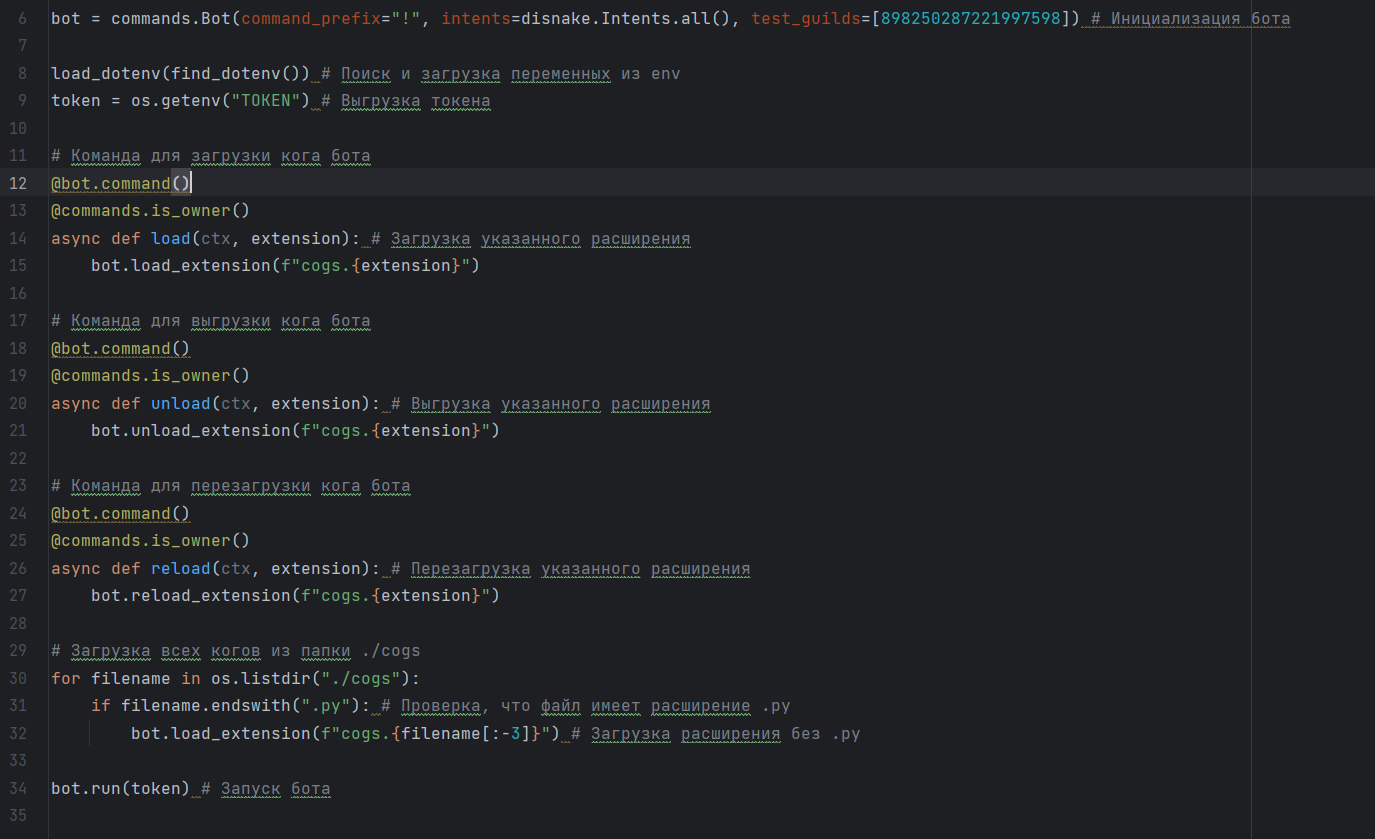
В этом файле просто нужно указать нестандартные библиотеки, которые вы использовали в своём коде в формате: название библиотеки == версия. Здесь записаны только две библиотеки, так как остальные являются стандартными и идут при установке python. Чтобы узнать версию вашей библиотеки вы можете зайти на pypi.org или узнать с помощью консольной команды pip show.

**Discord bot**

Так как способы реализации телеграм и дискорд бота немного отличаются, то структура папок и файлов у дискорд бота будет немного отличаться(см. диаграмма 1).



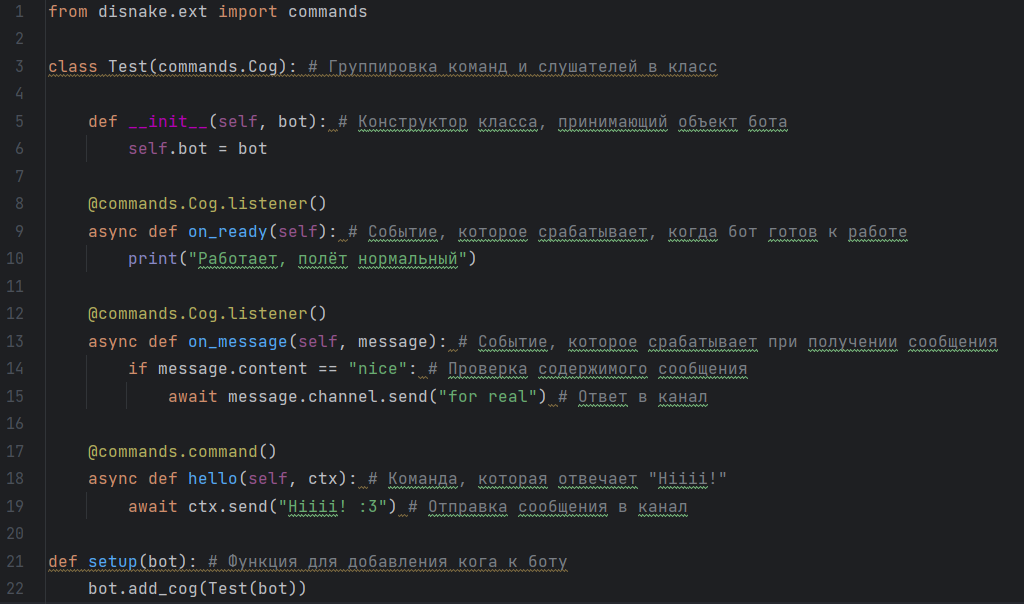
Для начала также идёт импорт библиотек и модулей.



Далее идёт основной код.

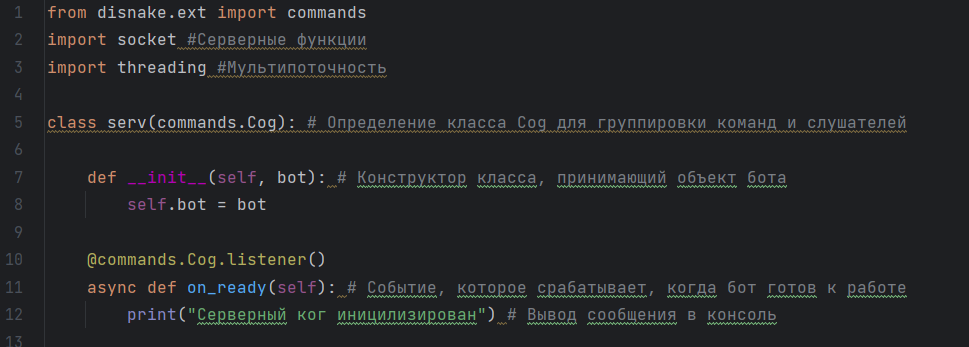
Различие между конструкцией телеграм бота и дискорд бота в этом случае заключается в разграничении кода на разные файлы: в случае дискорд бота исполняемый файл выступает "основой" которою дополняют коги (добавочные файлы), поэтому код главного файла реализует только функции взаимодействия с когами и запуска бота

Также в коде дискорд бота можно часто заметить декораторы – шаблоны кода для упрощения взаимодействия с API. В коде они помечаются жёлтым.

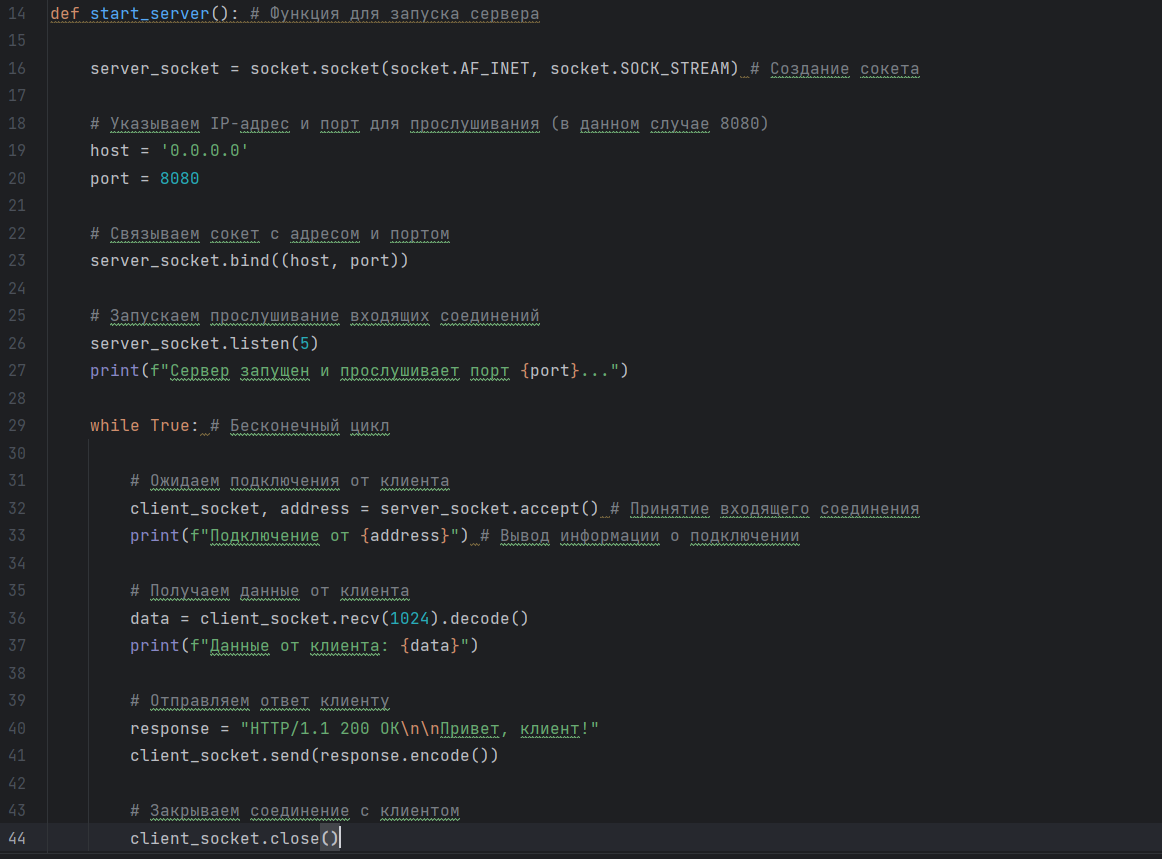


Этот ког нужен для проверки работоспособности бота. В нём создаются базовые команды, по типу ответа на сообщения или вывода в консоль при запуске.

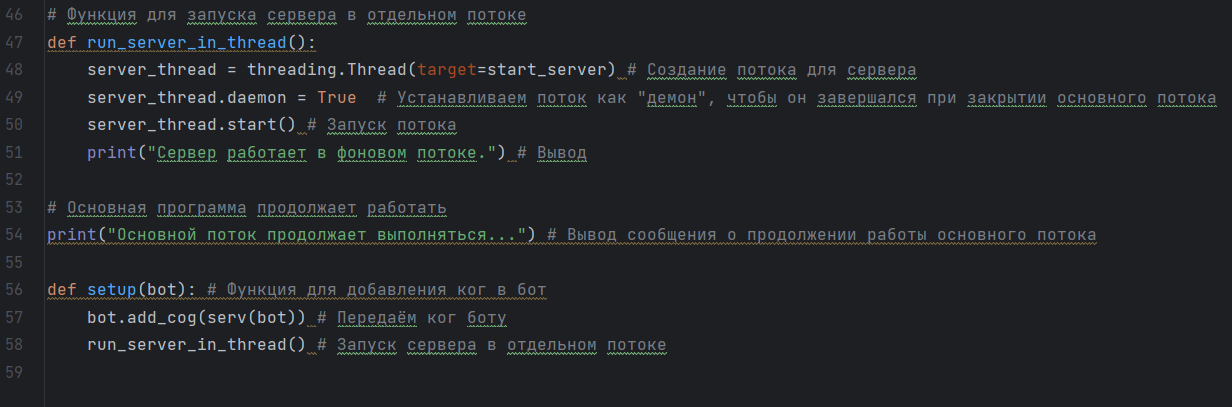
Группировка в класс используется для объединения всего кода в 1 ког, чтобы его можно было "присоединить" к основному файлу.



Завершающим будет серверный ког. Тут всё просто, сначала создаём класс.



Затем создаём сервер, такой же, как и у телеграмм бота.



И запускаем это всё.

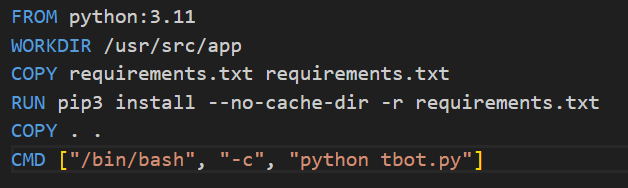


Requirements.txt тоже сильно не отличается от зависимостей телеграм бота.

**2.2 Создание и локальное тестирование контейнеров**

Ключевая часть проекта - конструирование Dockerfile и последующее создание docker image, docker container. Из первой части проекта стало понятно, что образ представляет собой исполняемый пакет, содержащий всё необходимое для работы приложения: код, зависимости, системные библиотеки и настройки. Он строится по слоям, каждый из которых соответствует инструкции в Dockerfile. Вспомним, как выглядит структура Dockerfile и образа (см. (тут название и номер рисунка по слоям))

Следовательно, для успешного деплоя бота нужно создать Dockerfile, включающий в себя все нужные зависимости и, соответственно, файлы бота.



Команда **FROM** копирует официальный образ Python версии **3.11** с общего хранилища образов DockerHub и делает его базовым.

Если бот не требует сложных зависимостей, можно использовать python:3.11-slim или python:3.11-alpine - более легковесные варианты обычного python.

**WORKDIR** устанавливает рабочую директорию в unix системе(образы обычно используют linux). В рабочей директории будут выполняться все последующие операции.

**COPY** – команда для копирования файлов. Copy requirements.txt копирует файл с зависимостями **из локальной папки проекта** в рабочую директорию контейнера, чтобы его можно было сразу установить командой RUN

**RUN** запускает установщик пакетов python pip, а после pip находит в рабочей папке зависимости и устанавливает их в системную область python в контейнере (не в рабочую папку)

**COPY . . означает копирование всех файлов из текущей директории, где находится Dockerfile, за исклочением тех, что прописаны в Dockerignore.**

**Важно подметить, что сначала копируются зависимости, а потом вообще все файлы. Это связано с оптимизацией под кеширование слоёв (см тут ссылка на фотку со слоями из первой части).** Если код изменился, но зависимости остались прежними, Docker переиспользует закешированные слои с зависимостями и долгая установка пакетов не повторяется.

Флаг --no-cache-dir удаляет временные файлы после установки, уменьшая размер образа.

**CMD в Dockerfile указывает команду и аргументы для выполнения внутри контейнера.** CMD на весь Dockerfile может быть только один, так как он завершает построение образа

В данном Dockerfile CMD запускает командную строку linux по пути /bin/bash, которая выполняет следующую за флагом -c строку команд

Обычно, при составлении Dockerfile, bash не использует, но чтобы программа корректно работала на back4app, лучше им воспользоваться.

Также при составлении Dockerfile могут пригодиться команды:

**LABEL** — описывает метаданные, например, сведения о том, кто создал и поддерживает образ.

**ENV** — устанавливает переменные среды в формате «ключ = значение». Эти переменные доступны внутри контейнера скриптам и различным приложениям. Благодаря этой команде можно, например, добавить токен в контейнер, если не хотите делать это файлом

**ADD** —может копировать распаковывать некоторые локальные файлы в контейнер.

**ARG** — задаёт переменные для передачи Docker во время сборки образа.

**ENTRYPOINT** — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются.

**EXPOSE** — открывает доступ контейнеру к определённому порту. Может пригодится если нужно воспользоваться сервером.

Dockerfile у телеграм бота и дискорд бота будет отличаться только названием запускаемого файла, потому что принципы создания контейнера для них одинаковы.

Для лучшей оптимизации кода предпочтительно создать .dockerignore, в котором будут указаны все ненужные файлы, которые не должны попасть в контейнер, например кеш от запуска бота:



Для схожих целей потребуется .gitignore. Туда следует указать те же файлы и папки, добавив .env, т.к. если репозиторий будет публичным, то все смогут воспользоваться токеном бота и управлять им

Теперь можно перейти к созданию image. Это можно сделать несколькими способами, но самым удобным способом является сборка в консоли и последующее тестирования с помощью официального приложения Docker Desktop.

Для создания образа нужно зайти в консоль, и перейти в рабочую директорию(место, где находится бот) командой cd /путь к папке.

Далее прописываем:



Docker build – обращение к докеру, чтобы он собрал образ

-t – указывает, что следующее слово будет именем

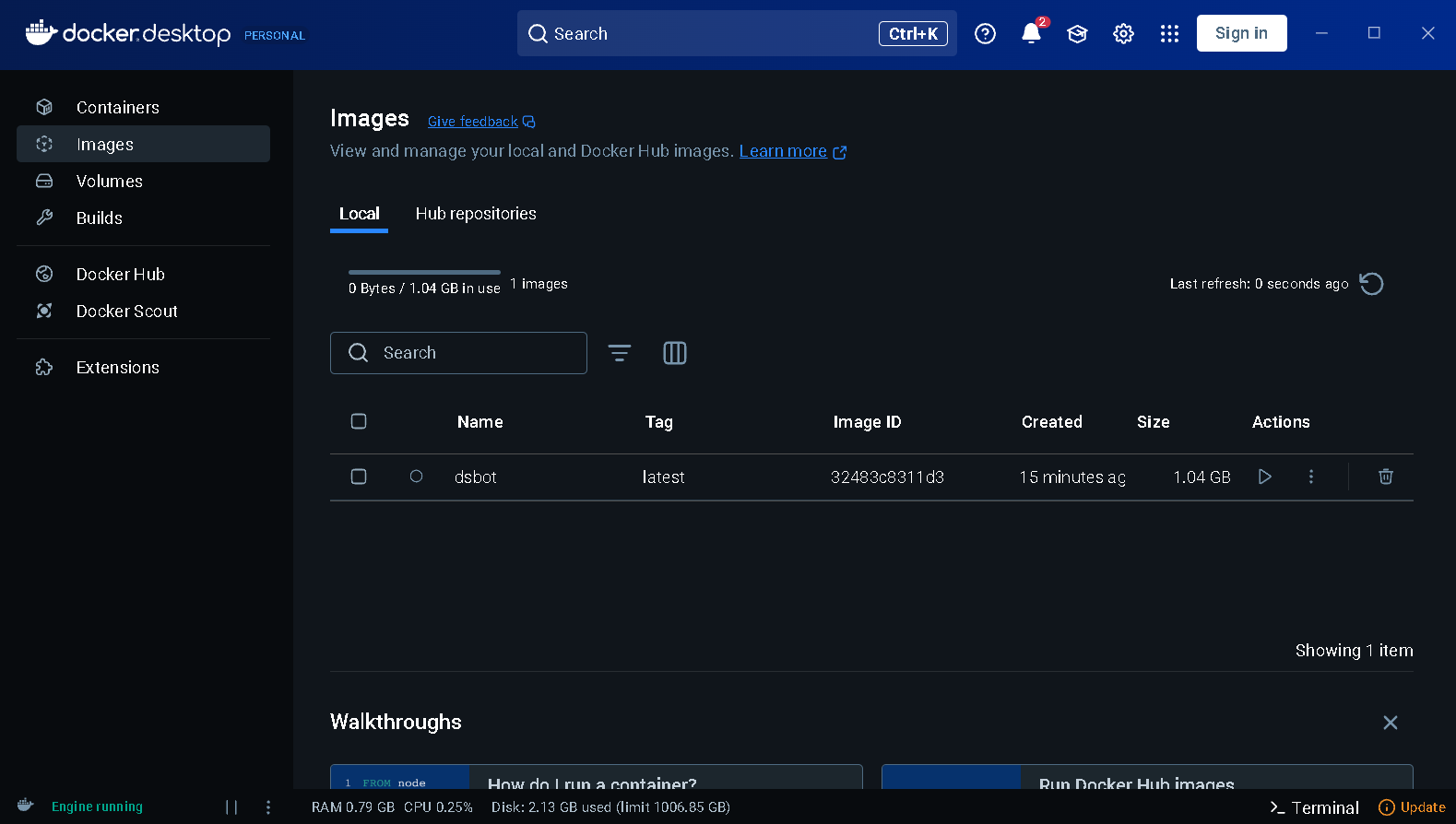
Завершающая точка указывает на местонахождение Dockerfile, то есть та же директория, что и открыта в консоли.

После запуска у вас должно высветится подобное сообщение:

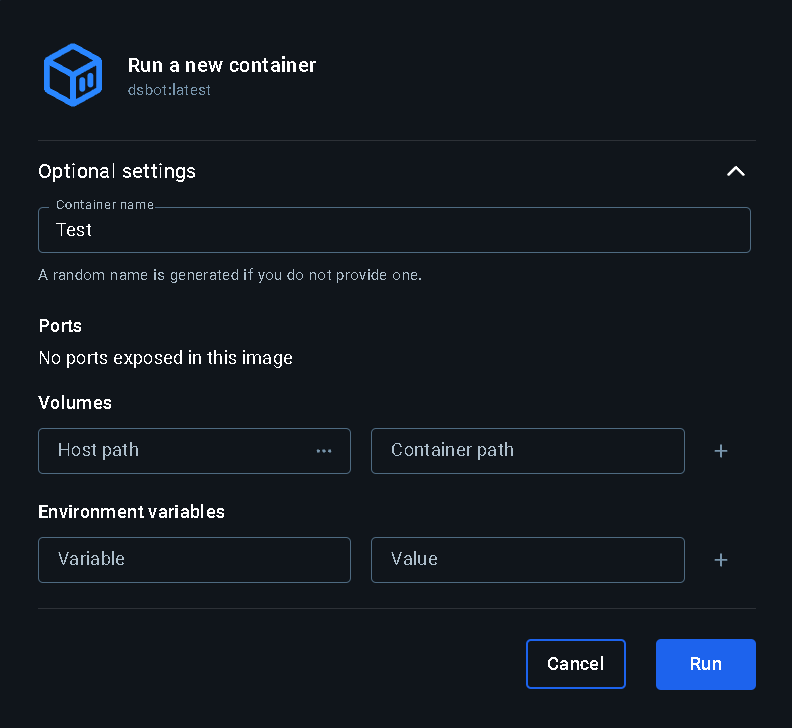


Это означает, что сборка образа прошла успешно.

Далее, предварительно скачав, нужно открыть Docker Desktop, где во вкладке images будет ждать собранный образ.

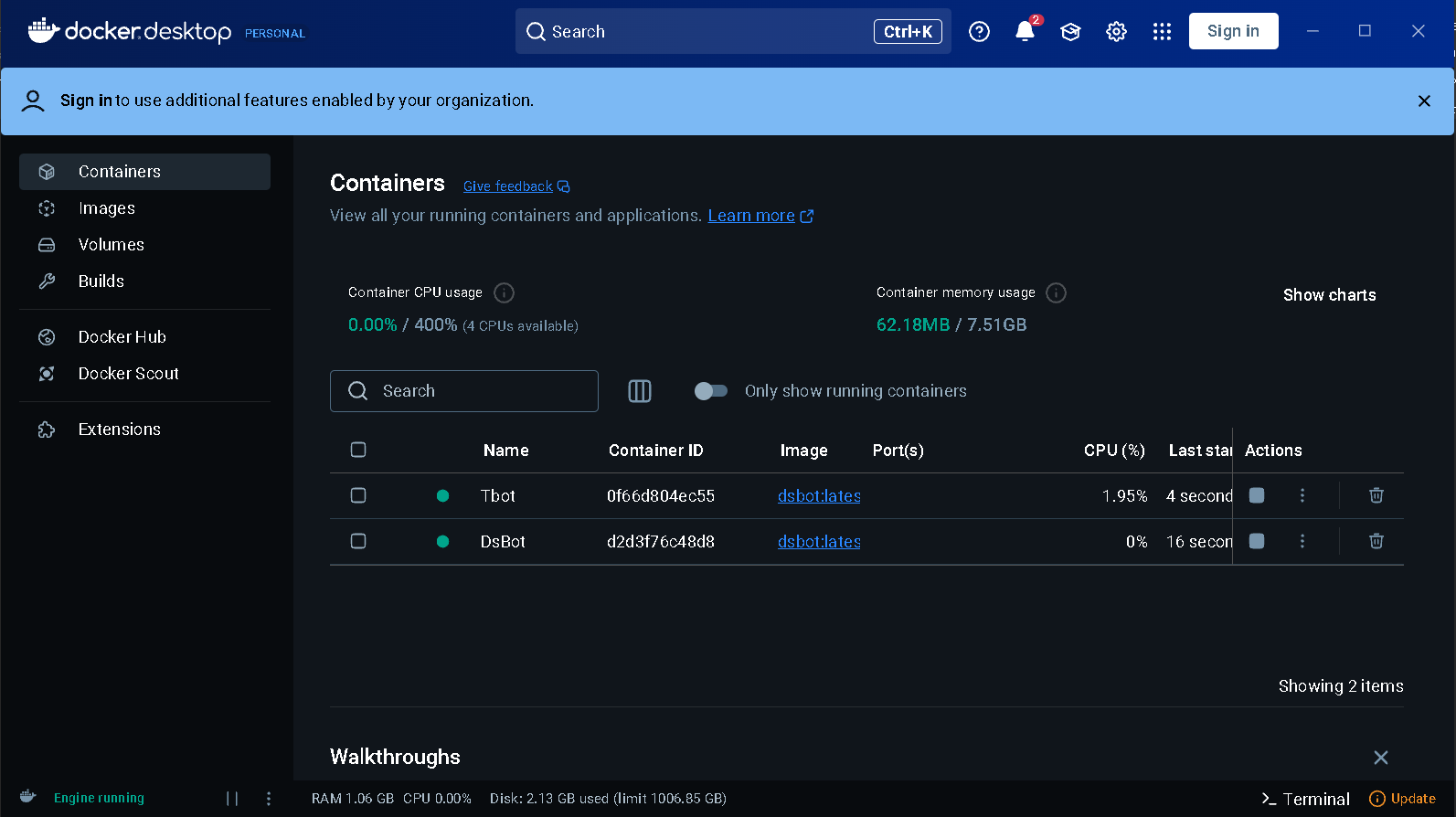


В столбце Actions у образа будет кнопка Run. При нажатии на неё будет открыто меню настроек контейнера. Там указываем можно указать **постоянное хранилище для контейнеров(Volumes) и переменные среды(Environment variables), открыть порты.**



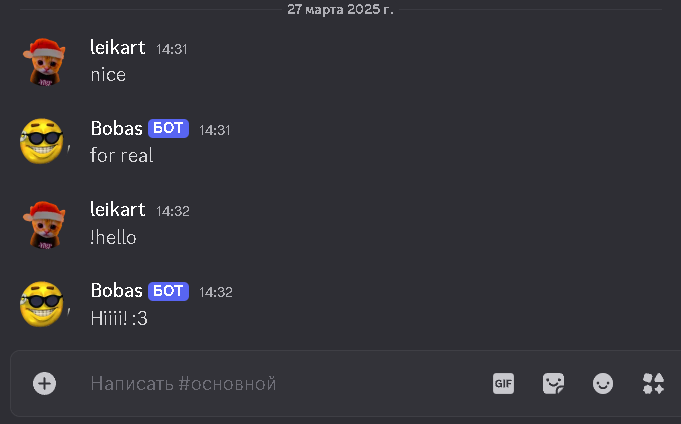
**Указываем всё нужное и нажимаем Run.**

**Всё тоже самое можно проделать с другим ботом.**

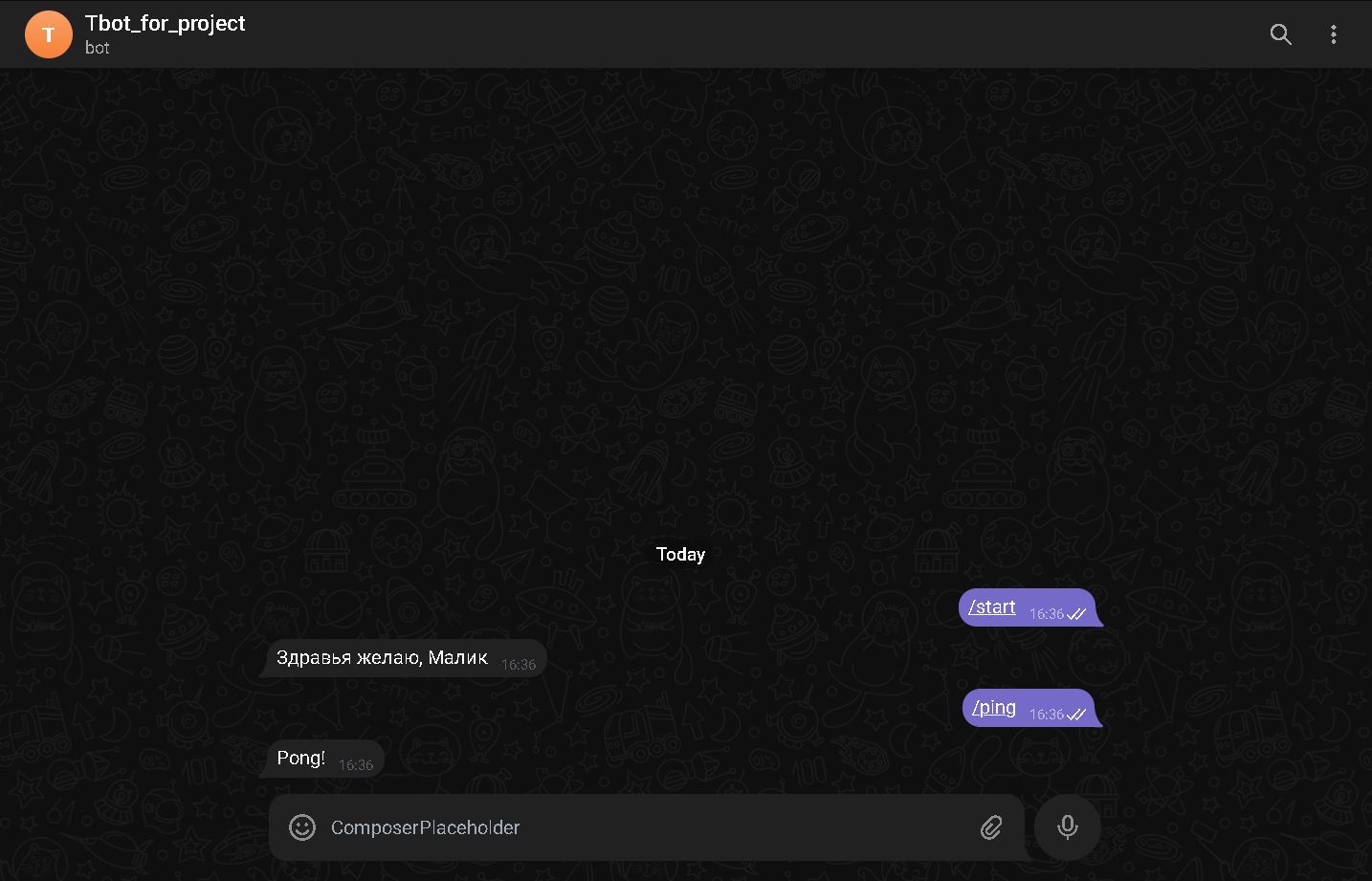


**В итоге получается конструкция из двух образов и двух контейнеров, созданных по образам. По образам можно создавать сколько угодно контейнеров.**

**Теперь можно проверить работоспособность ботов:**



**Команды со скрина (тут надо указать)**



**Реализуются команды со скрина (номер)**

**2.3 Развертывание приложений, проверка работоспособности.**

Теперь можно перейти к заключительной части проекта. Для этого понадобится установленный git и аккаунт на github.

Back4app деплоит проекты прямиком из репозитория github, т.к. он является крупнейшей площадкой для разработчиков. Github обеспечивает версионирование, простоту в совместной разработке, поддерживает непрерывное развёртывание.

Зарегистрировавшись, на главной странице нажимаем new, называем репозиторий и указываем, кому он будет доступен(private или public). После проделанных действий откроется созданный репозиторий, где будут команды по залитию файлов в репозиторий.

Чтобы загрузить бота в репозиторий, в папке с о всем содержимым нужно открыть git bash (ПКМ по папке 🡪 git bash) и ввести следующие комманды:

git init – инициализация гита в папке

git add . – добавление всех файлов в текущей директории, за исключением тех, что указаны в .gitignore

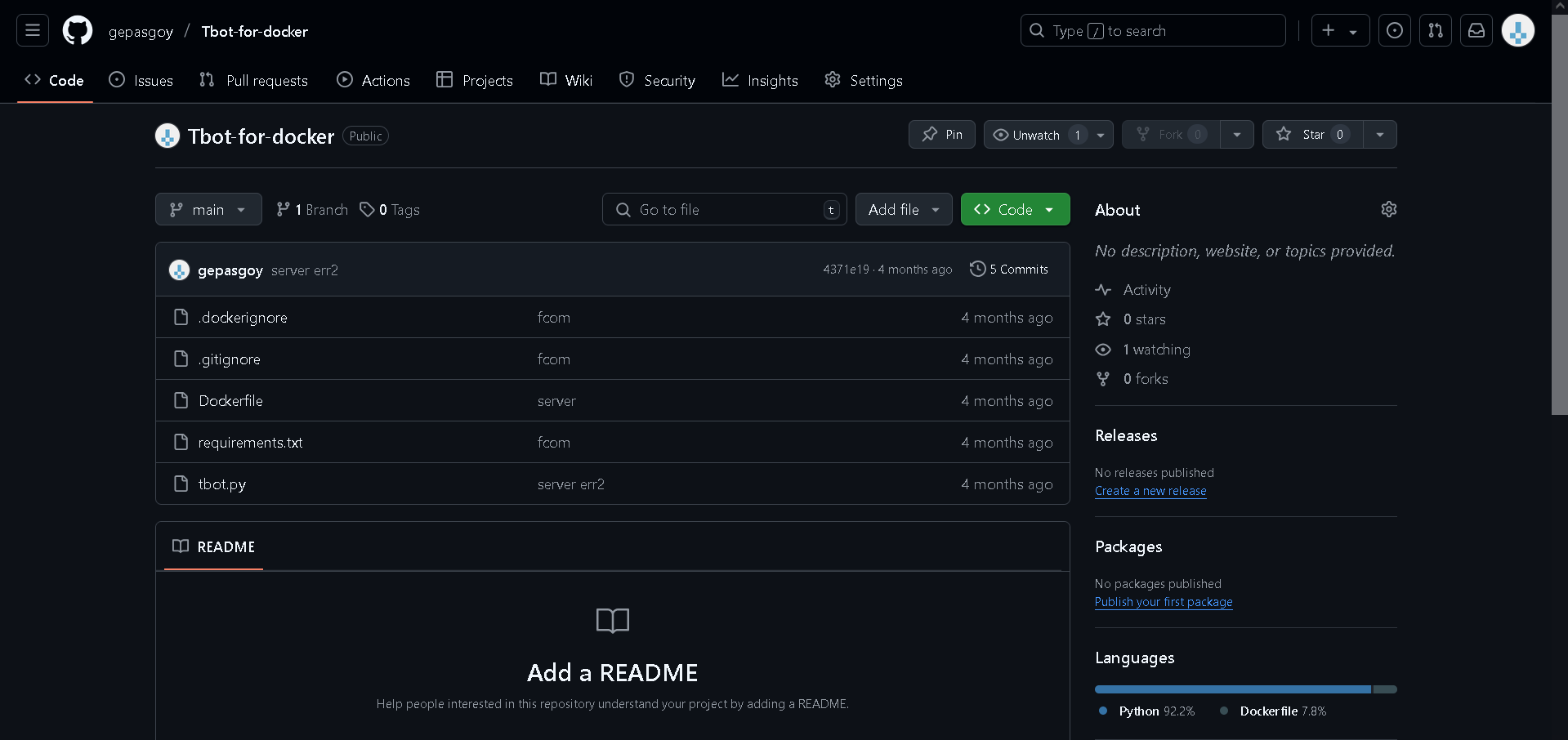
git status(опционально) – указывает, какие файлы будут закоммичены

git commit -m "firstcom" – сохраняет то, что будет отправлено в репозиторий. Флаг -m задаёт имя коммиту

git branch -M main – устанавливает главной веткой main(в репозиториях git есть система веток, которая позволяет работать каждому независимо от другого).

git remote add origin https://github.com/gepasgoy/Tbot-for-docker (ссылка на репозиторий) - добавление удалённого репозитория с именем origin, указывая на репозиторий на github по ссылке

git push -u origin main – отправляет (пушит) коммиты в удаленный репозиторий origin. Флаг -u устанавливает ветку как "отслеживаемую", что упрощает последующее отправление (git push) и сбор коммитов (git pull), т.к. можно будет писать команды без указания ветки.

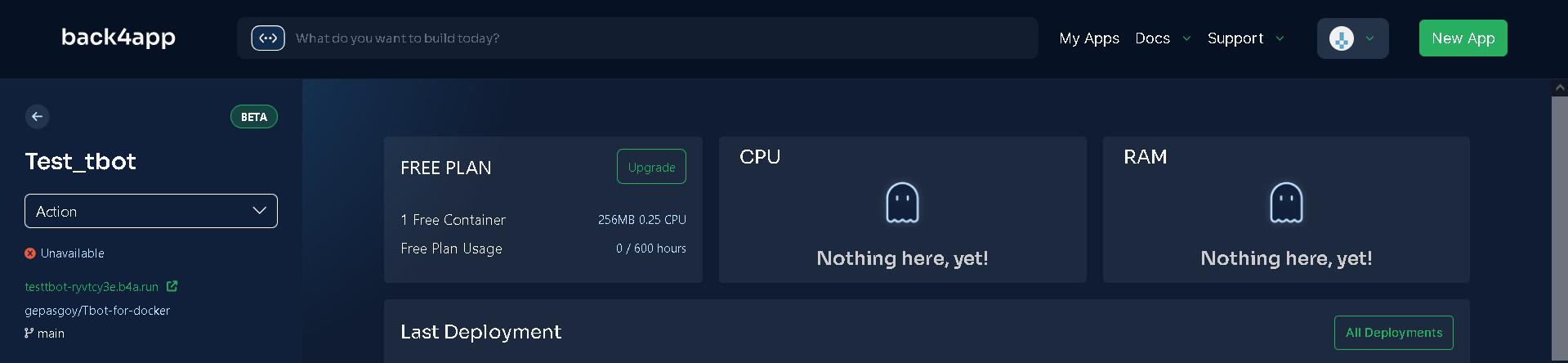


При успешном пуше должна получится похожая файловая структура.

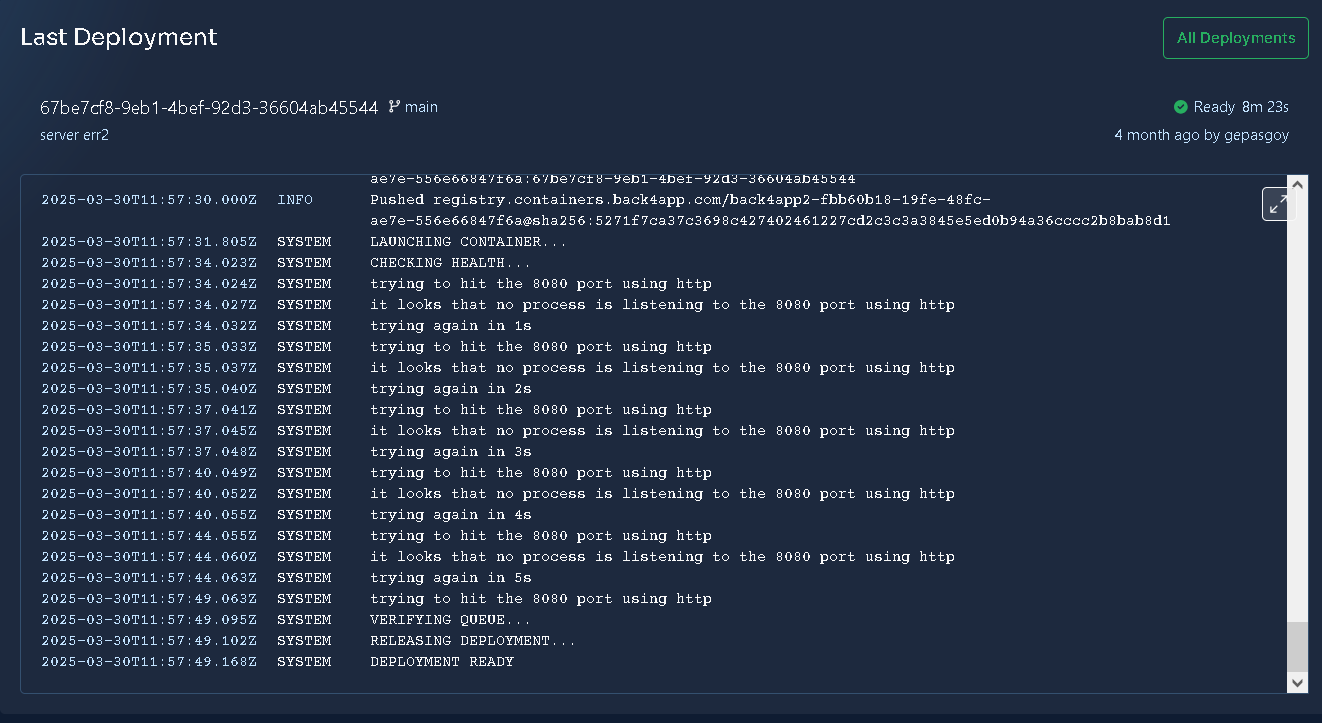
Для деплоя можно выбрать любую платформу, которая поддерживает загрузку с репозиториев, но я буду использовать Back4App, т.к. Она предоставляет бесплатно 600 часов пользования, хорошие характеристики сервера, обширную документацию и возможность загружать свои базы данных, настраивать их взаимодействие с захосченными приложениями.

Приложения на back4app деплоится так: загрузка с репозитория 🡪 сборка образа 🡪 запуск контейнера 🡪 проверка взаимодействия с сервером 🡪 вывод об успешном деплое

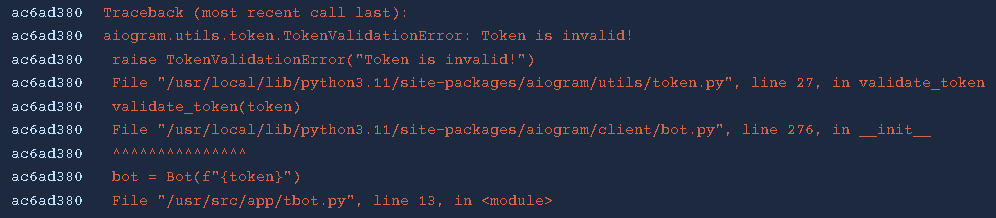
Регистрируемся на Back4App, нажимаем Dashboard 🡪 Web Deployment Platform 🡪 Deploy a Web App. Тут нужно указать репозиторий для деплоя. Откроется страница с настройками приложения. Тут даём имя приложению, указываем переменные среды(токен) и оставляем или отключаем autodeploy. Остальные настройки лучше оставить по умолчанию. После этих манипуляций откроется dashboard, где будет вся нужная информация по приложению:



Перед тем, как запустить создание контейнера, нужно зайти в настройки и указать порт сервера. Т.к. сервер создавался с заготовкой под порт 8080, то указываем его. После этого можно запускать.



Запуск прошёл успешно, если последней строкой будет выведено DEPLOYMENT READY. Если нет, то ниже окна Last Deployment есть логи, в них будут написаны все ошибки, например:



Эти логи указывают на ошибку в токене бота.

Также, весь вывод принтом будет выводится в дашборде:

**СПИСОК ЛИТЕРАТУРЫ**

**Книги**

### *Парминдер Сингх Кочер*, Микросервисы и контейнеры Docker, ДМК Пресс, год, 237 страниц

### *Моут Эдриен*, Использование Docker, ДМК Пресс, год, 346 страниц

### *Иан Милл,Эйдан Хобсон Сейерс*; Docker на практике, ДМК Пресс, год, 511 страниц

### Сайты

### Контейнеры в (IT), <https://www.atlex.ru/baza-znanij/tematicheskij-glossarij/kontejnery-v-it/>, 15.11.24

### Real world container use(актуальная статистика), [https://www.datadoghq.com/container-report](https://www.datadoghq.com/container-report/), 09.09.2024

### Как устроен Docker и почему он популярен, <https://yandex.cloud/ru/blog/posts/2022/03/docker-containers>, 09.09.2024

### Какие известные компании используют Docker и для чего, <https://habr.com/ru/companies/flant/articles/326784/>, 12.09.2024

### Основы контейнеризации (обзор Docker и Podman), <https://habr.com/ru/articles/659049/>, 13.12.2024

### 10 лет Docker: от революционной идеи до современной практики, <https://habr.com/ru/companies/yandex_praktikum/articles/760076/>, 08.12.2024

### Изучаем Docker, часть 3: файлы Dockerfile, <https://habr.com/ru/companies/ruvds/articles/439980/>, 23.12.2024

### Путеводитель по Docker. От основ контейнеризации до создания собственного докера, <https://habr.com/ru/articles/810777/>, 23,12,2024