

Ссылка на видео с кратким объяснением что к чему, на **YouTube**:

Всем привет 🙌

Это небольшие методические указания о том, как можно создать мессенджер используя Python и Docker.

Мы поступим проще, будем использовать свой компьютер как сервер. И реализуем нужный нам функционал.

НО

Если сделать следующие шаги, то можно полноценно увидеть online работу:

Для работы сервера и реализации переписки пользователей вам понадобятся следующие элементы:

1) Доменное имя (Domain Name): Доменное имя - это адрес вашего веб-сервера, по которому пользователи смогут получить доступ к вашему приложению. Например, "example.com".

2) Хостинг (Hosting): Хостинг предоставляет инфраструктуру для размещения вашего серверного приложения в сети Интернет. Вы можете выбрать облачный хостинг или использовать собственные физические серверы.

3) Веб-сервер (Web Server): Веб-сервер отвечает за обработку и доставку клиентского запроса, а также отправку ответа обратно. Некоторые популярные веб-серверы включают в себя Apache, Nginx и Microsoft IIS.

4) База данных (Database): Для хранения информации о пользователях и их переписке вам понадобится база данных. Вы можете выбрать различные СУБД (системы управления базами данных), такие как MySQL, PostgreSQL или MongoDB.

5) Платформа программирования: Вам нужно выбрать платформу программирования для реализации серверной логики. Некоторые популярные языки программирования для создания серверных приложений включают PHP, Node.js, Python, Ruby и Java.

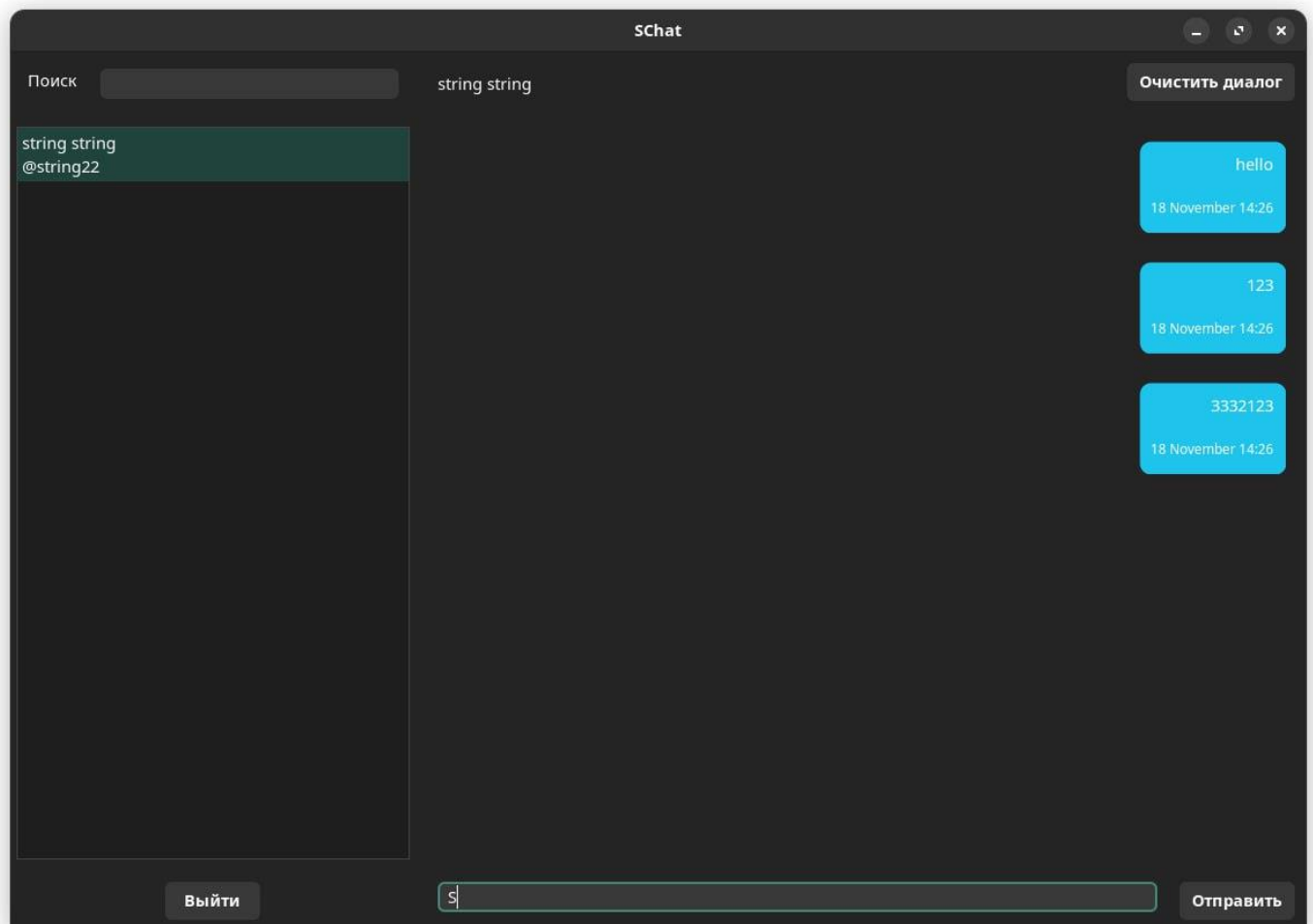
Чтобы реализовать переписку пользователей, вам нужно создать и настроить серверное приложение, которое будет обрабатывать запросы от клиентов, сохранять сообщения в базе данных и обеспечивать их передачу между пользователями.

Необходимыми шагами для реализации такого приложения будут:

- 1) Разработка серверной логики для обработки запросов и отправки ответов.
- 2) Настройка базы данных для хранения сообщений и информации о пользователях.
- 3) Создание API (интерфейса программирования приложений) для обмена данными между клиентами и сервером.
- 4) Создание пользовательского интерфейса на стороне клиента (например, веб-страницы или мобильного приложения), который будет взаимодействовать с сервером через API.

Процесс разработки серверного приложения весьма сложен, и требует знаний в области программирования, баз данных и сетевых технологий. Рекомендуется обратиться к специалисту или команде разработчиков, чтобы получить конкретную помощь и поддержку в создании вашего проекта переписки пользователей.

Для начала предлагаю взглянуть на предполагаемый дизайн приложения:

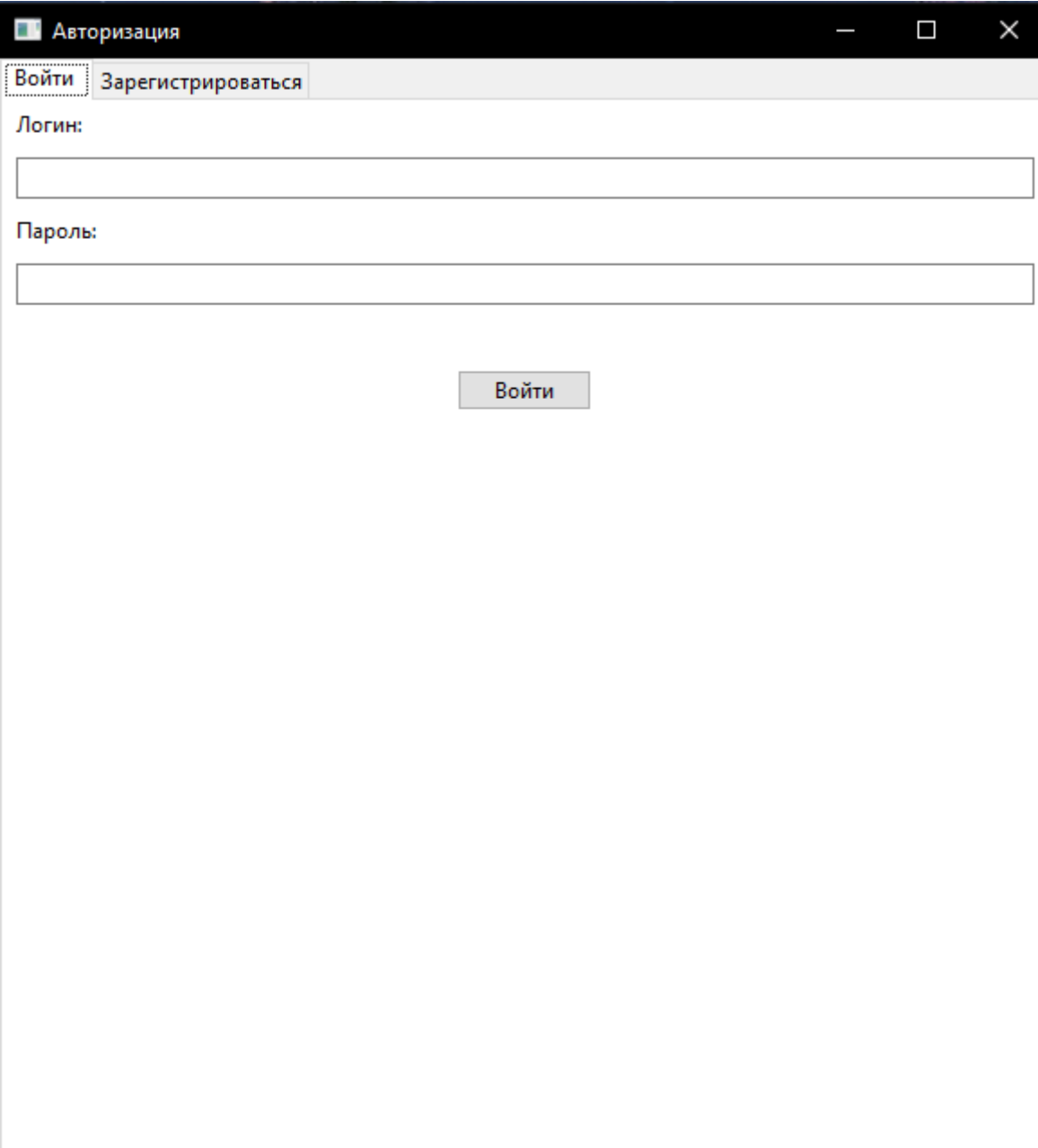


Как мы можем заметить у нас имеется следующий функционал:

- поиск пользователей;
- кнопка очистки диалога у всех пользователей;
- поле для ввода сообщения;
- кнопка отправки сообщения;
- кнопка выхода из приложения;

Рассмотрим, как работает программа

Когда у нас запускается приложение, то пользователь видит перед собой окно авторизации, где он может ввести логин и пароль и выполнить вход в приложение или же просто зарегистрироваться, а затем выполнить вход.



Авторизация

Войти Зарегистрироваться

Логин:

Пароль:

Войти

Регистрация нового пользователя.

Авторизация

Войти

Зарегистрироваться

Имя:

Костос

Фамилия:

КостосХ

Имя пользователя:

КостосХ

Почта:

КостосХ@mail.ru

Пароль:

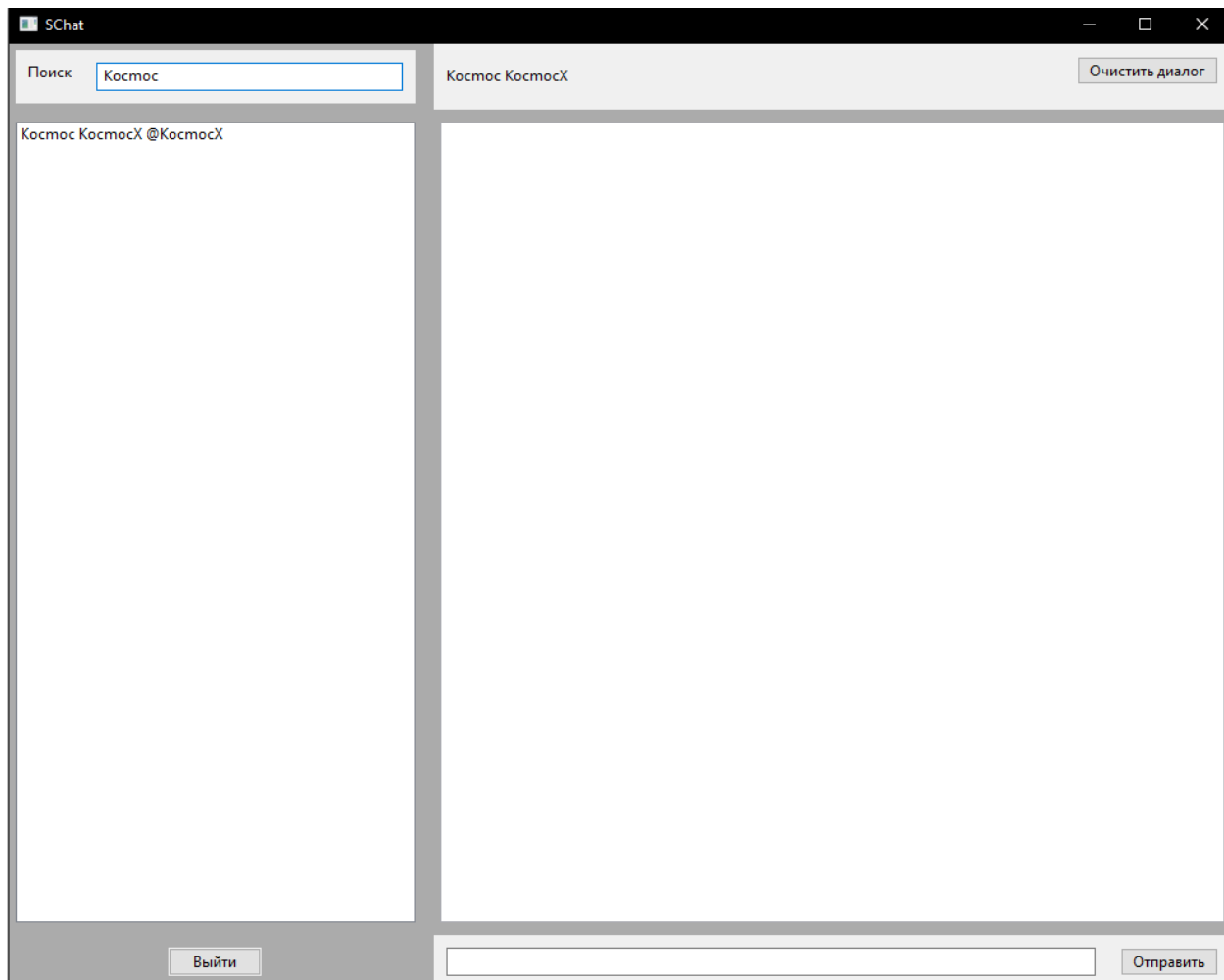
...

Повторите пароль:

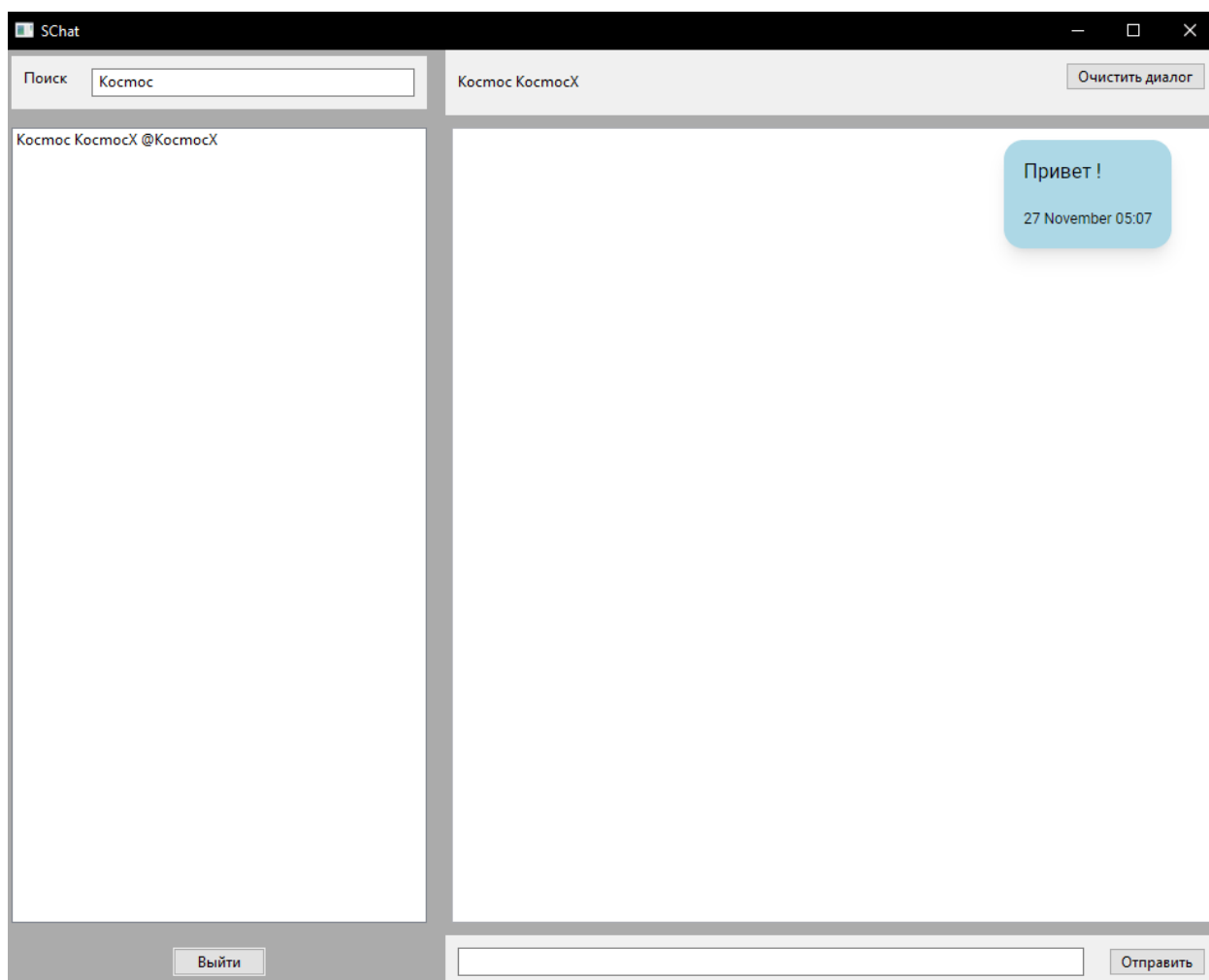
...

Зарегистрироваться

После успешной регистрации пользователь попадает в основное приложение, где сможет вести полноценную переписку с другим человеком, найдя его по нику или имени и фамилии.



Отправим наше 1 сообщение пользователю!

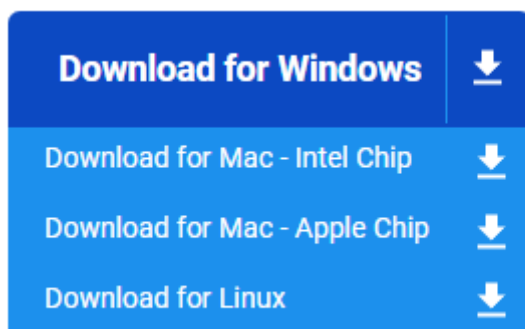
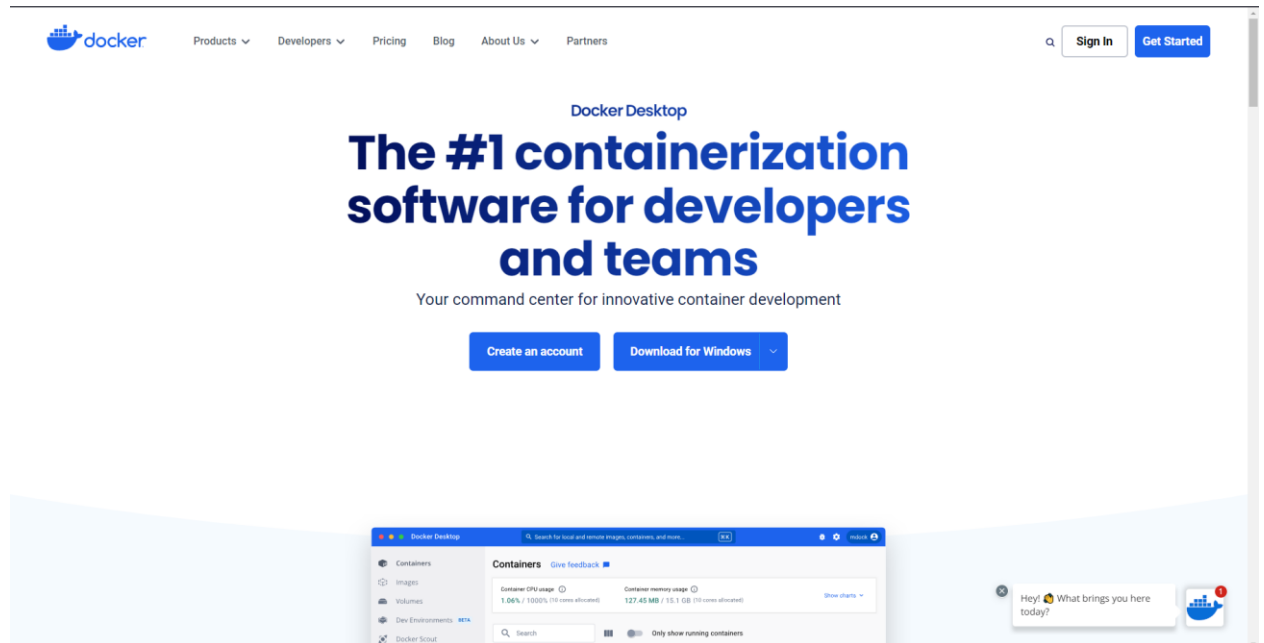


Далее я расскажу о том, как реализовать данный функционал, и что нужно использовать, чтобы всё получилось)

Шаг 1 – Установка Docker на свой компьютер. Что такое Docker и зачем он нужен?

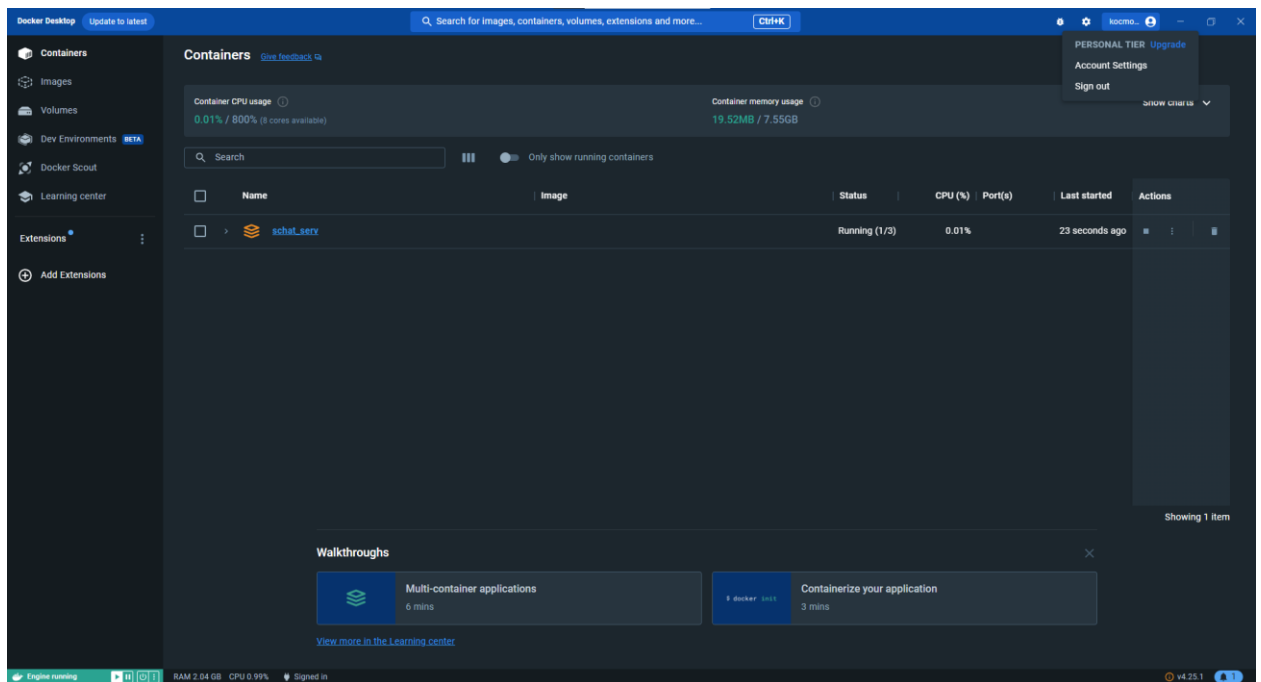
Как скачать Docker:

Заходим на официальный сайт: <https://www.docker.com/products/docker-desktop/>



Выбираем для какой операционной системы мы хотим скачать его.

После загрузки на компьютер, открываем установщик, устанавливаем Docker и входим в свою учётную запись. После запускаем приложение и видим такое окно.



Тут мы можем заметить, что во вкладке Containers уже запущен сервер. Но об этом будет рассказано далее, как его запустить, чтобы увидеть.

Что такое докер?

Докер — это открытая платформа для разработки, доставки и эксплуатации приложений. Docker разработан для более быстрого выкладывания ваших приложений. С помощью docker вы можете отделить ваше приложение от вашей инфраструктуры и обращаться с инфраструктурой как управляемым приложением. Docker помогает выкладывать ваш код быстрее, быстрее тестировать, быстрее выкладывать приложения и уменьшить время между написанием кода и запуском кода. Docker делает это с помощью легковесной платформы контейнерной виртуализации, используя процессы и утилиты, которые помогают управлять и выкладывать ваши приложения.

В своем ядре docker позволяет запускать практически любое приложение, безопасно изолированное в контейнере. Безопасная изоляция позволяет вам запускать на одном хосте много контейнеров одновременно. Легковесная природа контейнера, который запускается без дополнительной нагрузки гипервизора, позволяет вам добиваться больше от вашего железа.

Платформа и средства контейнерной виртуализации могут быть полезны в следующих случаях:

- упаковывание вашего приложения (и так же используемых компонент) в docker контейнеры;
- раздача и доставка этих контейнеров вашим командам для разработки и тестирования;
- выкладывания этих контейнеров на ваши продакшены, как в дата центры, так и в облака.

Для чего я могу использовать docker?

Быстрое выкладывание ваших приложений

Docker прекрасно подходит для организации цикла разработки. Docker позволяет разработчикам использовать локальные контейнеры с приложениями и сервисами. Что в последствии позволяет интегрироваться с процессом постоянной интеграции и выкладывания (continuous integration and deployment workflow).

Например, ваши разработчики пишут код локально и делятся своим стеком разработки (набором docker образов) с коллегами. Когда они готовы, отправляют код и контейнеры на тестовую площадку и запускают любые необходимые тесты. С тестовой площадки они могут оправить код и образы на продакшен.

Главные компоненты Docker

Docker состоит из двух главных компонент:

- Docker: платформа виртуализации с открытым кодом;
- Docker Hub: наша платформа-как-сервис для распространения и управления docker контейнерами.

Архитектура Docker

Docker использует архитектуру клиент-сервер. Docker клиент общается с демоном Docker, который берет на себя тяжесть создания, запуска, распределения ваших контейнеров. Оба, клиент и сервер могут работать на одной системе, вы можете подключить клиент к удаленному демону docker. Клиент и сервер общаются через сокет или через RESTful API.

Так как же работает Docker?

Пока мы знаем, что:

- можем создавать образы, в которых находятся наши приложения;
- можем создавать контейнеры из образов, для запуска приложений;
- можем распространять образы через Docker Hub

Как работает образ?

Мы уже знаем, что образ — это read-only шаблон, из которого создается контейнер.

Каждый образ состоит из набора уровней. Docker использует union file system для сочетания этих уровней в один образ. Union file system позволяет файлам и директориями из разных файловых систем (разным ветвям) прозрачно накладываться, создавая когерентную файловую систему.

Одна из причин, по которой docker легковесен — это использование таких уровней. Когда вы изменяете образ, например, обновляете приложение, создается новый уровень. Так, без замены всего образа или его пересборки, как вам возможно придется сделать с виртуальной машиной, только уровень добавляется или обновляется. И вам не нужно раздавать весь новый образ, раздается только обновление, что позволяет распространять образы проще и быстрее.

В основе каждого образа находится базовый образ. Например, ubuntu, базовый образ Ubuntu, или fedora, базовый образ дистрибутива Fedora. Так же вы можете использовать образы как базу для создания новых образов. Например, если у вас есть образ apache, вы можете использовать его как базовый образ для ваших веб-приложений.

Примечание! Docker обычно берет образы из реестра Docker Hub.

Docker образы могут создаваться из этих базовых образов, шаги описания для создания этих образов мы называем инструкциями. Каждая инструкция создает новый образ или уровень. Инструкциями будут следующие действия:

- запуск команды
- добавление файла или директории
- создание переменной окружения
- указания что запускать, когда запускается контейнер образа

Что происходит, когда запускается контейнер?

Или с помощью программы `docker`, или с помощью RESTful API, `docker` клиент говорит `docker` демону запустить контейнер.

```
$ sudo docker run -i -t ubuntu /bin/bash
```

Давайте разберемся с этой командой. Клиент запускается с помощью команды `docker`, с опцией `run`, которая говорит, что будет запущен новый контейнер. Минимальными требованиями для запуска контейнера являются следующие атрибуты:

- какой образ использовать для создания контейнера. В нашем случае `ubuntu`
- команду которую вы хотите запустить когда контейнер будет запущен. В нашем случае `/bin/bash`

Что же происходит под капотом, когда мы запускаем эту команду?

Docker, по порядку, делает следующее:

- **скачивает образ `ubuntu`:** `docker` проверяет наличие образа `ubuntu` на локальной машине, и если его нет — то скачивает его с [Docker Hub](https://hub.docker.com/). Если же образ есть, то использует его для создания контейнера;
- **создает контейнер:** когда образ получен, `docker` использует его для создания контейнера;

- **инициализирует файловую систему и монтирует read-only**

уровень: контейнер создан в файловой системе и read-only уровень добавлен образ;

- **инициализирует сеть/мост:** создает сетевой интерфейс, который

позволяет docker-у общаться хост машиной;

- **Установка IP адреса:** находит и задает адрес;

- **Запускает указанный процесс:** запускает ваше приложение;

• **Обработывает и выдает вывод вашего приложения:** подключается и логирует стандартный вход, вывод и поток ошибок вашего приложения, что бы вы могли отслеживать как работает ваше приложение.

Теперь у вас есть рабочий контейнер. Вы можете управлять своим контейнером, взаимодействовать с вашим приложением. Когда решите остановить приложение, удалите контейнер.

Containers [Give feedback](#)

Container CPU usage

0.00% / 800% (8 cores available)

















Container memory usage

19.52MB / 7.55GB

Show charts

Search

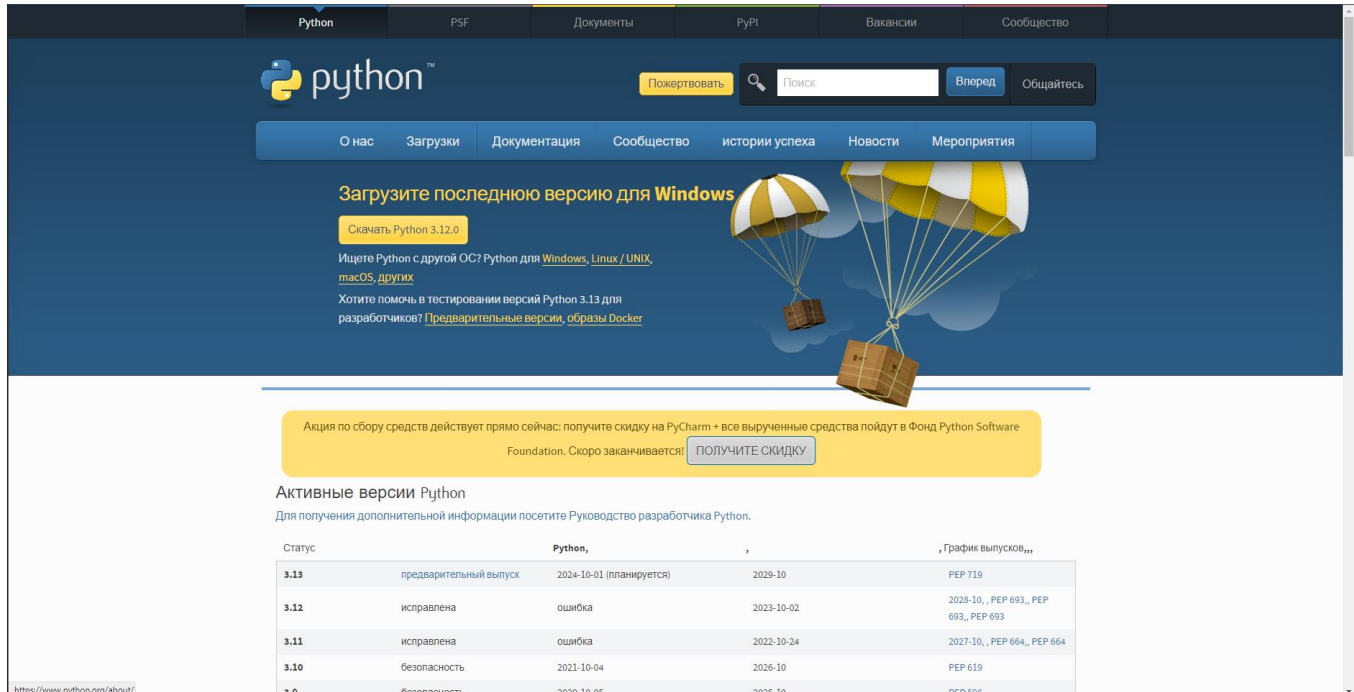
Only show running containers

<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	 schal_serv		Running (1/3)	0%		9 minutes ago	  
<input type="checkbox"/>	 schal_pma 32ecdc389b6b	phomyadmin/phomyadmin	Running	0%	8081:80	9 minutes ago	  
<input type="checkbox"/>	 schal_api 0f96144f1134	schal_serv-api	Exited	0%	3000:3000	5 days ago	  
<input type="checkbox"/>	 schal_db b5a26cd7382b	mysql	Exited	0%	3307:3306	5 days ago	  

Следующий шаг нашей разработки, это скачать Python на компьютер если он ещё не скачен.

Для начала заходим на официальный сайт Python

<https://www.python.org/downloads/>



python™

Пожертвовать Поиск Вперед Общайтесь

О нас Загрузки Документация Сообщество истории успеха Новости Мероприятия

Загрузите последнюю версию для Windows

Скачать Python 3.12.0

Ищете Python с другой ОС? Python для [Windows](#), [Linux](#) / [UNIX](#), [macOS](#), [других](#)

Хотите помочь в тестировании версий Python 3.13 для разработчиков? [Предварительные версии](#), [образы Docker](#)

Акция по сбору средств действует прямо сейчас: получите скидку на PyCharm + все вырученные средства пойдут в Фонд Python Software Foundation. Скоро заканчивается! ПОЛУЧИТЕ СКИДКУ

Активные версии Python

Для получения дополнительной информации посетите Руководство разработчика Python.

Статус	Python	Python	Python	График выпусков...
3.13	предварительный выпуск	2024-10-01 (планируется)	2029-10	PEP 719
3.12	исправлена	ошибка	2023-10-02	2025-10, PEP 693, PEP 693, PEP 693
3.11	исправлена	ошибка	2022-10-24	2027-10, PEP 664, PEP 664
3.10	безопасность	2021-10-04	2026-10	PEP 619
3.9	безопасность	2020-10-05	2025-10	PEP 596

Я создавал проект на Python 3.11

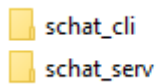
Скачиваем интересующую нас версию.

Для начала предлагаю создать серверную часть.

- 1) Создадим папку на рабочем столе и назовём её например schat.



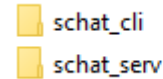
- 2) Внутри папки создадим уже 2 папки, которые понадобятся для реализации серверной и клиентской части мессенджера.



Первым шагом мы создадим серверную часть, и далее я опишу шаги создания сервера, а затем клиента.

Создадим все необходимые файлы для запуска сервера.

Внутри папки `schat_serv` создадим следующие файлы.



at > schat_serv >

Имя	Дата изменения	Тип	Размер
schat_serv	18.11.2023 20:02	Папка с файлами	
.dockerignore	20.11.2023 20:03	Файл "DOCKERIG...	2 КБ
.env	20.11.2023 19:57	Файл "ENV"	2 КБ
docker-compose.yml	20.11.2023 19:26	Исходный файл ...	5 КБ
Dockerfile	20.11.2023 20:11	Файл	2 КБ
pyproject.toml	20.11.2023 20:45	Исходный файл ...	1 КБ
README.md	20.11.2023 20:45	Исходный файл ...	0 КБ

Сейчас я расскажу зачем нужен каждый файл и какой код нужно написать.

1) .dockerignore

Файл `.dockerignore` используется при сборке Docker-образа и указывает, какие файлы и папки должны быть проигнорированы и не включены в образ.

Это полезно, когда вы хотите исключить определенные файлы или папки из контекста сборки Docker-образа. Например, вы можете исключить временные файлы, скомпилированный код, настройки локальной разработки или конфиденциальные данные, чтобы они не попали в Docker-образ.

В файле `.dockerignore`, вы можете указать шаблоны имен файлов или папок, которые следует игнорировать. При сборке Docker-образа Docker будет пропускать эти файлы и не включать их в итоговый образ.

`__pycache__`

Папка `"__pycache__"` содержит скомпилированные версии файлов Python (файлы `.pyc`), создаваемые интерпретатором Python для ускорения загрузки и повторного использования кода.

`.dockerignore`

Файл `".dockerignore"` используется при сборке Docker-образа и определяет файлы и папки, которые должны быть проигнорированы и не включены в образ.

`.docker`

Папка `".docker"` может содержать различные файлы и настройки, связанные с Docker, которые используются для контейнеризации приложений.

`poetry.lock`

Файл `"poetry.lock"` является частью инструмента управления зависимостями Poetry и содержит заблокированные версии зависимостей проекта.

`__pypackages__`

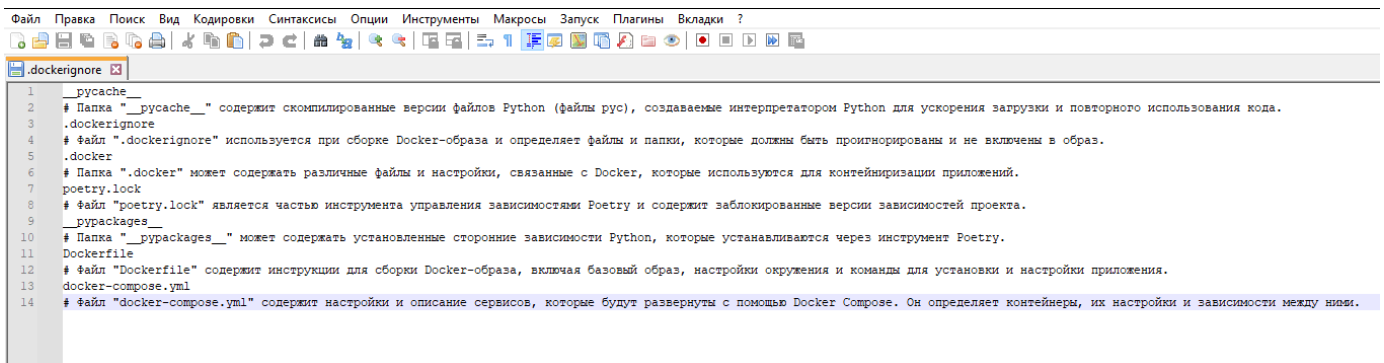
Папка `"__pypackages__"` может содержать установленные сторонние зависимости Python, которые устанавливаются через инструмент Poetry.

`Dockerfile`

Файл "Dockerfile" содержит инструкции для сборки Docker-образа, включая базовый образ, настройки окружения и команды для установки и настройки приложения.

docker-compose.yml

Файл "docker-compose.yml" содержит настройки и описание сервисов, которые будут развернуты с помощью Docker Compose. Он определяет контейнеры, их настройки и зависимости между ними.



2) .env

Файл `.env` используется для хранения конфиденциальных данных и настроек приложения, таких как параметры окружения, секретные ключи, настройки базы данных и прочие переменные окружения.

Файл `.env` часто используется в среде разработки и при запуске приложений в контейнерах Docker.

В файле `.env`, каждая строка представляет собой пару "ключ=значение", где ключ - это название переменной, а значение - соответствующее значение этой переменной.

При запуске приложения, оно может использовать значения из файла `.env`, чтобы настроить свою работу, подключиться к базе данных, установить порты, секреты аутентификации и другие параметры, заданные в файле `.env`.

app server settings

HOST="0.0.0.0"

Устанавливает хост (IP-адрес) сервера приложения. В данном случае "0.0.0.0" означает, что сервер будет прослушивать все доступные IP-адреса.

PORT=3000

Устанавливает номер порта сервера приложения. В данном случае задан порт "3000".

db settings

DB_NAME="app_db"

Задает имя базы данных. В данном случае используется имя "app_db".

DB_USER="app"

Задает имя пользователя базы данных. В данном случае используется имя "app".

DB_PASSWORD="root"

Задает пароль пользователя базы данных. В данном случае используется пароль "root".

DB_PORT="3306"

Задает номер порта базы данных. В данном случае используется порт "3306".

DB_HOST="schat_db"

Задает хост (адрес) базы данных. В данном случае используется хост "schat_db".

JWT_SECRET="supersecret"

Задает секретный ключ для генерации и проверки JWT-токенов аутентификации. В данном случае используется секретный ключ "supersecret".

CIPHER_KEY=9ZhsrBtahlfDmVWNVLLwe2VSjr6OtUWnSFWJCzWfbGc=

Задает ключ шифрования. В данном случае используется ключ "9ZhsrBtahlfDmVWNVLLwe2VSjr6OtUWnSFWJCzWfbGc=".

```
.env x
1 # app server settings
2 HOST="0.0.0.0"
3 # Устанавливает хост (IP-адрес) сервера приложения. В данном случае "0.0.0.0" означает, что сервер будет прослушивать все доступные IP-адреса
4
5 PORT=3000
6 # Устанавливает номер порта сервера приложения. В данном случае задан порт "3000".
7
8 # db settings
9 DB_NAME="app_db"
10 # Задает имя базы данных. В данном случае используется имя "app_db".
11
12 DB_USER="app"
13 # Задает имя пользователя базы данных. В данном случае используется имя "app".
14
15 DB_PASSWORD="root"
16 # Задает пароль пользователя базы данных. В данном случае используется пароль "root".
17
18 DB_PORT="3306"
19 # Задает номер порта базы данных. В данном случае используется порт "3306".
20
21 DB_HOST="schat_db"
22 # Задает хост (адрес) базы данных. В данном случае используется хост "schat_db".
23
24 JWT_SECRET="supersecret"
25 # Задает секретный ключ для генерации и проверки JWT-токенов аутентификации. В данном случае используется секретный ключ "supersecret".
26
27 CIPHER_KEY=9ZhsrBtahlfDmVWNVLLwe2VSjr6OtUWnSFWJCzWfbGc=
28 # Задает ключ шифрования. В данном случае используется ключ "9ZhsrBtahlfDmVWNVLLwe2VSjr6OtUWnSFWJCzWfbGc=".
```

3) docker-compose.yml

Файл `docker-compose.yml` используется для определения и настройки многоконтейнерных приложений с использованием Docker Compose. Он содержит описание сервисов, объединенных вместе, и их конфигурацию.

В файле `docker-compose.yml` определены различные сервисы, которые будут созданы и запущены с помощью Docker. Каждый сервис представляет собой отдельный контейнер, образ которого может быть указан в поле "image" или собран из текущей директории с помощью "build".

Внутри каждого сервиса можно определить различные параметры, такие как пробрасываемые порты ("ports"), переменные окружения ("env_file" или "environment"), монтирование томов ("volumes"), ссылки на другие сервисы ("links"), настройки сети ("networks"), перезапуск контейнера ("restart") и другие.

Файл `docker-compose.yml` позволяет определить зависимости между сервисами с помощью параметра "depends_on". Так, контейнеры могут быть запущены в правильном порядке и подключены друг к другу.

После создания файла `docker-compose.yml`, можно использовать команду `docker-compose up` для запуска приложения в соответствии с его конфигурацией. Docker Compose автоматически создаст и настроит все необходимые контейнеры, объединенные в сеть, их связи и объемы.

Использование `docker-compose.yml` упрощает процесс разворачивания и управления многоконтейнерными приложениями, позволяя определить все требуемые сервисы и их конфигурацию в едином файле и запускать их легко и повторяемо.

version: '3.3' # Версию формата файла конфигурации Docker Compose.

services: # Список сервисов, которые будут запущены.

api: #Сервис с именем "api".

build: . #Сборка контейнера будет выполнена из текущей директории.

container_name: schat_api #Устанавливает имя контейнера в "schat_api".

ports:

- "3000:3000" # Пробрасывает порт 3000 контейнера на порт 3000 хоста.

depends_on:

- db # Устанавливает зависимость от сервиса с именем "db", то есть контейнер "api" будет запущен только после запуска контейнера "db".

env_file:

- .env #файл с переменными окружения (.env), который будет использоваться контейнером "api".

networks:

- fnt # Подключает контейнер "api" к сети с именем "fnt".

db: # сервис с именем "db".

image: mysql # Использует официальный образ MySQL из Docker Hub.

container_name: schat_db # Устанавливает имя контейнера в "schat_db".

environment: #Устанавливает переменные окружения для контейнера MySQL,

```
MYSQL_ROOT_PASSWORD: ${DB_PASSWORD} # такие как пароль администратора (MYSQL_ROOT_PASSWORD),

MYSQL_DATABASE: ${DB_NAME} # имя базы данных (MYSQL_DATABASE),

MYSQL_USER: ${DB_USER} # имя пользователя (MYSQL_USER)

MYSQL_PASSWORD: ${DB_PASSWORD} # пароль пользователя (MYSQL_PASSWORD) || Значения переменных берутся из
файла окружения.

ports:

  - "3307:3306" # Пробрасывает порт 3306 контейнера на порт 3307 хоста.

volumes:

  - dbdata:/var/lib/mysql # Создает именованный том "dbdata" и монтирует его в директорию /var/lib/mysql контейнера, чтобы
сохранить данные MySQL между перезапусками контейнера.

env_file:

  - ./schat_serv/.env # Указывает на файл с переменными окружения (./schat_serv/.env), который будет использоваться
контейнером "db".

networks:

  - fnt # Подключает контейнер "db" к сети с именем "fnt".


phpmyadmin: # Определяет сервис с именем "phpmyadmin".

image: phpmyadmin/phpmyadmin # Использует официальный образ phpMyAdmin из Docker Hub.

container_name: schat_pma # Устанавливает имя контейнера в "schat_pma".

links:

  - db #Устанавливает связь с контейнером "db", чтобы phpMyAdmin мог подключиться к базе данных MySQL.

environment: #Устанавливает переменные окружения для контейнера phpMyAdmin,

  PMA_HOST: schat_db #такие как хост базы данных (PMA_HOST)

  PMA_PORT: 3306 #порт базы данных (PMA_PORT)

  PMA_ARBITRARY: 1 #флаг PMA_ARBITRARY для отключения проверки подключения.

restart: always #автоматическое перезапускание контейнера phpMyAdmin в случае его остановки.

ports: #Пробрасывает порт 80 контейнера на порт 8081 хоста.

  - 8081:80

networks: #Подключает контейнер "phpmyadmin" к сети с именем "fnt".

  - fnt

volumes: #Определяет именованный том "dbdata".

dbdata:


networks: #Определяет сеть с именем "fnt".

fnt:
```

```

1 version: '3.3' # Версия формата файла конфигурации Docker Compose.
2 services: # Список сервисов, которые будут запущены.
3   api: # Сервис с именем "api".
4     build: . # Сборка контейнера будет выполнена из текущей директории.
5     container_name: schat_api # Устанавливает имя контейнера в "schat_api".
6     ports:
7       - "3000:3000" # Прокладывает порт 3000 контейнера на порт 3000 хоста.
8     depends_on:
9       - db # Устанавливает зависимость от сервиса с именем "db", то есть контейнер "api" будет запущен только после запуска контейнера "db".
10    env_file:
11      - .env # Файл с переменными окружения (.env), который будет использоваться контейнером "api".
12    networks:
13      - fnt # Подключает контейнер "api" к сети с именем "fnt".
14
15  db: # сервис с именем "db".
16    image: mysql # Использует официальный образ MySQL из Docker Hub.
17    container_name: schat_db # Устанавливает имя контейнера в "schat_db".
18    environment: # Устанавливает переменные окружения для контейнера MySQL.
19      MYSQL_ROOT_PASSWORD: ${DB_PASSWORD} # такие как пароль администратора (MYSQL_ROOT_PASSWORD),
20      MYSQL_DATABASE: ${DB_NAME} # имя базы данных (MYSQL_DATABASE),
21      MYSQL_USER: ${DB_USER} # имя пользователя (MYSQL_USER)
22      MYSQL_PASSWORD: ${DB_PASSWORD} # пароль пользователя (MYSQL_PASSWORD) || Значения переменных берутся из файла окружения.
23    ports:
24      - "3307:3306" # Прокладывает порт 3306 контейнера на порт 3307 хоста.
25    volumes:
26      - dbdata:/var/lib/mysql # Создает именованный том "dbdata" и монтирует его в директорию /var/lib/mysql контейнера, чтобы сохранить данные MySQL между перезапусками контейнера.
27    env_file:
28      - ./schat_serv/.env # Указывает на файл с переменными окружения (./schat_serv/.env), который будет использоваться контейнером "db".
29    networks:
30      - fnt # Подключает контейнер "db" к сети с именем "fnt".
31
32  phpmyadmin: # Определяет сервис с именем "phpmyadmin".
33    image: phpmyadmin/phpmyadmin # Использует официальный образ phpMyAdmin из Docker Hub.
34    container_name: schat_pma # Устанавливает имя контейнера в "schat_pma".
35    links:
36      - db # Устанавливает связь с контейнером "db", чтобы phpMyAdmin мог подключиться к базе данных MySQL.
37    environment: # Устанавливает переменные окружения для контейнера phpMyAdmin.
38      PMA_HOST: schat_db # такие как хост базы данных (PMA_HOST)
39      PMA_PORT: 3306 # порт базы данных (PMA_PORT)
40      PMA_ABSTRACT: 1 # флаг PMA_ABSTRACT для отключения проверки подключения.
41    restart: always # автоматическое перезапускание контейнера phpMyAdmin в случае его остановки.
42    ports:
43      - 8081:80 # Прокладывает порт 80 контейнера на порт 8081 хоста.
44    networks:
45      - fnt # Подключает контейнер "phpmyadmin" к сети с именем "fnt".
46
47  volumes: # Определяет именованный том "dbdata".
48    dbdata:
49
50  networks: # Определяет сеть с именем "fnt".
51    fnt:

```

Код написан будет использован в дальнейшем для запуска сервера в docker

4) Dockerfile

Файл `Dockerfile` используется для создания Docker-образов, которые представляют собой исполняемые окружения с необходимыми зависимостями и настройками для выполнения приложений.

Внутри `Dockerfile` определяются команды, которые будут выполнены последовательно для создания и настройки образа. Ниже приведены некоторые распространенные команды, используемые в `Dockerfile` и их назначение:

1. **'FROM'**: Определяет базовый образ Docker, от которого будет наследоваться новый образ. В данном случае, выбран образ Python версии 3.11.
2. **'RUN'**: Выполняет команды внутри Docker-образа во время его сборки. Здесь используется команда "pip install poetry" для установки инструмента Poetry.
3. **'COPY'** или **'ADD'**: Копирует файлы и папки из локальной файловой системы внутрь Docker-образа. В данном случае, все файлы и папки из текущего каталога копируются в папку "/"app" внутри образа.
4. **'WORKDIR'**: Устанавливает рабочую директорию внутри Docker-образа, в которой будут выполняться последующие команды. В данном случае, установлена рабочая директория как "/"app".

5. `'EXPOSE'`: Объявляет порт, который будет слушать Docker-контейнер. В данном случае, объявлен порт 3000.

6. `'CMD'` или `'ENTRYPOINT'`: Задаёт команду, которая будет выполняться при запуске Docker-контейнера. В данном случае, используется команда `"poetry run python schat_serv/app.py"` для запуска приложения внутри контейнера.

Создание и сборка Docker-образа происходят на основе `'Dockerfile'` с использованием команды `'docker build'`. Команда считывает `'Dockerfile'`, выполняет в нём команды и создаёт образ, который в дальнейшем можно использовать для запуска контейнеров.

Использование `'Dockerfile'` позволяет определить все зависимости и настройки приложения в едином файле, что обеспечивает повторяемость и лёгкость развёртывания приложения в любом окружении, где установлен Docker.

FROM python:3.11

Базовый образ Docker, который будет использоваться для создания контейнера. В данном случае, используется образ Python версии 3.11.

RUN pip install poetry

Выполняется команда `"pip install poetry"`, чтобы установить инструмент управления зависимостями Poetry внутри Docker-образа.

COPY ./app

Копируются все файлы и папки из текущего каталога (где находится `Dockerfile`) в папку `"/app"` внутри Docker-образа.

WORKDIR /app

Устанавливается рабочая директория внутри Docker-образа как `"/app"`. Все последующие команды будут выполняться относительно этой директории.

RUN poetry install

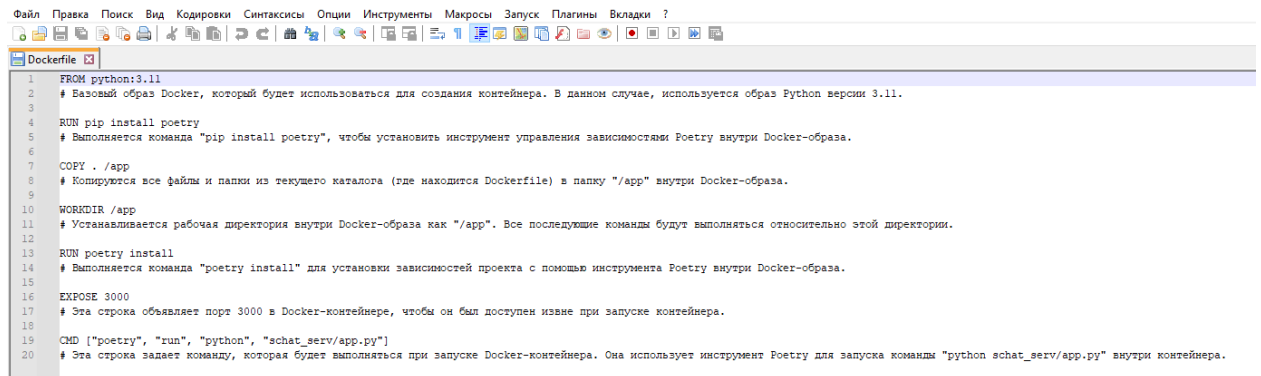
Выполняется команда `"poetry install"` для установки зависимостей проекта с помощью инструмента Poetry внутри Docker-образа.

EXPOSE 3000

Эта строка объявляет порт 3000 в Docker-контейнере, чтобы он был доступен извне при запуске контейнера.

CMD ["poetry", "run", "python", "schat_serv/app.py"]

Эта строка задает команду, которая будет выполняться при запуске Docker-контейнера. Она использует инструмент Poetry для запуска команды "python schat_serv/app.py" внутри контейнера.



```
1 FROM python:3.11
2 # Базовый образ Docker, который будет использоваться для создания контейнера. В данном случае, используется образ Python версии 3.11.
3
4 RUN pip install poetry
5 # Выполняется команда "pip install poetry", чтобы установить инструмент управления зависимостями Poetry внутри Docker-образа.
6
7 COPY . /app
8 # Копируются все файлы и папки из текущего каталога (где находится Dockerfile) в папку "/app" внутри Docker-образа.
9
10 WORKDIR /app
11 # Устанавливается рабочая директория внутри Docker-образа как "/app". Все последующие команды будут выполняться относительно этой директории.
12
13 RUN poetry install
14 # Выполняется команда "poetry install" для установки зависимостей проекта с помощью инструмента Poetry внутри Docker-образа.
15
16 EXPOSE 3000
17 # Эта строка объявляет порт 3000 в Docker-контейнере, чтобы он был доступен извне при запуске контейнера.
18
19 CMD ["poetry", "run", "python", "schat_serv/app.py"]
20 # Эта строка задает команду, которая будет выполняться при запуске Docker-контейнера. Она использует инструмент Poetry для запуска команды "python schat_serv/app.py" внутри контейнера.
```

В дальнейшем нам понадобится этот код, для понимания, как запустить сервер через консоль.

5) pyproject.toml

Файл `pyproject.toml` является частью проекта, использующего инструмент управления зависимостями Poetry. Он содержит конфигурацию проекта, включая зависимости и настройки сборки.

Внутри `pyproject.toml` определены различные секции, которые содержат информацию о проекте:

1. `[tool.poetry]`: Эта секция содержит основные метаданные о проекте, такие как название, версия, описание, авторы и ссылка на файл README. Эти данные используются инструментом Poetry для управления проектом.
2. `[tool.poetry.dependencies]`: В этой секции перечислены зависимости проекта, включая Python и различные библиотеки, которые нужны для работы приложения. Каждая зависимость указывается с ее версией или диапазоном версий.
3. `[build-system]`: В данной секции задаются требования для системы сборки проекта. Здесь указывается зависимость на Poetry Core и используемый backend для сборки проекта.

Файл `pyproject.toml` позволяет легко управлять зависимостями проекта и их версиями. Инструмент Poetry использует этот файл для установки и обновления зависимостей из указанных источников, включая PyPI (Python Package Index).

Кроме того, `pyproject.toml` обеспечивает консистентность и повторяемость среды разработки, так как все зависимости проекта перечислены в одном файле. Это упрощает установку и настройку проекта на различных машинах или серверах.

Обратите внимание, что `pyproject.toml` является стандартным файлом конфигурации для проектов, использующих инструмент Poetry. Он заменил более старые файлы конфигурации, такие как `requirements.txt` или `setup.py`, и стал предпочтительным способом управления зависимостями в экосистеме Python.

```
[tool.poetry]
```

```
name = "schat_serv"
```

```
version = "0.1.0"
```

```
description = ""
```

```
authors = ["Your Name <you@example.com>"]
```

```
readme = "README.md"
```

Основные метаданные проекта, такие как название, версия, описание, авторы и ссылка на README файл.

```
[tool.poetry.dependencies]
```

```
python = "^3.11"
```

```
fastapi = "^0.104.1"
```

```
peewee = "^3.17.0"
```

```
python-dotenv = "^1.0.0"
```

```
uvicorn = "^0.24.0.post1"
```

```
 pymysql = "^1.1.0"
```

```
 cryptography = "^41.0.5"
```

```
 pydantic = "^2.4.2"
```

```
 pyjwt = "^2.8.0"
```

```
 passlib = "^1.7.4"
```

```
 websockets = "^12.0"
```

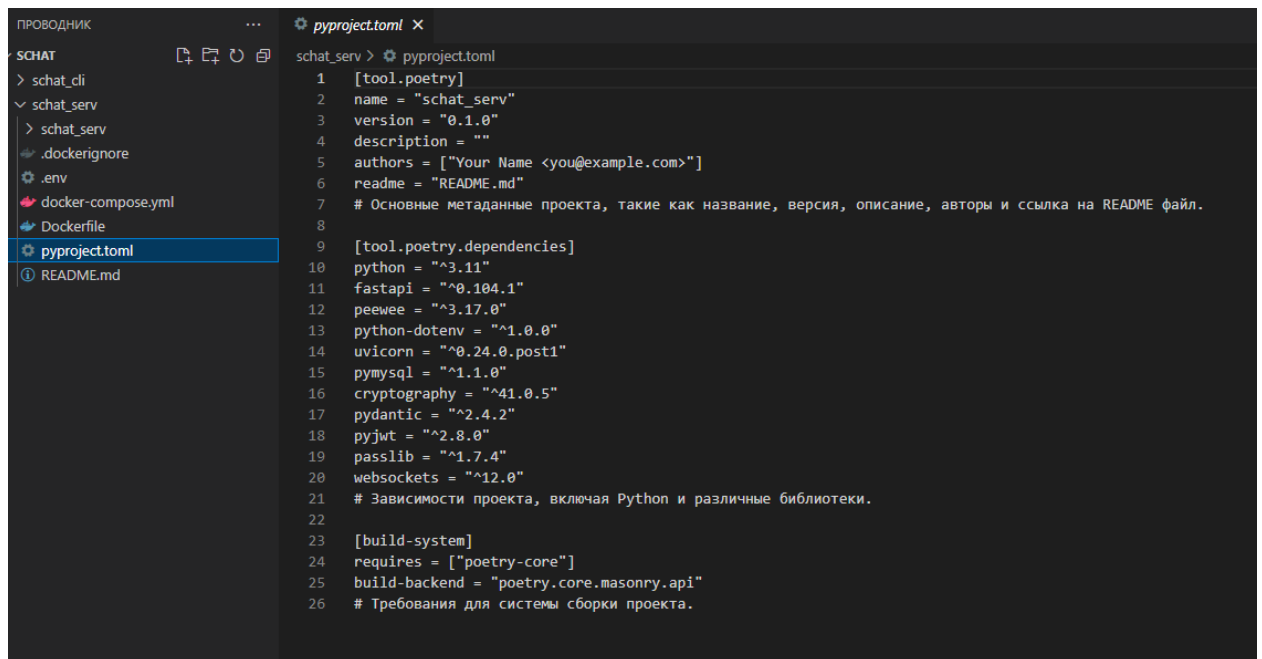
Зависимости проекта, включая Python и различные библиотеки.

```
[build-system]
```

```
requires = ["poetry-core"]
```

```
build-backend = "poetry.core.masonry.api"
```

Требования для системы сборки проекта.



Следующим шагом будет создание папки schat_serv

Поделиться		Вид			
schat		schat_serv		▼	🔄
				🔍	
	Имя	Дата изменения	Тип	Размер	
📁	schat_serv	18.11.2023 20:02	Папка с файлами		
📄	.dockerignore	20.11.2023 20:03	Файл "DOCKERIG...	2 КБ	
📄	.env	20.11.2023 19:57	Файл "ENV"	2 КБ	
📄	docker-compose.yml	20.11.2023 19:26	Исходный файл ...	5 КБ	
📄	Dockerfile	20.11.2023 20:11	Файл	2 КБ	
📄	pyproject.toml	20.11.2023 20:45	Исходный файл ...	1 КБ	
📄	README.md	20.11.2023 20:45	Исходный файл ...	0 КБ	

Внутри папки `schat_serv` создадим все необходимые файлы для работы самого сервера.

Структура самой папки должна выглядеть следующим образом:

Создадим все необходимые файлы!

schat > schat_serv > schat_serv >						
	Имя	Дата изменения	Тип	Размер		
	models	18.11.2023 20:00	Папка с файлами			
	routers	18.11.2023 20:00	Папка с файлами			
	schemas	18.11.2023 20:00	Папка с файлами			
	services	18.11.2023 20:00	Папка с файлами			
	.env	18.11.2023 20:16	Файл "ENV"	1 КБ		
	.env.example	18.11.2023 20:00	Файл "EXAMPLE"	1 КБ		
	app.py	18.11.2023 20:00	Исходный файл ...	2 КБ		

Для начала рассмотрим что это за файлы и зачем они нужны.

1) Файл `.env`

Файл `.env` (Environment file) используется для хранения конфиденциальной информации и настроек окружения в проектах. Он содержит пары ключ-значение, представляющие переменные окружения, которые могут быть использованы в приложении.

В данном конкретном примере, файл `.env` содержит настройки для сервера приложения и базы данных, а также секретный ключ для JWT (JSON Web Tokens) и ключ для шифрования.

Некоторые причины использования файла `.env` в проекте включают:

1. Управление настройками окружения: `.env` обеспечивает удобный способ управления настройками окружения в проекте. Вы можете определить переменные окружения, такие как хост, порт, пароли и другие конфиденциальные данные, и использовать их в своем коде. Это позволяет легко настраивать приложение для разных сред разработки, таких как локальная разработка, тестирование или развертывание на сервере.

2. Безопасность: Чувствительные данные, такие как пароли, ключи API или секреты аутентификации, не должны быть хранены в открытом виде в коде или

репозитории. Использование файла `.env` для хранения таких данных позволяет сохранить конфиденциальность информации. Файл `.env` обычно добавляется в файл `.gitignore`, чтобы не попадать в общее хранилище кода.

3. Удобство в разработке: Работа с переменными окружения через файл ``.env`` обеспечивает гибкость и удобство. Вы можете легко изменять значения переменных в файле ``.env``, без необходимости изменять сам код. Это делает процесс разработки более гибким и позволяет настраивать приложение под различные сценарии использования.

```
# app server settings
```

HOST="0.0.0.0" # Хост, на котором будет запущен сервер приложения

PORT=3000 **# Порт, на котором будет слушать сервер приложения**

db settings

DB_NAME="app_db" # Имя базы данных, к которой приложение будет подключаться

DB_USER="app" # Имя пользователя базы данных

```
DB_PASSWORD="root" # Пароль пользователя базы данных
```

DB PORT="3306" # Порт базы данных

```
DB_HOST="schat_db" # Хост базы данных
```

```
JWT_SECRET="supersecret" # Секретный ключ для JWT (JSON Web Tokens)
```

CIPHER_KEY=9ZhsrBtahlfDmVWNVLIwe2VSjr6OtUWnSFWJCzWfbGc= # Ключ для шифрования

A screenshot of a code editor window. The title bar shows standard Windows menu items: 'Файл', 'Правка', 'Поиск', 'Вид', 'Кодировки', 'Опции', 'Инструменты', 'Макросы', 'Запуск', 'Плагины', 'Вкладки', and a question mark. Below the title bar is a toolbar with various icons for file operations, editing, and development. The editor's tab bar shows a single tab named '.env'. The main text area contains the following configuration:

```
1 # app server settings
2 HOST="0.0.0.0"           # Хост, на котором будет запущен сервер приложения
3 PORT=3000                # Порт, на котором будет слушать сервер приложения
4
5 # db settings
6 DB_NAME="app_db"         # Имя базы данных, к которой приложение будет подключаться
7 DB_USER="app"            # Имя пользователя базы данных
8 DB_PASSWORD="root"       # Пароль пользователя базы данных
9 DB_PORT="3306"           # Порт базы данных
10 DB_HOST="schat_db"       # Хост базы данных
11
12 JWT_SECRET="supersecret" # Секретный ключ для JWT (JSON Web Tokens)
13 CIPHER_KEY=9ZhSrBtalfDmVWNVLLwe2VSjr6OtUWnSFWJCzWfbGc= # Ключ для шифрования
14
```

2) Файл .env.example

Файл `.env.example` используется в качестве шаблона для конфигурационного файла `.env`. Он содержит заготовку для переменных окружения, которые приложение ожидает для правильной работы.

Основная цель `.env.example` - предоставить разработчикам и пользователям приложения примеры переменных окружения, которые должны быть настроены в файле `.env`.

В приведенном примере, файл `.env.example` предоставляет пустые строки для всех настроек приложения и базы данных, таких как `HOST`, `PORT`, `DB_NAME`, `DB_USER`, `DB_PASSWORD`, `DB_PORT`, `DB_HOST` и `JWT_SECRET`.

Можно скопировать `.env.example` в `.env` и затем заполнить значения для каждой переменной в соответствии с требованиями своей среды разработки или развертывания. Это позволяет настраивать приложение для работы на конкретном сервере или с конкретной базой данных без необходимости изменения самого кода приложения.

app server settings

`HOST=""` # Переменная для хранения хоста (адреса) сервера приложения

`PORT=""` # Переменная для хранения порта сервера приложения

db settings

`DB_NAME=""` # Переменная для хранения имени базы данных

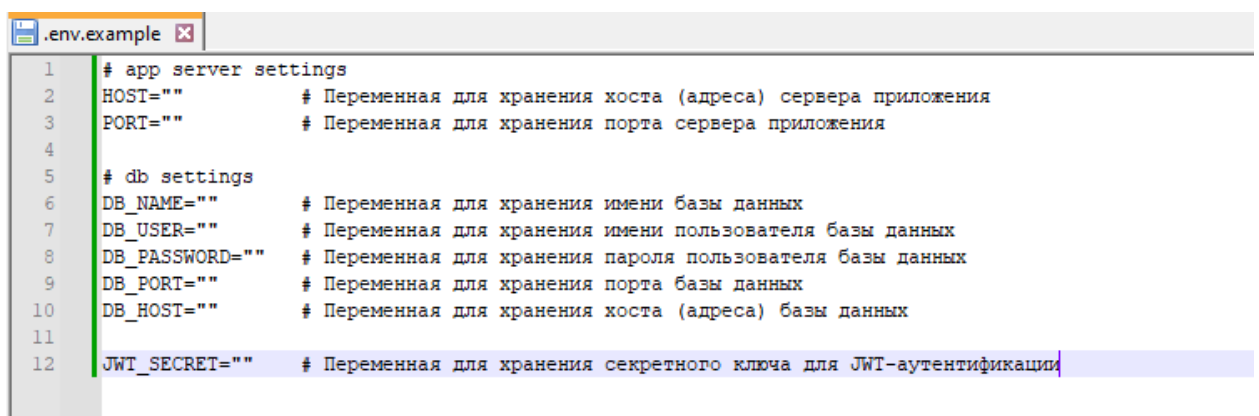
`DB_USER=""` # Переменная для хранения имени пользователя базы данных

`DB_PASSWORD=""` # Переменная для хранения пароля пользователя базы данных

`DB_PORT=""` # Переменная для хранения порта базы данных

`DB_HOST=""` # Переменная для хранения хоста (адреса) базы данных

`JWT_SECRET=""` # Переменная для хранения секретного ключа для JWT-аутентификации

A screenshot of a code editor window titled ".env.example". The editor shows 12 lines of code. Lines 1-4 are grouped under the comment "# app server settings". Lines 5-11 are grouped under the comment "# db settings". Line 12 is for the JWT secret. The code is as follows:

```
1 # app server settings
2 HOST="" # Переменная для хранения хоста (адреса) сервера приложения
3 PORT="" # Переменная для хранения порта сервера приложения
4
5 # db settings
6 DB_NAME="" # Переменная для хранения имени базы данных
7 DB_USER="" # Переменная для хранения имени пользователя базы данных
8 DB_PASSWORD="" # Переменная для хранения пароля пользователя базы данных
9 DB_PORT="" # Переменная для хранения порта базы данных
10 DB_HOST="" # Переменная для хранения хоста (адреса) базы данных
11
12 JWT_SECRET="" # Переменная для хранения секретного ключа для JWT-аутентификации
```

3) Файл `app.py`

Файл `app.py` является главным файлом приложения. Он выполняет несколько важных функций:

1. Импортирует необходимые модули и библиотеки:

- ``os``: модуль для работы с операционной системой, используется для работы с переменными окружения и файловой системой.
- ``dotenv``: библиотека для загрузки переменных окружения из файла ``.env``.
- ``cryptography.fernet``: библиотека для генерации и работы с шифровальными ключами.
- ``uvicorn``: библиотека для запуска FastAPI-приложения.
- ``FastAPI``: фреймворк для создания веб-приложений на основе Python.
- ``models``: модуль с определением моделей базы данных.
- ``routers``: пакет с маршрутами (роутерами) приложения.

2. Загружает переменные окружения из файла ``.env`` с помощью ``load_dotenv()``.

Файл ``.env`` содержит различные настройки для приложения, такие как адрес хоста и порт, секретный ключ шифрования и другие параметры.

3. Инициализирует объект FastAPI с помощью ``FastAPI()``.

Здесь определяются основные настройки приложения, такие как название, URL для документации API и версия.

4. Включает все роутеры (маршруты) в основное приложение с помощью ``app.include_router()``.

Роутеры определены в различных модулях внутри пакета ``routers``. Каждый роутер отвечает за обработку запросов к определенному набору эндпоинтов (URL) и имеет связанный тег, который помогает в организации документации и группировке эндпоинтов.

5. Определяет корневой эндпоинт ``/`` с помощью ``@app.get("/")``.

При обращении к этому эндпоинту будет возвращаться сообщение "API server or {title}", где ``{title}`` - заголовок приложения.

6. Если файл `app.py` запускается напрямую (а не импортируется), то вызывается функция ``models.database_create()`` для создания базы данных (если

необходимо) и запускается сервер с помощью ``uvicorn.run()`. Здесь указывается модуль `app:app`, где первый `app` - название файла, а второй `app` - экземпляр FastAPI-приложения. Также указываются порт, хост и флаг `reload=True`, чтобы сервер автоматически перезапускался при изменении файлов при разработке.`

Файл ``app.py`` отвечает за конфигурацию и запуск FastAPI-приложения, включая определение эндпоинтов, подключение роутеров, настройку переменных окружения и запуск сервера.

```
import os

from dotenv import load_dotenv

from cryptography.fernet import Fernet

import uvicorn

from fastapi import FastAPI

import models

from routers import auth, user, dialog, message

# Загрузить переменные окружения из файла .env

load_dotenv()

# Переменные из файла .env

title = "schat" # Заголовок приложения

port = int(os.getenv("PORT") or 3000) # Порт, используемый сервером (берется из переменной
окружения PORT, если она задана, иначе используется 3000)

host = os.getenv("HOST") or "localhost" # Хост, используемый сервером (берется из переменной
окружения HOST, если она задана, иначе используется "localhost")

# Если переменная окружения CIPHER_KEY не задана, генерировать новый шифровальный
ключ и сохранить его в файл .env

if not os.getenv("CIPHER_KEY"):
```

```

key = Fernet.generate_key() # Генерировать новый шифровальный ключ

with open(".env", "a") as env_file:

    env_file.write(f'CIPHER_KEY={key.decode()}') # Сохранить ключ в файл .env

# Инициализация объекта FastAPI с заданными параметрами
app = FastAPI(

    title=title, # Заголовок приложения

    docs_url="/api/docs", # URL для документации API

    openapi_url="/api/openapi.json", # URL для открытой спецификации OpenAPI

    version="1.0" # Версия приложения
)

# Подключение всех роутеров (маршрутов) к основному приложению
app.include_router(auth.auth_route, prefix="/api/auth", tags=["Auth"])
app.include_router(user.user_route, prefix="/api/user", tags=["User"])
app.include_router(dialog.dialog_route, prefix="/api/dialog", tags=["Dialog"])
app.include_router(message.message_router, prefix="/api/message", tags=["Message"])

# Корневой эндпоинт
@app.get("/")

async def root():

    return f"API server or {title}"

# Если файл app.py запускается напрямую (а не импортируется)
if __name__ == "__main__":

    models.database_create() # Создание базы данных

# Запуск сервера с параметрами из переменных
uvicorn.run("app:app", port=port, host=host, reload=True)

```

```

1 import os
2 from dotenv import load_dotenv
3 from cryptography.fernet import Fernet
4
5 import uvicorn
6 from fastapi import FastAPI
7
8 import models
9 from routers import auth, user, dialog, message
10
11 # Загрузить переменные окружения из файла .env
12 load_dotenv()
13
14 # Переменные из файла .env
15 title = "schat" # Заголовок приложения
16 port = int(os.getenv("PORT") or 3000) # Порт, используемый сервером (берется из переменной окружения PORT, если она задана, иначе используется 3000)
17 host = os.getenv("HOST") or "localhost" # Хост, используемый сервером (берется из переменной окружения HOST, если она задана, иначе используется "localhost")
18
19 # Если переменная окружения CIPHER_KEY не задана, генерировать новый шифровальный ключ и сохранить его в файл .env
20 if not os.getenv("CIPHER_KEY"):
21     key = Fernet.generate_key() # Генерировать новый шифровальный ключ
22     with open(".env", "a") as env_file:
23         env_file.write(f"CIPHER_KEY={key.decode()}") # Сохранить ключ в файл .env
24
25 # Инициализация объекта FastAPI с заданными параметрами
26 app = FastAPI(
27     title=title, # Заголовок приложения
28     docs_url="/api/docs", # URL для документации API
29     openapi_url="/api/openapi.json", # URL для открытой спецификации OpenAPI
30     version="1.0" # Версия приложения
31 )
32
33 # Подключение всех роутеров (маршрутов) к основному приложению
34 app.include_router(auth.auth_route, prefix="/api/auth", tags=["Auth"])
35 app.include_router(user.user_route, prefix="/api/user", tags=["User"])
36 app.include_router(dialog.dialog_route, prefix="/api/dialog", tags=["Dialog"])
37 app.include_router(message.message_route, prefix="/api/message", tags=["Message"])
38
39 # Корневой эндпоинт
40 @app.get("/")
41 async def root():
42     return f"API server or {title}"
43
44 # Если файл app.py запускается напрямую (а не импортируется)
45 if __name__ == "__main__":
46     models.database_create() # Создание базы данных
47
48     # Запуск сервера с параметрами из переменных
49     uvicorn.run("app:app", port=port, host=host, reload=True)

```


Теперь начнём разбирать следующие файлы в папках для создания внутренностей сервера.

t > schat_serv > schat_serv >		▼	↺	Поиск в: schat_serv	
Имя	Дата изменения	Тип	Размер		
models	18.11.2023 20:00	Папка с файлами			
routers	18.11.2023 20:00	Папка с файлами			
schemas	18.11.2023 20:00	Папка с файлами			
services	18.11.2023 20:00	Папка с файлами			
.env	18.11.2023 20:16	Файл "ENV"	1 КБ		
.env.example	18.11.2023 20:00	Файл "EXAMPLE"	1 КБ		
app.py	18.11.2023 20:00	Исходный файл ...	2 КБ		

1) Рассмотрим файлы внутри папки models (модели)

Создадим папку модели, где будет лежать наша база данных для дальнейшего использования, а именно создадим следующие файлы с расширением .py

at > schat_serv > schat_serv > models		▼	↺	Поиск в: models	
Имя	Дата изменения	Тип	Размер		
__init__.py	18.11.2023 20:00	Исходный файл ...	1 КБ		
database.py	18.11.2023 20:00	Исходный файл ...	1 КБ		
dialog.py	18.11.2023 20:00	Исходный файл ...	1 КБ		
dialog_message.py	18.11.2023 20:00	Исходный файл ...	1 КБ		
message.py	18.11.2023 20:00	Исходный файл ...	1 КБ		
user.py	18.11.2023 20:00	Исходный файл ...	1 КБ		

1) Для начала файл напишем код для файла **_init_.py** он нужен для того, чтобы сделать

создание таблиц в базе данных для моделей User, Message, Dialog и DialogMessage.

Он использует объект базы данных (database) и модели, чтобы создать соответствующие таблицы.

```
from models.database import database #Здесь мы импортируем базу данных, которая
будет использоваться в нашем проекте.
from models.user import User #Здесь мы импортируем модель User, которая будет
представлять пользователя в нашей базе данных.
from models.dialog import Dialog #Здесь мы импортируем модель Dialog, которая
будет представлять диалог (чат) в нашей базе данных.
from models.dialog_message import DialogMessage #Здесь мы импортируем модель
DialogMessage, которая будет представлять сообщение внутри диалога в нашей базе
данных.
from models.message import Message #Здесь мы импортируем модель Message, которая
будет представлять сообщение отдельно от диалога в нашей базе данных.

def database_create(): #Здесь мы объявляем функцию с именем "database_create".
    with database: #Здесь мы используем контекстный менеджер "with" для открытия
базы данных.
        database.create_tables([User, Message, Dialog, DialogMessage]) #Здесь мы
вызываем метод "create_tables" из базы данных, передавая ему список моделей
(User, Message, Dialog, DialogMessage). Этот метод создаст таблицы в базе данных
для каждой модели, если они еще не существуют.
```

2) Далее создадим файл **database.py**

Используется для настройки подключения к базе данных MySQL с использованием ORM-библиотеки Peewee.

Здесь мы получаем конфигурационные данные (имя базы данных, имя пользователя, пароль, хост и порт) из переменных окружения, загружаемых из файла .env с помощью модуля dotenv.

Затем мы создаем объект MySQLDatabase из Peewee, передавая ему полученные значения для установления соединения с базой данных. Этот объект будет использоваться для выполнения операций CRUD с моделями, определенными в проекте.

Далее, мы определяем базовую модель BaseModel, от которой будут наследоваться остальные модели в проекте. В этой базовой модели устанавливаем соединение с базой данных, указывая на использование созданного ранее объекта MySQLDatabase.

Такой подход позволяет абстрагироваться от непосредственного взаимодействия с базой данных и работать с моделями и объектами, что делает код проще в поддержке и повышает его читаемость. При необходимости изменения параметров подключения к базе данных достаточно будет внести изменения только в переменные окружения или файл `.env`, без необходимости изменять код самого приложения.

```
import os #строка импортирует модуль os, который предоставляет функции для работы
с операционной системой.
from dotenv import load_dotenv #для хранения конфиденциальной информации, такой
как данные для подключения к базе данных.
from peewee import Model, MySQLDatabase # простая в использовании ORM (объектно-
реляционное отображение) для Python, которая позволяет работать с базами данных
через объектно-ориентированный подход.

load_dotenv() #Эта строка вызывает функцию load_dotenv(), которая загружает
переменные окружения из файла .env в текущую среду.

db_name = os.getenv("DB_NAME") #Здесь мы используем модуль os для получения
значения переменной окружения DB_NAME, которая должна содержать имя базы данных.
db_user = os.getenv("DB_USER") # Аналогично, мы получаем значение переменной
окружения DB_USER, которая содержит имя пользователя базы данных.
db_password = os.getenv("DB_PASSWORD") #Здесь мы получаем значение переменной
окружения DB_PASSWORD, которая содержит пароль для подключения к базе данных.
db_host = os.getenv("DB_HOST") or "localhost" #получаем значение переменной
окружения DB_HOST, которая содержит адрес хоста базы данных. Если переменная
окружения не установлена, используется значение "localhost" по умолчанию.
db_port = int(os.getenv("DB_PORT") or 3306) #Здесь мы получаем значение
переменной окружения DB_PORT, которая должна содержать порт для подключения к
базе данных. Если переменная окружения не установлена, используется значение 3306
(стандартный порт MySQL) по умолчанию.

database = MySQLDatabase( #Мы создаем экземпляр класса MySQLDatabase из модуля
peewee и передаем значения переменных окружения, полученные ранее, для настройки
подключения к базе данных.
    db_name, host=db_host, port=db_port, user=db_user, password=db_password
)

class BaseModel(Model): #Здесь мы определяем внутренний класс Meta внутри
BaseModel. В этом классе можно указать дополнительные настройки модели, такие как
указание используемой базы данных.
    class Meta: #Здесь мы определяем внутренний класс Meta внутри BaseModel. В
этом классе можно указать дополнительные настройки модели, такие как указание
используемой базы данных.
        database = database #Мы присваиваем классу Meta атрибут database, который
ссылается на ранее созданный объект базы данных. Это указывает peewee
```

использовать этот объект для выполнения операций CRUD (создание, чтение, обновление, удаление) с моделями.

3) Следующий файл **dialog.py**

Данный код определяет модель `dialog.py` которая имеет два поля `first_user` и `second_user`, каждое из которых является внешним ключом, указывающим на модель `User`. Он позволяет установить отношение между моделями `Dialog` и `User`, где каждый диалог связан с двумя пользователями. Это может быть полезно для создания системы чатов или обмена сообщениями, где требуется связь между пользователями и диалогами.

```
from peewee import ForeignKeyField #Импортируем класс ForeignKeyField из модуля
peewee. ForeignKeyField используется для определения внешнего ключа в базе
данных.
from models.database import BaseModel #Импортируем базовую модель BaseModel из
модуля models.database. BaseModel является базовым классом моделей, в котором
устанавливается соединение с базой данных.
from models.user import User #Импортируем модель User из модуля models.user. User
представляет модель пользователя в системе.

class Dialog(BaseModel): #Определяем класс Dialog, который наследуется от
BaseModel. Таким образом, модель Dialog будет иметь все функциональности и
свойства, определенные в базовой модели.
    first_user = ForeignKeyField(User) # type: User Указывается тип поля. В
данном случае, поле first_user будет иметь тип данных User.
    second_user = ForeignKeyField(User) # type: User Определяем поле second_user
в модели Dialog.
```

4) Следующий файл **dialog_message.py**

Файл `dialog_message.py` содержит определение модели `DialogMessage`, которая представляет связь между диалогом и сообщением в базе данных.

В модели `DialogMessage` объявлены два внешних ключа: `dialog` и `message`.

- Внешний ключ `dialog` указывает на связанный диалог из модели `Dialog`.
- Внешний ключ `message` указывает на связанное сообщение из модели `Message`.

Таким образом, модель `DialogMessage` позволяет устанавливать связь между конкретным диалогом и сообщением в базе данных.

```

from peewee import ForeignKeyField # Импорт ForeignKeyField из модуля peewee
from models.database import BaseModel # Импорт базовой модели BaseModel из
модуля models.database
from models.dialog import Dialog # Импорт модели Dialog из модуля models.dialog
from models.message import Message # Импорт модели Message из модуля
models.message

class DialogMessage(BaseModel): # Определение нового класса DialogMessage,
наследующего BaseModel
    dialog = ForeignKeyField(Dialog) # Определение поля 'dialog' в модели
DialogMessage типа ForeignKeyField, связанного с моделью Dialog
    message = ForeignKeyField(Message) # Определение поля 'message' в модели
DialogMessage типа ForeignKeyField, связанного с моделью Message

```

5) Следующий файл **message.py**

Файл `message.py` содержит определение класса `Message`, который представляет модель сообщения в приложении. Каждое сообщение имеет следующие атрибуты:

1. `sender`: это поле представляет отправителя сообщения и ссылается на модель пользователя `User`. Оно объявлено как поле внешнего ключа (`ForeignKeyField`), что означает, что значение этого поля должно быть существующим пользователем в базе данных.
2. `created_at`: это поле представляет время создания сообщения и является объектом класса `DateTimeField`. Оно не может быть пустым (`null=False`), что означает, что для каждого сообщения должно быть указано время создания.
3. `content`: это поле содержит текстовое содержимое сообщения и объявлено как объект класса `TextField`. Оно также не может быть пустым (`null=False`), поэтому каждое сообщение должно иметь текстовое содержимое.

Файл важен для функциональности приложения, связанной с обменом сообщениями между пользователями. Он определяет структуру таблицы в базе данных, связанной с сообщениями, и позволяет выполнять операции создания, чтения, обновления и удаления (CRUD) сообщений в базе данных.

```

from peewee import DateTimeField, ForeignKeyField, TextField # импорт
from models.database import BaseModel
from models.user import User
from datetime import datetime

# Определение модели сообщения

class Message(BaseModel):
    sender = ForeignKeyField(User) # Определение поля-внешнего ключа "sender",
которое ссылается на модель "User"
    created_at = DateTimeField(null=False) # Определение поля "created_at" со
значением времени и даты создания сообщения; null=False означает, что поле не
может быть пустым

```

```
content = TextField(null=False) # Определение поля "content" для хранения
текстового содержимого сообщения; null=False означает, что поле не может быть
пустым
```

6) Следующий файл **user.py**

Файл `user.py` определяет модель `User` для работы с базой данных с использованием библиотеки Peewee.

Модель `User` имеет следующие поля:

- `first_name`: текстовое поле (CharField) с максимальной длиной 100 символов, обязательное для заполнения (null=False) и неуникальное (unique=False). Хранит имя пользователя.
- `last_name`: текстовое поле (CharField) с максимальной длиной 100 символов, обязательное для заполнения (null=False) и неуникальное (unique=False). Хранит фамилию пользователя.
- `email`: текстовое поле (CharField) с максимальной длиной 150 символов, обязательное для заполнения (null=False) и уникальное (unique=True). Хранит адрес электронной почты пользователя.
- `nickname`: текстовое поле (CharField) с максимальной длиной 150 символов, обязательное для заполнения (null=False) и уникальное (unique=True). Хранит никнейм (псевдоним) пользователя.
- `password_hash`: текстовое поле (CharField) с максимальной длиной 255 символов, обязательное для заполнения (null=False) и неуникальное (unique=False). Хранит хэш пароля пользователя.
- `token`: текстовое поле (TextField) со значением null, которое может быть установлено (null=True), и неуникальное (unique=False). Хранит токен, который может быть использован для аутентификации пользователя (если установлен).

Этот файл определяет структуру и типы данных, которые будут использоваться для создания таблицы `User` в базе данных, а также содержит необходимые правила валидации и ограничения для каждого поля.

```
from peewee import CharField, TextField
from models.database import BaseModel

class User(BaseModel):
    first_name = CharField(max_length=100, null=False, unique=False)
    last_name = CharField(max_length=100, null=False, unique=False)
    email = CharField(max_length=150, null=False, unique=True)
    nickname = CharField(max_length=150, null=False, unique=True)
    password_hash = CharField(max_length=255, null=False, unique=False)
    token = TextField(null=True, unique=False)
```

2) Рассмотрим файлы внутри папки routers (маршрутизаторы)

1) Рассмотрим данный файл auth.py

Файл auth.py представляет собой модуль, который содержит реализацию роутов для аутентификации и авторизации веб-приложения. Он использует библиотеку FastAPI для создания API-роутера и определения эндпоинтов.

Этот модуль включает в себя следующие функции и классы:

1. ``APIRouter``: Класс, который предоставляет функциональность для определения API-роутов. В данном случае, объект ``auth_route`` создается как экземпляр этого класса для группировки роутов аутентификации и авторизации.

2. ``HTTPBearer``: Класс, предоставляемый FastAPI для использования схемы аутентификации по токену Bearer. В данном случае, объект ``security`` создается как экземпляр этого класса для обеспечения аутентификации с помощью токена Bearer в определенных эндпоинтах.

3. ``AuthService``: Класс, который представляет сервис аутентификации и авторизации. Объект ``auth_service`` создается как экземпляр этого класса для выполнения операций аутентификации и авторизации.

4. ``get_db``: Функция, которая устанавливает соединение с базой данных перед выполнением операций. Она используется в качестве зависимости для эндпоинтов, которые требуют доступ к базе данных.

5. Роуты для эндпоинтов:

- ``/local/signup``: POST-запрос для регистрации пользователя с использованием локальных учетных данных. Он принимает объект ``dto`` в качестве тела запроса, содержащий данные пользователя. Возвращает объект ``Tokens``, который содержит информацию о сгенерированных токенах доступа и обновления.

- ``/local/signin``: POST-запрос для аутентификации пользователя с использованием локальных учетных данных. Он принимает объект ``dto`` в качестве тела запроса, содержащий данные пользователя. Возвращает объект ``Tokens``, который содержит информацию о сгенерированных токенах доступа и обновления.

- ``/refresh``: POST-запрос для обновления токенов доступа. Он требует аутентификации с помощью токена Bearer. Возвращает объект ``Tokens``, который содержит новые токены доступа и обновления.

- ``/logout``: POST-запрос для выхода пользователя из системы. Он также требует аутентификации с помощью токена Bearer.

Файл `auth.py` используется для обработки запросов, связанных с аутентификацией и авторизацией пользователей в веб-приложении.

```
# Импорт необходимых модулей и классов из библиотек
from fastapi import APIRouter, Depends, Security, status
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer
from services.auth_service import AuthService
from models import database, user
from schemas import token, user

# Создание экземпляра APIRouter для группировки роутов аутентификации и
авторизации
auth_route = APIRouter()

# Создание экземпляра HTTPBearer для работы с схемой аутентификации Bearer
security = HTTPBearer()

# Создание экземпляра AuthService для выполнения операций аутентификации и
авторизации
auth_service = AuthService()

# Функция для установки соединения с базой данных
def get_db():
    try:
        database.connect()
    finally:
```



```

        if not database.is_closed():
            database.close()

# Роут для регистрации пользователя
@auth_route.post(
    "/local/signup",
    status_code=status.HTTP_201_CREATED,
    response_model=token.Tokens,
    dependencies=[Depends(get_db)],
)
async def signup(dto: user.UserSignUp) -> token.Tokens:
    return auth_service.signup(dto)

# Роут для аутентификации пользователя
@auth_route.post(
    "/local/signin",
    status_code=status.HTTP_200_OK,
    response_model=token.Tokens,
    dependencies=[Depends(get_db)],
)
async def signin(dto: user.UserSignIn) -> token.Tokens:
    return auth_service.signin(dto)

# Роут для обновления токенов доступа
@auth_route.post(
    "/refresh",
    status_code=status.HTTP_200_OK,
    response_model=token.Tokens,
    dependencies=[Depends(get_db)],
)
async def refresh(
    credentials: HTTPAuthorizationCredentials = Security(security),
) -> token.Tokens:
    return auth_service.refresh_tokens(credentials.credentials)

# Роут для выхода пользователя из системы
@auth_route.post(
    "/logout",
    status_code=status.HTTP_200_OK,
    dependencies=[Depends(get_db)],
)
async def logout(
    credentials: HTTPAuthorizationCredentials = Security(security),
):
    auth_service.logout(credentials.credentials)

```

2) Рассмотрим данный папку `dialog.py`

Файл ``dialog.py`` является модулем, содержащим определение маршрутов (endpoints) для работы с диалогами в API. В этом файле определены два маршрута:

1. ``get_dialog`` - GET-запрос для получения информации о конкретном диалоге по его идентификатору (``id``). Этот маршрут требует авторизации с использованием токена, передаваемого в заголовке запроса. Если токен валидный, то вызывается метод ``get_dialog`` из сервиса ``DialogService``, который возвращает информацию о диалоге. В противном случае возвращается значение ``None``.

2. ``get_dialogs`` - GET-запрос для получения списка диалогов пользователя. Этот маршрут также требует авторизации с использованием токена. Если токен валидный, то вызывается метод ``get_dialogs`` из сервиса ``DialogService``, который возвращает список диалогов пользователя. В противном случае возвращается значение ``None``.

Оба маршрута имеют зависимость ``get_db``, которая отвечает за установку и закрытие соединения с базой данных.

Использование модулей ``APIRouter``, ``Depends``, ``HTTPException`` и других классов и функций из библиотеки FastAPI позволяет создать эффективный и структурированный код для обработки запросов связанных с диалогами.

```
from fastapi import APIRouter, Depends, HTTPException, Security, status
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer
from services.dialog_service import DialogService
from schemas.dialog import DialogOut
from services.auth_service import AuthService
from models import database

# Создание экземпляра маршрутизатора API
dialog_route = APIRouter()

# Создание экземпляра класса HTTPBearer для обеспечения аутентификации с
использованием Bearer-токена
security = HTTPBearer()

# Создание экземпляра класса AuthService для работы с аутентификацией
auth_service = AuthService()

# Создание экземпляра класса DialogService для работы с диалогами
dialog_service = DialogService()
```

```

# Функция зависимости для подключения к базе данных
def get_db():
    try:
        # Установка соединения с базой данных
        database.connect()
    finally:
        if not database.is_closed():
            # Закрытие соединения с базой данных, если оно было открыто
            database.close()

# Маршрут для получения информации о конкретном диалоге
@dialog_route.get(
   ("/{id}",
    status_code=status.HTTP_200_OK,
    response_model=DialogOut | None,
    dependencies=[Depends(get_db)],
)
async def get_dialog(
    id: str,
    cred: HTTPAuthorizationCredentials = Security(security),
) -> DialogOut | None:
    try:
        dialog_id = int(id)
        if auth_service.decode_token(cred.credentials):
            # Вызов метода get_dialog() из DialogService для получения информации
            # о диалоге
            return dialog_service.get_dialog(dialog_id)
        return None
    except ValueError:
        # Генерация исключения HTTPException в случае некорректного значения
        # идентификатора диалога
        raise HTTPException(400, "ID param it's not number")

# Маршрут для получения списка диалогов пользователя
@dialog_route.get(
    "/",
    status_code=status.HTTP_200_OK,
    response_model=list[DialogOut] | None,
    dependencies=[Depends(get_db)],
)
async def get_dialogs(
    cred: HTTPAuthorizationCredentials = Security(security),
) -> list[DialogOut] | None:
    try:
        user_id = int(auth_service.decode_token(cred.credentials)["id"])
        if auth_service.decode_token(cred.credentials):
            # Вызов метода get_dialogs() из DialogService для получения списка
            # диалогов пользователя
            return dialog_service.get_dialogs(user_id)
        return None

```

```
except ValueError:
    # Генерация исключения HTTPException в случае некорректного значения
    # идентификатора пользователя
    raise HTTPException(400, "ID param it's not number")
```

3) Рассмотрим следующий файл `message.py`

Этот файл `message.py` содержит роутер API для обработки запросов, связанных с сообщениями.

В данном файле определены следующие функции-обработчики запросов:

1. `get_dialog(dialog_id: str, cred: HTTPAuthorizationCredentials)`: Обработывает GET-запрос для получения сообщений из диалога. Принимает параметр `dialog_id` в качестве пути и проверяет авторизацию пользователя с помощью токена `cred`. Если пользователь авторизован, возвращает список сообщений, иначе возвращает `None`.
2. `delete_history(dialog_id: str, cred: HTTPAuthorizationCredentials)`: Обработывает DELETE-запрос для удаления истории сообщений из диалога. Принимает параметр `dialog_id` в качестве пути и проверяет авторизацию пользователя с помощью токена `cred`. Если пользователь авторизован, удаляет историю сообщений и возвращает список удаленных сообщений, иначе возвращает `None`.
3. `websocket(websocket: WebSocket, user_id: int)`: Обработывает WebSocket-запрос для установления соединения с веб-сокетом. Принимает объект `WebSocket` и `user_id` в качестве пути. Используется для обработки веб-сокет сообщений между клиентом и сервером.

Кроме того, в файле также импортируются классы, схемы и сервисы из других модулей, которые необходимы для работы с сообщениями.

```
from fastapi import APIRouter, Depends, HTTPException, Security, WebSocket,
status
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer
from schemas.message import MessageOut
from services.message_service import MessageService
from services.auth_service import AuthService
from models import database

message_router = APIRouter() # Создание роутера API
```

```

security = HTTPBearer() # Инициализация HTTPBearer для аутентификации

auth_service = AuthService() # Инициализация сервиса аутентификации
message_service = MessageService() # Инициализация сервиса сообщений

def get_db(): # Функция для получения доступа к базе данных
    try:
        database.connect() # Устанавливаем соединение с базой данных
    finally:
        if not database.is_closed(): # Проверяем не закрыта ли база данных
            database.close() # Закрываем соединение

@message_router.get(
   ("/{dialog_id}",
    status_code=status.HTTP_200_OK,
    response_model=list[MessageOut] | None,
    dependencies=[Depends(get_db)],
)
async def get_dialog(
    dialog_id: str,
    cred: HTTPAuthorizationCredentials = Security(security),
) -> list[MessageOut] | None:
    try:
        did = int(dialog_id) # Преобразование строки dialog_id в целое число
        if auth_service.decode_token(cred.credentials): # Проверка валидности
            токена аутентификации
            return message_service.get_messages(did) # Возвращает список
            сообщений
        return None
    except ValueError:
        raise HTTPException(400, "ID param it's not number") # Генерируем
        исключение при некорректном dialog_id

@message_router.delete(
   ("/{dialog_id}",
    status_code=status.HTTP_200_OK,
    response_model=list[MessageOut] | None,
    dependencies=[Depends(get_db)],
)
async def delete_history(
    dialog_id: str,
    cred: HTTPAuthorizationCredentials = Security(security),
) -> list[MessageOut] | None:
    try:
        did = int(dialog_id) # Преобразование строки dialog_id в целое число
        if auth_service.decode_token(cred.credentials): # Проверка валидности
            токена аутентификации
            return message_service.delete_history(did) # Удаляет историю
            сообщений
        return None

```

```

except ValueError:
    raise HTTPException(400, "ID param it's not number") # Генерируем
исключение при некорректном dialog_id

@message_router.websocket(
    "/ws/{user_id}",
    dependencies=[Depends(get_db)],
)
async def websocket(websocket: WebSocket, user_id: int):
    await message_service.websocket(websocket, user_id) # Обрабатывает веб-сокеты
сообщения между клиентом и сервером

```

4) Рассмотрим следующий файл user.py

Файл user.py используется для определения маршрутов (routes) и функций-обработчиков, связанных с пользователями (users) в веб-приложении на основе FastAPI.

В этом файле определен API-роутер (APIRouter) с именем "user_route", который содержит два маршрута:

1. Маршрут "/":

- HTTP метод: GET
- О функции-обработчике:
 - Зависимости (dependencies): функция "get_db" (используется для подключения к базе данных).
 - Параметр запроса: "cred" (HTTPAuthorizationCredentials), содержащий авторизационные данные пользователя.
 - Если токен успешно расшифрован с использованием сервиса аутентификации (AuthService), то возвращается список пользователей, полученных с помощью сервиса пользователя (UserService). Если расшифровка токена не удалась, возвращается значение None.

2. Маршрут "/self":

- HTTP метод: GET
- О функции-обработчике:
 - Зависимости (dependencies): функция "get_db" (используется для подключения к базе данных).

- Параметр запроса: "cred" (HTTPAuthorizationCredentials), содержащий авторизационные данные пользователя.

- Если токен успешно расшифрован с использованием сервиса аутентификации (AuthService), то из расшифрованного токена извлекается идентификатор пользователя. Затем вызывается функция "get_user_self" сервиса пользователя (UserService) с этим идентификатором, чтобы получить информацию о самом пользователе. Если расшифровка токена не удалась, возвращается значение None.

Файл user.py связывает маршруты и функции-обработчики с соответствующими зависимостями, моделями данных и схемами ответов, которые объявлены в других файлах (например, "schemas.user" и "services.user_service"). Это помогает организовать код и управлять логикой обработки запросов, связанных с пользователями, в приложении.

```
# Импорт необходимых модулей и классов
from fastapi import APIRouter, Depends, Security, status
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer
from schemas.user import UserOut
from services.auth_service import AuthService
from services.user_service import UserService
from models import database

# Создание экземпляра роутера APIRouter
user_route = APIRouter()

# Создание экземпляра HTTPBearer для аутентификации
security = HTTPBearer()

# Создание экземпляра сервиса аутентификации (AuthService)
auth_service = AuthService()

# Создание экземпляра сервиса пользователя (UserService)
user_service = UserService()

# Определение функции для подключения к базе данных
def get_db():
    try:
        # Подключение к базе данных
        database.connect()
    finally:
        if not database.is_closed():
            # Закрытие соединения с базой данных
            database.close()

# Определение маршрута для получения списка пользователей
@user_route.get(
    "/",
```

```

        status_code=status.HTTP_200_OK,
        response_model=list[UserOut] | None,
        dependencies=[Depends(get_db)],
    )
    async def users(
        cred: HTTPAuthorizationCredentials = Security(security),
    ) -> list[UserOut] | None:
        # Расшифровка токена с использованием сервиса аутентификации
        if auth_service.decode_token(cred.credentials):
            # Получение списка пользователей с помощью сервиса пользователя
            return user_service.get_users()
        return None

# Определение маршрута для получения информации о текущем пользователе
@user_route.get(
    "/self",
    status_code=status.HTTP_200_OK,
    response_model=UserOut | None,
    dependencies=[Depends(get_db)],
)
    async def self(
        cred: HTTPAuthorizationCredentials = Security(security),
    ) -> UserOut | None:
        # Расшифровка токена с использованием сервиса аутентификации
        if auth_service.decode_token(cred.credentials):
            # Извлечение идентификатора пользователя из расшифрованного токена
            user_id = auth_service.decode_token(cred.credentials)["id"]
            # Получение информации о самом пользователе с помощью сервиса
            пользователя
            return user_service.get_user_self(user_id)
        return None

```


3) Рассмотрим файлы внутри папки schemas (схемы)

1) Рассмотрим следующий файл dialog.py

Файл `dialog.py` содержит определение модели данных `DialogOut`, которая используется для представления информации о диалоге между пользователями. Она имеет следующие поля: `id` (идентификатор диалога) и `first_user` и `second_user`, которые представляют информацию о пользователях, участвующих в диалоге. Поле `first_user` и `second_user` являются экземплярами модели `UserOut` из файла `user.py`, которая определена в отдельном модуле. Эти поля содержат информацию о пользователе.

```
from pydantic import BaseModel

from schemas.user import UserOut

class DialogOut(BaseModel): #Здесь импортируется класс UserOut из модуля
schemas.user. В файле schemas.user определена модель данных для пользователя.
    id: int # поле типа int, представляющее идентификатор диалога.
    first_user: UserOut #поле типа UserOut, представляющее информацию о первом
пользователе.
    second_user: UserOut #поле типа UserOut, представляющее информацию о втором
пользователе.
```

2) Рассмотрим следующий файл message.py

Файл message.py содержит определение модели сообщения (MessageOut), которая используется для структурирования и валидации данных о сообщении. Эта модель имеет три поля:

sender_id: целое число, представляющее идентификатор отправителя сообщения;

content: строка, содержащая текст сообщения;

created_at: объект datetime, указывающий дату и время создания сообщения.

Использование BaseModel из пакета pydantic позволяет определить структуру модели и автоматически выполнять проверку данных на соответствие этой структуре. Таким образом, модель MessageOut может использоваться для создания объектов сообщений с корректными данными и обеспечения их согласованности при обработке в других частях программы.

```

from datetime import datetime

from pydantic import BaseModel

from schemas.user import UserOut # Импортируем модель UserOut из модуля
schemas.user

class MessageOut(BaseModel): # Определяем класс MessageOut, наследуемый от
BaseModel
    sender_id: int # Поле sender_id типа int
    content: str # Поле content типа str
    created_at: datetime # Поле created_at типа datetime

```

3) Рассмотрим следующий файл token.py

Файл `token.py` используется для определения модели данных, связанных с токенами доступа и обновления.

В данном коде:

- Импортируется класс `BaseModel` из модуля `pydantic`, который используется для определения моделей данных.
- Определяется класс `Tokens`, наследующийся от `BaseModel`. Этот класс представляет модель данных для токенов доступа и обновления.
- Внутри класса `Tokens` определены два поля:
 - `access_token` - строковое поле, представляющее токен доступа.
 - `refresh_token` - строковое поле, представляющее токен обновления.

Файл `token.py` позволяет структурировать и использовать данные, связанные с токенами, в более удобном и типобезопасном формате, благодаря использованию `pydantic`.

```

from pydantic import BaseModel #используется для определения моделей данных,
которые могут содержать поля со своими типами и валидацией.

class Tokens(BaseModel): #определяется новый класс Tokens, который наследуется от
BaseModel. Этот класс будет представлять модель данных для токенов.
    access_token: str #Это объявления полей в классе Tokens. Здесь определены два
поля access_token и refresh_token
    refresh_token: str#которые являются типом str, то есть строками. Эти поля
будут хранить значения токенов доступа и токенов обновления соответственно.

```

4) Рассмотрим следующий файл user.py

Файл user.py содержит определение моделей данных, связанных с пользователями, используемых в приложении. Он предоставляет структуры данных для работы с пользователями, такие как модели для регистрации новых пользователей (UserSignUp), входа в систему (UserSignIn), а также вывод информации о пользователе (UserOut).

Файл user.py определяет базовую модель пользователя (UserBase), которая содержит основные поля, такие как имя, фамилия, адрес электронной почты и псевдоним.

Модель UserSignIn используется для проверки учетных данных пользователя при входе в систему, содержит поля для логина и пароля.

Модель UserSignUp наследуется от базовой модели пользователя (UserBase) и добавляет поле для пароля. Она используется при регистрации новых пользователей.

Модель UserOut также наследуется от базовой модели пользователя (UserBase) и добавляет поле для идентификатора пользователя. Эта модель используется для представления информации о пользователе при выводе.

Таким образом, файл user.py предоставляет структуры данных, необходимые для работы с пользователями в приложении.

```
from pydantic import BaseModel

class UserBase(BaseModel):
    first_name: str
    last_name: str
    email: str
    nickname: str

class UserSignIn(BaseModel):
    login: str
    password: str

class UserSignUp(UserBase):
    password: str

class UserOut(UserBase):
    id: int
```

4) Разберём файлы внутри папки services (службы)

1) Рассмотрим следующий файл auth_service.py

Файл `auth_service.py` представляет собой модуль, который отвечает за аутентификацию и авторизацию пользователей. Он содержит класс `AuthService`, который предоставляет методы для регистрации, входа, обновления токенов и выхода из системы.

Вот краткое описание каждого метода в классе `AuthService`:

- `signup(dto) -> Tokens`: Метод для регистрации нового пользователя. Принимает данные пользователя (`dto`) и выполняет необходимую валидацию. Если данные валидны, создает нового пользователя и генерирует пару токенов доступа и обновления. Возвращает объект `Tokens`, содержащий сгенерированные токены.

- `signin(dto) -> Tokens`: Метод для входа пользователя в систему. Принимает данные пользователя (`dto`) и выполняет проверку правильности введенных данных. Если данные верны, генерирует пару токенов доступа и обновления. Возвращает объект `Tokens`, содержащий сгенерированные токены.

- `refresh_tokens(refresh_token: str) -> Tokens`: Метод для обновления токенов доступа и обновления. Принимает существующий refresh-токен и проверяет его валидность. Если токен действителен, обновляет токен доступа и генерирует новый refresh-токен. Возвращает объект `Tokens`, содержащий обновленные токены.

- `logout(token)`: Метод для выхода пользователя из системы. Принимает токен доступа и удаляет соответствующий refresh-токен из базы данных.

Класс `AuthService` также содержит приватные методы для работы с шифрованием паролей и токенов, кодирования и декодирования токенов, а также для обновления информации о refresh-токене у пользователя.

```
import os
from typing import TypedDict
from dotenv import load_dotenv

import jwt
from fastapi import HTTPException
from passlib.context import CryptContext

from datetime import datetime, timedelta
from models.user import User
```

```

from schemas.token import Tokens
from services.user_service import UserService

load_dotenv()

# Определение типа словаря для полезной нагрузки токена
class Payload(TypedDict):
    id: int
    username: str

class AuthService:
    # Создание экземпляра хешера bcrypt для хэширования паролей
    _hasher_bcrypt = CryptContext(schemes=["bcrypt"])
    # Создание экземпляра хешера sha256 для хэширования токенов
    _hasher_sha256 = CryptContext(schemes=["sha256_crypt"])
    # Получение секретного ключа из переменных окружения
    _secret = os.getenv("JWT_SECRET") or ""
    # Создание экземпляра сервиса пользователей
    _user_service = UserService()

    def signup(self, dto) -> Tokens:
        # Валидация данных пользователя
        if (
            dto.email == ""
            or dto.nickname == ""
            or dto.first_name == ""
            or dto.last_name == ""
            or dto.password == ""
        ):
            raise HTTPException(400, "Some fields are empty")

        # Проверка наличия пользователя с таким же email
        if self._user_service.get_user_by_email(dto.email):
            raise HTTPException(400, "Email already exist")

        # Проверка наличия пользователя с таким же ником
        if self._user_service.get_user_by_nickname(dto.nickname):
            raise HTTPException(400, "Nickname already exist")

        # Создание нового пользователя
        new_user = self._user_service.create_user(
            dto, self._get_password_hash(dto.password)
        )

        # Генерация токенов доступа и обновления
        access_token = self._encode_token(new_user.id, str(new_user.email))
        refresh_token = self._encode_refresh_token(new_user.id,
            str(new_user.email))

```

```

# Обновление refresh-токена у пользователя
self._update_user_refresh_token(new_user, refresh_token)

# Возврат сгенерированных токенов
return Tokens(access_token=access_token, refresh_token=refresh_token)

def signin(self, dto) -> Tokens:
    # Поиск пользователя по email или никнейму
    user_on_email = self._user_service.get_user_by_email(dto.login)
    user_on_nickname = self._user_service.get_user_by_nickname(dto.login)

    # Проверка наличия пользователя
    user = user_on_email or user_on_nickname
    if not user:
        raise HTTPException(400, "Invalid login")

    # Проверка правильности введенного пароля
    if not self._verify_password(dto.password, str(user.password_hash)):
        raise HTTPException(400, "Invalid password")

    # Генерация токенов доступа и обновления
    access_token = self._encode_token(user.id, str(user.email))
    refresh_token = self._encode_refresh_token(user.id, str(user.email))

    # Обновление refresh-токена у пользователя
    self._update_user_refresh_token(user, refresh_token)

    # Возврат сгенерированных токенов
    return Tokens(access_token=access_token, refresh_token=refresh_token)

def refresh_tokens(self, refresh_token: str) -> Tokens:
    # Проверка наличия пользователя по email, указанному в токене
    user = self._user_service.get_user_by_email(
        self._decode_refresh_token(refresh_token)["username"]
    )

    # Проверка наличия пользователя
    if not user:
        raise HTTPException(404, "User not found")

    # Проверка валидности refresh-токена
    if user.token and not self._verify_tokens(refresh_token,
str(user.token)):
        self._update_user_refresh_token(user, None)
        raise HTTPException(401, "Invalid refresh token")

    # Обновление токена доступа и генерация нового refresh-токена
    access_token = self._refresh_token(refresh_token)
    refresh_token = self._encode_refresh_token(user.id, str(user.email))

```

```

        # Обновление refresh-токена у пользователя
        self._update_user_refresh_token(user, refresh_token)

        # Возврат обновленных токенов
        return Tokens(access_token=access_token, refresh_token=refresh_token)

def logout(self, token):
    # Получение пользователя по идентификатору, полученному из токена
    user = self._user_service.get_user_by_id(self.decode_token(token)["id"])
    if user:
        # Удаление refresh-токена у пользователя
        self._update_user_refresh_token(user, None)

def _get_password_hash(self, password: str) -> str:
    # Хэширование пароля с использованием хешера sha256
    return self._hasher_sha256.hash(password)

def _get_token_hash(self, token: str) -> str:
    # Хэширование токена с использованием хешера bcrypt
    return self._hasher_bcrypt.hash(token)

def _verify_password(self, password: str, encoded_password: str) -> bool:
    # Проверка правильности пароля с использованием хешера sha256
    return self._hasher_sha256.verify(password, encoded_password)

def _verify_tokens(self, token: str, encoded_token: str) -> bool:
    # Проверка правильности токенов с использованием хешера bcrypt
    return self._hasher_bcrypt.verify(token, encoded_token)

def _encode_token(self, user_id: int, username: str) -> str:
    # Генерация токена с помощью библиотеки JWT
    payload = {
        "exp": datetime.utcnow() + timedelta(days=0, minutes=20),
        "iat": datetime.utcnow(),
        "scope": "access_token",
        "sub": user_id,
        "username": username,
    }
    return jwt.encode(payload, self._secret, algorithm="HS256")

def decode_token(self, token: str) -> Payload:
    # Расшифровка токена и проверка его валидности
    try:
        payload = jwt.decode(token, self._secret, algorithms=["HS256"])

        if payload["scope"] == "access_token":
            return {"id": payload["sub"], "username": payload["username"]}
        raise HTTPException(
            status_code=401, detail="Scope for the token is invalid"
        )
    
```

```

except jwt.ExpiredSignatureError:
    raise HTTPException(status_code=401, detail="Token expired")

except jwt.InvalidTokenError:
    raise HTTPException(status_code=401, detail="Invalid token")

def _decode_refresh_token(self, token: str) -> Payload:
    # Расшифровка refresh-токена и проверка его валидности
    try:
        payload = jwt.decode(token, self._secret, algorithms=["HS256"])

        if payload["scope"] == "refresh_token":
            return {"id": payload["sub"], "username": payload["username"]}
        raise HTTPException(
            status_code=401, detail="Scope for the token is invalid"
        )

    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token expired")

    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail="Invalid token")

def _encode_refresh_token(self, user_id: int, username: str) -> str:
    # Генерация refresh-токена с помощью библиотеки JWT
    payload = {
        "exp": datetime.utcnow() + timedelta(days=14, hours=0),
        "iat": datetime.utcnow(),
        "scope": "refresh_token",
        "sub": user_id,
        "username": username,
    }
    return jwt.encode(payload, self._secret, algorithm="HS256")

def _refresh_token(self, refresh_token: str) -> str:
    # Обновление токена доступа на основе refresh-токена
    try:
        payload = jwt.decode(refresh_token, self._secret,
algorithms=["HS256"])

        if payload["scope"] == "refresh_token":
            user_id = payload["sub"]
            username = payload["username"]
            new_token = self._encode_token(user_id, username)
            return new_token
        raise HTTPException(status_code=401, detail="Invalid scope for
token")

    except jwt.ExpiredSignatureError:

```



```

        raise HTTPException(status_code=401, detail="Refresh token expired")

    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail="Invalid refresh token")

    def _update_user_refresh_token(self, user: User, refresh_token: str | None):
        user: User = User.get(User.id == user.id)
        if not user:
            raise HTTPException(404, "User not found")
        user.token = self._get_token_hash(refresh_token) if refresh_token else
None
        # TODO: if needed here to add last active
        user.save()

```

2) Рассмотрим следующий файл cipher_service.py

Файл cipher_service.py отвечает за шифрование и дешифрование сообщений с помощью симметричного алгоритма шифрования.

```

import os
from cryptography.fernet import Fernet

class CipherService:
    def __init__(self):
        # Получение ключа из системной переменной окружения CIPHER_KEY
        # Если переменная окружения не установлена, будет использован пустой ключ
        key = os.getenv("CIPHER_KEY") or ""
        # Создание экземпляра класса Fernet с использованием ключа
        self._fernet: Fernet = Fernet(key)

    def encrypt_message(self, message: str) -> str:
        # Шифрование сообщения с использованием Fernet и кодирование в строку
        return self._fernet.encrypt(message.encode()).decode()

    def decrypt_message(self, message: str) -> str:
        # Дешифрование сообщения с использованием Fernet и декодирование в строку
        return self._fernet.decrypt(message).decode()

```

3) Рассмотрим следующий файл connection_manager.py

Файл `connection_manager.py` представляет собой класс `ConnectionManager`, который служит для управления соединениями WebSocket. Он предоставляет несколько методов для установки, разрыва и отправки сообщений через WebSocket.

Этот класс имеет следующие методы:

1. `connect(self, websocket: WebSocket, user_id)`: Метод, вызываемый при подключении нового WebSocket. Он проверяет, не существует ли уже активного соединения для данного `user_id`. Если активное соединение уже существует, он принимает новое соединение, а затем закрывает его с кодом ошибки 4000. Если активного соединения нет, он принимает новое соединение и добавляет его в словарь `active_connections` под ключом `user_id`. Метод возвращает `True`, если соединение было успешно установлено, и `False`, если было обнаружено активное соединение.

2. `disconnect(self, websocket: WebSocket)`: Метод, вызываемый при разрыве WebSocket-соединения. Он удаляет соответствующее соединение из словаря `active_connections`, исходя из переданного объекта `websocket`.

3. `send_personal_message(self, message: str, websocket: WebSocket)`: Метод для отправки персонального сообщения через WebSocket. Он отправляет текстовое сообщение `message` через указанное соединение `websocket`.

4. `send_user_message(self, message: str, user_id: int)`: Метод для отправки сообщения определенному пользователю с помощью WebSocket. Он получает соединение для указанного `user_id` из словаря `active_connections` и отправляет сообщение через это соединение.

5. `broadcast(self, message: str)`: Метод для отправки сообщения всем активным соединениям через WebSocket. Он итерирует по словарю `active_connections` и отправляет сообщение `message` через каждое соединение.

Этот класс полезен при создании сервера FastAPI, который поддерживает WebSocket-соединения и требует управления активными соединениями.

```
from fastapi import WebSocket

class ConnectionManager:
    def __init__(self):
        # Создание пустого словаря для хранения активных соединений
        self.active_connections = {}

    async def connect(self, websocket: WebSocket, user_id):
        # Метод для подключения нового WebSocket-соединения
        if user_id in list(self.active_connections.keys()):
            # Проверка, существует ли активное соединение для данного user_id
            await websocket.accept()
            await websocket.close(code=4000)
```

```

        return False

    await websocket.accept()
    # Принятие нового WebSocket-соединения
    self.active_connections[user_id] = websocket
    # Добавление соединения в словарь active_connections под ключом user_id

    return True

def disconnect(self, websocket: WebSocket):
    # Метод для разрыва WebSocket-соединения
    del self.active_connections[
        list(
            filter(
                lambda x: self.active_connections[x] == websocket,
                self.active_connections,
            )
        )[0]
    ]
    # Удаление соединения из active_connections на основе переданного объекта
websocket

    async def send_personal_message(self, message: str, websocket: WebSocket):
        # Метод для отправки персонального сообщения через WebSocket
        await websocket.send_text(message)
        # Отправка текстового сообщения message через указанное соединение
websocket

    async def send_user_message(self, message: str, user_id: int):
        # Метод для отправки сообщения определенному пользователю с помощью
WebSocket
        websocket = self.active_connections[user_id]
        # Получение соединения для указанного user_id из active_connections
        await websocket.send_text(message)
        # Отправка текстового сообщения message через полученное соединение

    async def broadcast(self, message: str):
        # Метод для отправки сообщения всем активным соединениям через WebSocket
        for _user_id, connection in self.active_connections.items():
            # Итерация по парам (user_id, connection) в active_connections
            await connection.send_text(message)
            # Отправка текстового сообщения message через каждое соединение

```

4) Рассмотрим следующий файл `dialog_service.py`

Файл `dialog_service.py` используется для реализации сервисной функциональности связанной с диалогами пользователей. В нём определен класс `DialogService`, который содержит методы для работы с диалогами.

Методы класса `DialogService` включают:

1. `get_dialogs(self, user_id: int) -> list[DialogOut]`: Возвращает список диалогов, принадлежащих пользователю по указанному `user_id`. Для каждого диалога проверяется, является ли пользователь участником диалога, и возвращается список объектов `DialogOut` - представление диалога в формате, который будет возвращен из API.

2. `get_dialog(self, dialog_id: int) -> DialogOut | None`: Возвращает информацию о диалоге с указанным `dialog_id`. Если диалог не найден, вызывается исключение `HTTPException` с кодом 404. Иначе возвращается объект `DialogOut`, представляющий информацию о диалоге.

3. `create_dialog(self, first_user_id: int, second_user_id: int) -> DialogOut | None`: Создает новый диалог между двумя пользователями с указанными `first_user_id` и `second_user_id`. Если один из пользователей не найден, вызывается исключение `HTTPException` с кодом 404. Возвращается объект `DialogOut`, представляющий информацию о созданном диалоге.

4. `get_dialog_out(self, dialog: Dialog)`: Вспомогательный метод, который преобразует объект `Dialog` в объект `DialogOut`. Используется для создания представления диалога при получении списка диалогов или информации о диалоге.

Это основные методы для работы с диалогами пользователей.

```
from fastapi import HTTPException # Импорт класса HTTPException из модуля
fastapi
from models.user import User # Импорт класса User из модуля models.user
from schemas.dialog import DialogOut # Импорт класса DialogOut из модуля
schemas.dialog
from services.user_service import UserService # Импорт класса UserService из
модуля services.user_service
from models.dialog import Dialog # Импорт класса Dialog из модуля models.dialog

class DialogService:
    _user_service = UserService() # Создание экземпляра класса UserService и
присваивание его атрибуту _user_service
```

```

def get_dialogs(self, user_id: int) -> list[DialogOut]:
    print(user_id) # Вывод значения переменной user_id в консоль

    dialogs = Dialog.select() # Выбор всех записей из таблицы Dialog

    dialogs_out = [] # Создание пустого списка
    for dialog in dialogs: # Итерация по каждому элементу dialogs
        if dialog.first_user_id == user_id or dialog.second_user_id ==
user_id: # Проверка, является ли пользователь участником диалога
            dialogs_out.append(self.get_dialog_out(dialog)) # Добавление
представления диалога в список dialogs_out

    return dialogs_out # Возврат списка dialogs_out

def get_dialog(self, dialog_id: int) -> DialogOut | None:
    dialog = Dialog.get(Dialog.id == dialog_id) # Получение диалога с
указанным dialog_id

    if not dialog: # Если диалог не найден
        raise HTTPException(404, "Dialog not found") # Вызов исключения
HTTPException с кодом 404

    return DialogOut(
        id=dialog.id,
        first_user=self._user_service.get_user_out(dialog.first_user),
        second_user=self._user_service.get_user_out(dialog.second_user),
    ) # Возврат объекта DialogOut, представляющего информацию о диалоге

def create_dialog(
    self, first_user_id: int, second_user_id: int
) -> DialogOut | None:
    first_user = User.get(User.id == first_user_id) # Получение пользователя
с указанным first_user_id
    if not first_user: # Если пользователь не найден
        raise HTTPException(404, "First user not found") # Вызов исключения
HTTPException с кодом 404

    second_user = User.get(User.id == second_user_id) # Получение
пользователя с указанным second_user_id
    if not second_user: # Если пользователь не найден
        raise HTTPException(404, "Second user not found") # Вызов исключения
HTTPException с кодом 404

    self.get_dialog_out(
        Dialog.create(
            first_user=first_user,
            second_user=second_user,
        )
    ) # Создание нового диалога и вызов метода get_dialog_out для получения
его представления

```

```
def get_dialog_out(self, dialog: Dialog):
    return DialogOut(
        id=dialog.id,
        first_user=self._user_service.get_user_out(dialog.first_user),
        second_user=self._user_service.get_user_out(dialog.second_user),
    ) # Создание объекта DialogOut, представляющего информацию о диалоге
```

5) Рассмотрим следующий файл `message_service.py`

Файл `message_service.py` является модулем, который содержит реализацию класса `MessageService`, предоставляющего функциональность для работы с сообщениями и взаимодействия с WebSocket.

Вот основные функции и методы, которые предоставляет `MessageService`:

1. `get_messages(dialog_id: int) -> list[MessageOut]`: Этот метод получает список сообщений для заданного диалога (`dialog_id`) и возвращает список объектов `MessageOut`, содержащих информацию о сообщениях.
2. `get_message_out(message: Message) -> MessageOut`: Этот метод конвертирует объект `Message` в объект `MessageOut`, который содержит информацию о сообщении, готовую для отправки.
3. `delete_history(chat_id)`: Этот метод удаляет историю сообщений для заданного чата (`chat_id`), включая связанные объекты `DialogMessage` и `Message`.
4. `new_message(data) -> Message`: Этот метод создает новое сообщение на основе переданных данных (`data`) и возвращает созданный объект `Message`. Он также создает или находит соответствующий диалог (`Dialog`) и создает связь с сообщением (`DialogMessage`).
5. `async def websocket(websocket: WebSocket, user_id: int)`: Этот асинхронный метод обрабатывает WebSocket соединение и обрабатывает входящие сообщения. Он принимает объект `WebSocket` и идентификатор пользователя (`user_id`), проверяет соединение, и затем внутри цикла `while` принимает текстовые сообщения (`rec`). Разбирает полученные данные в формате JSON и создает новое сообщение с использованием `new_message()`. Затем сообщение широковещательно отправляется всем клиентам, подключенным к WebSocket.

Этот модуль также импортирует необходимые зависимости, такие как классы и методы из других модулей, такие как `CipherService`, `ConnectionManager`, `UserService` и схемы данных, используемые для сериализации и десериализации сообщений.

```
import json # Импорт модуля json для работы с JSON
from fastapi import WebSocket, WebSocketDisconnect # Импорт классов WebSocket и
WebSocketDisconnect из модуля fastapi
from models.message import Message # Импорт класса Message из модуля
models.message
from models.dialog import Dialog # Импорт класса Dialog из модуля models.dialog
from services.cipher_service import CipherService # Импорт класса CipherService
из модуля services.cipher_service
from services.connection_manager import ConnectionManager # Импорт класса
ConnectionManager из модуля services.connection_manager
from services.user_service import UserService # Импорт класса UserService из
модуля services.user_service
from schemas.message import MessageOut # Импорт класса MessageOut из модуля
schemas.message
from models.dialog_message import DialogMessage # Импорт класса DialogMessage из
модуля models.dialog_message
from datetime import datetime # Импорт класса datetime из модуля datetime

class MessageService:
    manager = ConnectionManager() # Создание экземпляра класса ConnectionManager
и присваивание его атрибуту manager
    _user_service = UserService() # Создание экземпляра класса UserService и
присваивание его атрибуту _user_service
    keys = ["sender_id", "reciver_id", "content"] # Задание списка ключей keys
    _cipher_service = CipherService() # Создание экземпляра класса CipherService
и присваивание его атрибуту _cipher_service

    def get_messages(self, dialog_id: int) -> list[MessageOut]:
        out = [] # Создание пустого списка
        # Выбор всех сообщений в диалоге с указанным dialog_id, присоединение
таблицы Message и сортировка по времени создания
        messages = DialogMessage.select().where(DialogMessage.dialog_id ==
dialog_id).join(Message).order_by(DialogMessage.message.created_at)
        for message in messages: # Итерация по каждому сообщению
            out.append(self.get_message_out(message.message)) # Добавление
представления сообщения в список out
        return out # Возврат списка out

    def get_message_out(self, message: Message) -> MessageOut:
        return MessageOut(
            sender_id=message.sender.id,
            content=self._cipher_service.decrypt_message(str(message.content)),
# Расшифровка содержимого сообщения
            created_at=message.created_at,
```

```

        ) # Создание объекта MessageOut, представляющего информацию о сообщении

    def delete_history(self, chat_id):
        dialog_messages = DialogMessage.select() # Выбор всех записей из таблицы DialogMessage

        for dialog_message in dialog_messages: # Итерация по каждой записи
            if dialog_message.dialog.id == chat_id: # Проверка, принадлежит ли запись указанному chat_id
                q = DialogMessage.delete().where(DialogMessage.id == dialog_message.id) # Удаление записи из таблицы DialogMessage
                q.execute()
                q = Message.delete().where(Message.id == dialog_message.message.id) # Удаление связанного сообщения из таблицы Message
                q.execute()

        q = Dialog.delete().where(Dialog.id == chat_id) # Удаление диалога с указанным chat_id
        q.execute()

    def new_message(self, data) -> Message:
        sender_id = data["sender_id"] # Получение sender_id из данных
        reciver_id = data["reciver_id"] # Получение reciver_id из данных
        content = data["content"] # Получение content из данных

        sender_user = self._user_service.get_user_by_id(sender_id) # Получение пользователя по указанному sender_id
        message = Message.create(
            sender=sender_user,
            content=self._cipher_service.encrypt_message(content), created_at=datetime.now()
        ) # Создание нового сообщения, зашифрованного содержимого и указанного времени создания

        dialog = None
        # FIXME: when peewee developers would be middle level
        for d in Dialog.select(): # Итерация по каждому диалогу
            fi = d.first_user_id
            si = d.second_user_id
            if (sender_id == fi or sender_id == si) and (
                reciver_id == fi or reciver_id == si
            ):
                dialog = d
                break

        if not dialog: # Если диалог не найден
            reciver_user = self._user_service.get_user_by_id(reciver_id) # Получение пользователя по указанному reciver_id

            dialog = Dialog.create(
                first_user=sender_user,

```



```

        second_user=reciver_user,
    ) # Создание нового диалога между отправителем и получателем

    DialogMessage.create(
        dialog=dialog,
        message=message,
    ) # Создание связи между диалогом и сообщением

    return message # Возврат созданного сообщения

    async def websocket(self, websocket: WebSocket, user_id: int):
        if await self.manager.connect(websocket, int(user_id)) == False: #
Подключение websocket к менеджеру соединений
            return

        try:
            while True:
                rec = await websocket.receive_text() # Получение текстового
сообщения от websocket
                data: dict = json.loads(rec) # Преобразование текстового
сообщения в словарь

                if self.keys == list(data.keys()): # Проверка, что все ключи из
self.keys присутствуют в данных
                    message: Message = self.new_message(data) # Создание нового
сообщения на основе данных
                else:
                    print("JSON is invalid")
                    continue

                await
self.manager.broadcast(f"{self.get_message_out(message).json()}") # Рассылка
представления сообщения всем подключенным клиентам

        except WebSocketDisconnect: # Если websocket отключается
            self.manager.disconnect(websocket) # Отключение websocket от
менеджера соединений

```

6) Рассмотрим следующий файл `user_service.py`

Файл `'user_service.py'` определяет класс `'UserService'`, который предоставляет различные методы для работы с пользователями. Вот некоторые из основных функций и их назначение:

- `'get_users'`: Получает список всех пользователей и возвращает их в виде списка объектов `'UserOut'`.

- `'get_user_by_email'`: Получает пользователя по указанному электронному адресу (`'email'`) и возвращает соответствующий объект `'User'`. Если пользователя с таким адресом не существует, возвращается значение `'None'`.

- `'get_user_by_nickname'`: Получает пользователя по указанному никнейму (`'nickname'`) и возвращает соответствующий объект `'User'`. Если пользователя с таким никнеймом не существует, возвращается значение `'None'`.

- `'get_user_by_id'`: Получает пользователя по указанному идентификатору (`'id'`) и возвращает соответствующий объект `'User'`. Если пользователя с таким идентификатором не существует, возвращается значение `'None'`.

- `'create_user'`: Создает нового пользователя на основе предоставленных данных (`'UserSignUp'`) и зашифрованного пароля. Возвращает созданный объект `'User'`.

- `'get_user_self'`: Получает информацию о текущем пользователе на основе его идентификатора (`'id'`). Если пользователь существует, возвращает его информацию в виде объекта `'UserOut'`. Если пользователь не найден, вызывается исключение `'HTTPException'` с кодом 404 и сообщением "Self user not found".

- `'get_user_out'`: Преобразует объект `'User'` в объект `'UserOut'`, содержащий только определенные данные о пользователе (идентификатор, имя, фамилию, электронный адрес, никнейм).

Файл `'user_service.py'` предоставляет набор функций для работы с пользователями, таких как получение информации о пользователях, создание новых пользователей и поиск пользователей по различным параметрам.

```
from fastapi import HTTPException
from schemas.user import UserOut, UserSignUp
from models.user import User
import peewee
```

```

class UserService:
    def get_users(self) -> list[UserOut]:
        # Получение списка всех пользователей
        users_out = []
        for user in User.select():
            users_out.append(self.get_user_out(user))
        return users_out

    def get_user_by_email(self, email: str) -> User | None:
        # Получение пользователя по электронной почте
        try:
            return User.get(User.email == email)
        except peewee.DoesNotExist:
            return None

    def get_user_by_nickname(self, nickname: str) -> User | None:
        # Получение пользователя по псевдониму (никнейму)
        try:
            return User.get(User.nickname == nickname)
        except peewee.DoesNotExist:
            return None

    def get_user_by_id(self, id: int) -> User | None:
        # Получение пользователя по идентификатору
        try:
            return User.get(User.id == id)
        except peewee.DoesNotExist:
            return None

    def create_user(self, dto: UserSignUp, encoded_password: str) -> User:
        # Создание нового пользователя
        return User.create(
            first_name=dto.first_name,
            last_name=dto.last_name,
            email=dto.email,
            nickname=dto.nickname,
            password_hash=encoded_password,
        )

    def get_user_self(self, id) -> UserOut | None:
        # Получение информации о текущем пользователе
        user = self.get_user_by_id(id)

        if user:
            return self.get_user_out(user)

        # Если пользователь не найден, возбуждается исключение
        raise HTTPException(404, "Self user not found")

    def get_user_out(self, user: User) -> UserOut:

```

```
# Преобразование объекта пользователя в объект UserOut
return UserOut(
    id=int(user.id),
    first_name=str(user.first_name),
    last_name=str(user.last_name),
    email=str(user.email),
    nickname=str(user.nickname),
)
```

После создания серверной части и всех файлов в ней, прочитаем немного теории.

Докер

Еще несколько релизов назад запуск Докера на OS X и Windows был проблемным. Но команда разработчиков проделала огромную работу, и сегодня весь процесс — проще некуда. Этот tutorial *getting started* включает в себя подробные инструкции по установке на [Мак](#), [Linux](#) и [Windows](#).

Проверим, все ли установлено корректно:

```
$ docker run hello-world
```

```
Hello from Docker.
```

```
This message shows that your installation appears to be working correctly.
```

```
...
```

Проверьте версию:

```
$ python --version  
Python 3.11
```

Играем с Busybox

Теперь, когда все необходимое установлено, пора взяться за работу. В этом разделе мы запустим контейнер [Busybox](#) на нашей системе и попробуем запустить `docker run`.

Для начала, запустите следующую команду:

```
$ docker pull busybox
```

Внимание: в зависимости от того, как вы устанавливали Докер на свою систему, возможно появление сообщения `permission denied`. Если вы на Маке, то удостоверьтесь, что движок Докер запущен. Если вы на Линуксе, то запустите эту команду с `sudo`. Или можете [создать группу docker](#) чтобы избавиться от этой проблемы.

Команда `pull` скачивает образ `busybox` из [регистра Докера](#) и сохраняет его локально. Можно использовать команду `docker images`, чтобы посмотреть список образов в системе.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
busybox              latest             c51f86c28340       4 weeks ago
1.109 MB
```

1.1 Docker Run

Отлично! Теперь давайте запустим **Докер-контейнер** с этим образом. Для этого используем волшебную команду `docker run`:

```
$ docker run busybox
$
```

Подождите, ничего не произошло! Это баг? Ну, нет. Под капотом произошло много всего. Докер-клиент нашел образ (в нашем случае, `busybox`), загрузил контейнер и запустил команду внутри этого контейнера. Мы сделали `docker run busybox`, но не указали никаких команд, так что контейнер загрузился, запустилась пустая команда и программа завершилась. Ну, да, как-то обидно, так что давайте сделаем что-то поинтереснее.

```
$ docker run busybox echo "hello from busybox"
hello from busybox
```

Ура, наконец-то какой-то вывод. В нашем случае клиент Докера послушно запустил команду `echo` внутри контейнера, а потом вышел из него. Вы, наверное, заметили, что все произошло очень быстро. А теперь представьте себе, как нужно загружать виртуальную машину, запускать в ней команду и выключать ее. Теперь ясно, почему говорят, что контейнеры быстрые!

Теперь давайте взглянем на команду `docker ps`. Она выводит на экран список всех запущенных контейнеров.

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	

Контейнеров сейчас нет, поэтому выводится пустая строка. Не очень полезно, поэтому давайте запустим более полезный вариант: `docker ps -a`

```
$ docker ps -a
```

CONTAINER ID	IMAGE	PORTS	COMMAND	CREATED
STATUS			NAMES	
305297d7a235	busybox		"uptime"	11 minutes ago
Exited (0) 11 minutes ago			distracted_goldstine	
ff0a5c3750b9	busybox		"sh"	12 minutes ago
Exited (0) 12 minutes ago			elated_ramanujan	

Теперь виден список всех контейнеров, которые мы запускали. В колонке `STATUS` можно заметить, что контейнеры завершили свою работу несколько минут назад.

Вам, наверное, интересно, как запустить больше одной команды в контейнере. Давайте попробуем:

```
$ docker run -it busybox sh
/ # ls
bin  dev  etc  home  proc  root  sys  tmp  usr  var
/ # uptime
05:45:21 up 5:58, 0 users, load average: 0.00, 0.01, 0.04
```

Команда `run` с флагом `-it` подключает интерактивный `tty` в контейнер. Теперь можно запускать сколько угодно много команд внутри. Попробуйте.

Опасно! Если хочется острых ощущений, то можете попробовать `rm -rf bin` в контейнере. Но удостоверьтесь, что запускаете ее внутри контейнера, а **не снаружи**. Если сделаете это снаружи, на своем компьютере, то будет очень плохо, и команды вроде `ls`, `echo` перестанут работать. Когда внутри контейнера все перестанет работать, просто выйдете и запустите его заново командой `docker run -it busybox sh`. Докер создает новый контейнер при запуске, поэтому все заработает снова.

На этом захватывающий тур по возможностям команды `docker run` закончен. Скорее всего, вы будете использовать эту команду довольно часто. Так что важно, чтобы мы поняли как с ней обращаться. Чтобы узнать больше о `run`, используйте `docker run --help`, и увидите полный список поддерживаемых флагов. Скоро мы увидим еще несколько способов использования `docker run`.

Перед тем, как продолжать, давайте вкратце рассмотрим удаление контейнеров. Мы видели выше, что с помощью команды `docker ps -a` все еще можно увидеть остатки завершенных контейнеров. На протяжении этого пособия, вы будете запускать `docker run` несколько раз, и оставшиеся, бездомные контейнеры будут съедать дисковое пространство. Так что я взял за правило удалять контейнеры после завершения работы с ними. Для этого используется команда `docker rm`. Просто скопируйте ID (можно несколько) из вывода выше и передайте параметрами в команду.

```
$ docker rm 305297d7a235 ff0a5c3750b9
305297d7a235
ff0a5c3750b9
```

При удалении идентификаторы будут снова выведены на экран. Если нужно удалить много контейнеров, то вместо ручного копирования и вставления можно сделать так:


```
$ docker rm $(docker ps -a -q -f status=exited)
```

Эта команда удаляет все контейнеры, у которых статус `exited`. Флаг `-q` возвращает только численные ID, а флаг `-f` фильтрует вывод на основе предоставленных условий. Последняя полезная деталь — команде `docker run` можно передать флаг `--rm`, тогда контейнер будет автоматически удаляться при завершении. Это очень полезно для разовых запусков и экспериментов с Докером.

Также можно удалять ненужные образы командой `docker rmi`.

1.2 Терминология

В предыдущем разделе мы использовали много специфичного для Докера жаргона, и многих это может запутать. Перед тем, как продолжать, давайте разберем некоторые термины, которые часто используются в экосистеме Докера.

- *Images* (образы) - Схемы нашего приложения, которые являются основой контейнеров. В примере выше мы использовали команду `docker pull` чтобы скачать образ **busybox**.
- *Containers* (контейнеры) - Создаются на основе образа и запускают само приложение. Мы создали контейнер командой `docker run`, и использовали образ `busybox`, скачанный ранее. Список запущенных контейнеров можно увидеть с помощью команды `docker ps`.
- *Docker Daemon* (демон Докера) - Фоновый сервис, запущенный на хост-машине, который отвечает за создание, запуск и уничтожение Докер-контейнеров. Демон — это процесс, который запущен на операционной системе, с которой взаимодействует клиент.
- *Docker Client* (клиент Докера) - Утилита командной строки, которая позволяет пользователю взаимодействовать с демоном. Существуют другие формы клиента, например, [Kitematic](#), с графическим интерфейсом.
- *Docker Hub* - [Регистр](#) Докер-образов. Грубо говоря, архив всех доступных образов. Если нужно, то можно содержать собственный регистр и использовать его для получения образов.

Dockerfile

[Dockerfile](#) — это простой текстовый файл, в котором содержится список команд Докер-клиента. Это простой способ автоматизировать процесс создания образа. Самое классное, что [команды](#) в Dockerfile *почти* идентичны своим аналогам в Linux. Это значит, что в принципе не нужно изучать никакой новый синтаксис чтобы начать работать с докерфайлами.

В директории с приложением есть Dockerfile, но так как мы делаем все впервые, нам нужно создать его с нуля. Создайте новый пустой файл в любимом текстовом редакторе, и сохраните его в **той же** директории, где находится flask-приложение. Назовите файл `Dockerfile`.

Для начала укажем базовый образ. Для этого нужно использовать ключевое слово `FROM`.

```
FROM python:3-onbuild
```

Дальше обычно указывают команды для копирования файлов и установки зависимостей. Но к счастью, `onbuild`-версия базового образа берет эти задачи на себя. Дальше нам нужно указать порт, который следует открыть. Наше приложение работает на порту 5000, поэтому укажем его:

```
EXPOSE 5000
```

Последний шаг — указать команду для запуска приложения. Это просто `python ./app.py`. Для этого используем команду [CMD](#):

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

Главное предназначение `CMD` — это сообщить контейнеру какие команды нужно выполнить при старте. Теперь наш `Dockerfile` готов. Вот как он выглядит:

```
# our base image
FROM python:3-onbuild

# specify the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "./app.py"]
```

Теперь можно создать образ. Команда `docker build` занимается сложной задачей создания образа на основе `Dockerfile`.

Листинг ниже демонстрирует процесс. Перед тем, как запустите команду сами (не забудьте точку в конце), проверьте, чтобы там был ваш `username` вместо моего. `Username` должен соответствовать тому, что использовался при регистрации на [Docker hub](#). Если вы еще не регистрировались, то сделайте это до выполнения команды. Команда `docker build` довольно проста: она принимает опциональный `тег` с флагом `-t` и путь до директории, в которой лежит `Dockerfile`.

```
$ docker build -t prakhar1989/catnip .
Sending build context to Docker daemon 8.704 kB
Step 1 : FROM python:3-onbuild
# Executing 3 build triggers...
Step 1 : COPY requirements.txt /usr/src/app/
---> Using cache
Step 1 : RUN pip install --no-cache-dir -r requirements.txt
---> Using cache
Step 1 : COPY . /usr/src/app
---> 1d61f639ef9e
Removing intermediate container 4de6ddf5528c
Step 2 : EXPOSE 5000
---> Running in 12cfcf6d67ee
---> f423c2f179d1
Removing intermediate container 12cfcf6d67ee
Step 3 : CMD python ./app.py
---> Running in f01401a5ace9
---> 13e87ed1fbc2
Removing intermediate container f01401a5ace9
Successfully built 13e87ed1fbc2
```

Если у вас нет образа `python:3-onbuild`, то клиент сначала скачает его, а потом возьмется за создание вашего образа. Так что, вывод на экран может отличаться от моего. Посмотрите внимательно, и найдете триггеры `onbuild`. Если все прошло хорошо, то образ готов! Запустите `docker images` и увидите свой образ в списке.

Последний шаг — запустить образ и проверить его работоспособность (замените `username` на свой):

```
$ docker run -p 8888:5000 prakhar1989/catnip
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Зайдите на указанный URL и увидите приложение в работе.

Docker Compose

До этого момента мы изучали клиент Докера. Но в экосистеме Докера есть несколько других инструментов с открытым исходным кодом, которые хорошо взаимодействуют с Докером. Некоторые из них это:

1. [Docker Machine](#) позволяет создавать Докер-хосты на своем компьютере, облачном провайдере или внутри дата-центра.
2. [Docker Compose](#) — инструмент для определения и запуска много-контейнерных приложений.
3. [Docker Swarm](#) — нативное решение для кластеризации.

В этом разделе мы поговорим об одном из этих инструментов — Docker Compose, и узнаем, как он может упростить работу с несколькими контейнерами.

У Docker Compose довольно интересная предыстория. Примерно два года назад компания OrchardUp запустила инструмент под названием Fig. Идея была в том, чтобы создавать изолированные рабочие окружения с помощью Докера. Проект очень хорошо восприняли на [Hacker News](#) - я смутно помню, что читал о нем, но не особо понял его смысла.

[Первый комментарий](#) на самом деле неплохо объясняет, зачем нужен Fig и что он делает:

На самом деле, смысл Докера в следующем: запускать процессы. Сегодня у Докера есть неплохое API для запуска процессов: расшаренные между контейнерами (иными словами, запущенными образами) разделы или директории (shared volumes), перенаправление портов с хост-машины в контейнер, вывод логов, и так далее. Но больше ничего: Докер сейчас работает только на уровне процессов.

Не смотря на то, что в нем содержатся некоторые возможности оркестрации нескольких контейнеров для создания единого "приложения", в Докере нет ничего, что помогало бы с управлением такими группами контейнеров как одной сущностью. И вот зачем нужен инструмент вроде Fig: чтобы обращаться с группой контейнеров как с единой сущностью. Чтобы думать о "запуске приложений" (иными словами, "запуске оркестрированного кластера контейнеров") вместо "запуска контейнеров".

Оказалось, что многие пользователи Докера согласны с такими мыслями. Постепенно, Fig набрал популярность, Docker Inc. заметили, купили компанию и назвали проект Docker Compose.

Итак, зачем используется *Compose*? Это инструмент для простого определения и запуска многоконтейнерных Докер-приложений. В нем есть файл `docker-compose.yml`, и с его помощью можно одной командой поднять приложение с набором сервисов.

Давайте посмотрим, сможем ли мы создать файл `docker-compose.yml` для нашего приложения SF-Foodtrucks и проверим, способен ли он на то, что обещает.

Но вначале нужно установить Docker Compose. Есть у вас Windows или Mac, то Docker Compose уже установлен — он идет в комплекте с Docker Toolbox. На Linux можно установить Docker Compose следуя [простым инструкциям](#) на сайте документации. Compose написан на Python, поэтому можно сделать просто `pip install docker-compose`. Проверить работоспособность так:

```
$ docker-compose version
docker-compose version 1.7.1, build 0a9ab35
docker-py version: 1.8.1
CPython version: 2.7.9
OpenSSL version: OpenSSL 1.0.1j 15 Oct 2014
```

Теперь можно перейти к следующему шагу, то есть созданию файла `docker-compose.yml`. Синтаксис `yml`-файлов очень простой, и в репозитории уже есть [пример](#), который мы будем использовать

```
version: "2"
services:
  es:
    image: elasticsearch
  web:
    image: prakhar1989/foodtrucks-web
    command: python app.py
    ports:
      - "5000:5000"
    volumes:
      - ./code
```

Давайте я разберу это подробнее. На родительском уровне мы задали название неймспейса для наших сервисов: `es` и `web`. К каждому сервису можно добавить дополнительные параметры, среди которых `image` — обязательный. Для `es` мы указываем доступный на Docker Hub образ `elasticsearch`. Для Flask-приложения — тот образ, который мы создали самостоятельно в начале этого раздела.

С помощью других параметров вроде `command` и `ports` можно предоставить информацию о контейнере. `volumes` отвечает за локацию монтирования, где будет находиться код в контейнере `web`. Это опциональный параметр, он полезен, если нужно обращаться к логам и так далее. Подробнее о параметрах и возможных значениях можно [прочитать в документации](#).

Замечание: Нужно находиться в директории с файлом `docker-compose.yml` чтобы запускать большую часть команд Compose.

Отлично! Файл готов, давайте посмотрим на `docker-compose` в действии. Но вначале нужно удостовериться, что порты свободны. Так что если у вас запущены контейнеры Flask и ES, то пора их остановить:

```
$ docker stop $(docker ps -q)
39a2f5df14ef
2a1b77e066e6
```

Теперь можно запускать `docker-compose`. Перейдите в директорию с приложением Foodtrucks и выполните команду `docker-compose up`.

```
$ docker-compose up
Creating network "foodtrucks_default" with the default driver
Creating foodtrucks_es_1
Creating foodtrucks_web_1
Attaching to foodtrucks_es_1, foodtrucks_web_1
es_1 | [2016-01-11 03:43:50,300][INFO ][node                               ] [Comet]
version[2.1.1], pid[1], build[40e2c53/2015-12-15T13:05:55Z]
es_1 | [2016-01-11 03:43:50,307][INFO ][node                               ] [Comet]
initializing ...
es_1 | [2016-01-11 03:43:50,366][INFO ][plugins                           ] [Comet] loaded
[], sites []
es_1 | [2016-01-11 03:43:50,421][INFO ][env                               ] [Comet] using [1]
data paths, mounts [[/usr/share/elasticsearch/data (/dev/sda1)], net usable_space
[16gb], net total_space [18.1gb], spins? [possibly], types [ext4]
es_1 | [2016-01-11 03:43:52,626][INFO ][node                               ] [Comet]
initialized
es_1 | [2016-01-11 03:43:52,632][INFO ][node                               ] [Comet] starting
...
es_1 | [2016-01-11 03:43:52,703][WARN ][common.network                   ] [Comet] publish
address: {0.0.0.0} is a wildcard address, falling back to first non-loopback:
{172.17.0.2}
es_1 | [2016-01-11 03:43:52,704][INFO ][transport                         ] [Comet]
publish_address {172.17.0.2:9300}, bound_addresses {[:]:9300}
es_1 | [2016-01-11 03:43:52,721][INFO ][discovery                         ] [Comet]
elasticsearch/cEk4s7pdQ-evRc9MqS2wqw
es_1 | [2016-01-11 03:43:55,785][INFO ][cluster.service                  ] [Comet]
new_master {Comet}{cEk4s7pdQ-evRc9MqS2wqw}{172.17.0.2}{172.17.0.2:9300}, reason: zen-
disco-join(elected_as_master, [0] joins received)
es_1 | [2016-01-11 03:43:55,818][WARN ][common.network                   ] [Comet] publish
address: {0.0.0.0} is a wildcard address, falling back to first non-loopback:
{172.17.0.2}
es_1 | [2016-01-11 03:43:55,819][INFO ][http                             ] [Comet]
publish_address {172.17.0.2:9200}, bound_addresses {[:]:9200}
es_1 | [2016-01-11 03:43:55,819][INFO ][node                             ] [Comet] started
es_1 | [2016-01-11 03:43:55,826][INFO ][gateway                           ] [Comet] recovered
[0] indices into cluster_state
es_1 | [2016-01-11 03:44:01,825][INFO ][cluster.metadata                 ] [Comet] [sfdata]
creating index, cause [auto(index api)], templates [], shards [5]/[1], mappings
[truck]
es_1 | [2016-01-11 03:44:02,373][INFO ][cluster.metadata                 ] [Comet] [sfdata]
update_mapping [truck]
es_1 | [2016-01-11 03:44:02,510][INFO ][cluster.metadata                 ] [Comet] [sfdata]
update_mapping [truck]
es_1 | [2016-01-11 03:44:02,593][INFO ][cluster.metadata                 ] [Comet] [sfdata]
update_mapping [truck]
```

```
es_1 | [2016-01-11 03:44:02,708][INFO ][cluster.metadata      ] [Comet] [sfdata]
update_mapping [truck]
es_1 | [2016-01-11 03:44:03,047][INFO ][cluster.metadata      ] [Comet] [sfdata]
update_mapping [truck]
web_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Перейдите по IP чтобы увидеть приложение. Круто, да? Всего лишь пара строк конфигурации и несколько Docker-контейнеров работают в унисон. Давайте остановим сервисы и перезапустим в detached mode:

```
web_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
Killing foodtrucks_web_1 ... done
Killing foodtrucks_es_1 ... done

$ docker-compose up -d
Starting foodtrucks_es_1
Starting foodtrucks_web_1

$ docker-compose ps
```

Name	Command	State	Ports
foodtrucks_es_1	/docker-entrypoint.sh elas ...	Up	9200/tcp, 9300/tcp
foodtrucks_web_1	python app.py	Up	0.0.0.0:5000->5000/tcp

Не удивительно, но оба контейнера успешно запущены. Откуда берутся имена? Их Compose придумал сам. Но что насчет сети? Его Compose тоже делаем сам? Хороший вопрос, давайте выясним.

Для начала, остановим запущенные сервисы. Их всегда можно вернуть одной командой:

```
$ docker-compose stop
Stopping foodtrucks_web_1 ... done
Stopping foodtrucks_es_1 ... done
```

Заодно, давайте удалим сеть `foodtrucks`, которую создали в прошлый раз. Эта сеть нам не потребуется, потому что *Compose* автоматически сделает все за нас.


```
$ docker network rm foodtrucks
$ docker network ls
NETWORK ID          NAME                DRIVER
4eec273c054e        bridge             bridge
9347ae8783bd        none              null
54df57d7f493        host              host
```

Класс! Теперь в этом чистом состоянии можно проверить, способен ли *Compose* на волшебство.

```
$ docker-compose up -d
Recreating foodtrucks_es_1
Recreating foodtrucks_web_1
$ docker ps
CONTAINER ID        IMAGE                                     COMMAND                  CREATED
STATUS            PORTS                               NAMES
f50bb33a3242      prakhar1989/foodtrucks-web            "python app.py"         14 seconds
ago                Up 13 seconds                      0.0.0.0:5000->5000/tcp   foodtrucks_web_1
e299ceeb4caa      elasticsearch                          "/docker-entrypoint.s"  14 seconds
ago                Up 14 seconds                      9200/tcp, 9300/tcp      foodtrucks_es_1
```

Пока все хорошо. Проверим, создались ли какие-нибудь сети:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
0c8b474a9241        bridge             bridge
293a141faac3        foodtrucks_default bridge
b44db703cd69        host              host
0474c9517805        none              null
```

Видно, что *Compose* самостоятельно создал сеть `foodtrucks_default` и подсоединил оба сервиса в эту сеть, так, чтобы они могли общаться друг с другом. Каждый контейнер для сервиса подключен к сети, и оба контейнера доступны другим контейнерам в сети. Они доступны по `hostname`, который совпадает с названием контейнера. Давайте проверим, находится ли эта информация в `/etc/hosts`.

```
$ docker ps
CONTAINER ID        IMAGE                               COMMAND                  CREATED
STATUS            PORTS                            NAMES
bb72dcebd379      prakhar1989/foodtrucks-web        "python app.py"         20 hours
ago                Up 19 hours                      0.0.0.0:5000->5000/tcp   foodtrucks_web_1
3338fc79be4b      elasticsearch                      "/docker-entrypoint.s"  20 hours
ago                Up 19 hours                      9200/tcp, 9300/tcp      foodtrucks_es_1

$ docker exec -it bb72dcebd379 bash
root@bb72dcebd379:/opt/flask-app# cat /etc/hosts
127.0.0.1    localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.18.0.2  bb72dcebd379
```

Упс! Оказывается, файл понятия не имеет о `es`. Как же наше приложение работает? Давайте попингуем его по названию хоста:

```
root@bb72dcebd379:/opt/flask-app# ping es
PING es (172.18.0.3) 56(84) bytes of data.
64 bytes from foodtrucks_es_1.foodtrucks_default (172.18.0.3): icmp_seq=1 ttl=64
time=0.049 ms
64 bytes from foodtrucks_es_1.foodtrucks_default (172.18.0.3): icmp_seq=2 ttl=64
time=0.064 ms
^C
--- es ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.049/0.056/0.064/0.010 ms
```

Вуаля! Работает! Каким-то магическим образом контейнер смог сделать пинг хоста `es`. Оказывается, Docker 1.10 добавили новую сетевую систему, которая производит обнаружение сервисов через DNS-сервер. Если интересно, то почитайте подробнее о [предложении](#) и [release notes](#).

На этом наш тур по Docker Compose завершен. С этим инструментом можно ставить сервисы на паузу, запускать отдельные команды в контейнере и даже масштабировать систему, то есть увеличивать количество контейнеров. Также советую изучить некоторые другие [примеры](#) использования Docker Compose.

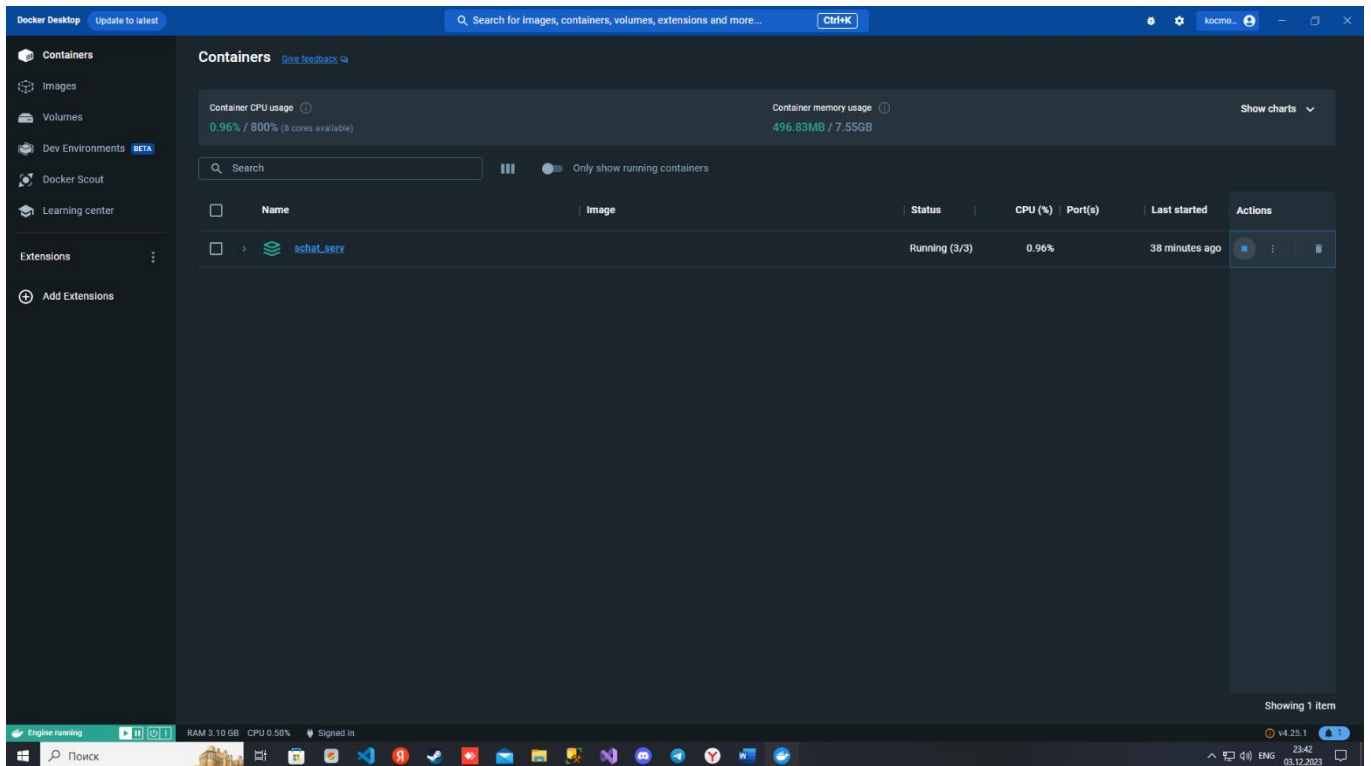
Надеюсь, я продемонстрировал как на самом деле просто управлять многоконтейнерной средой с Compose. В последнем разделе мы задеплоим все на AWS!

Заключение

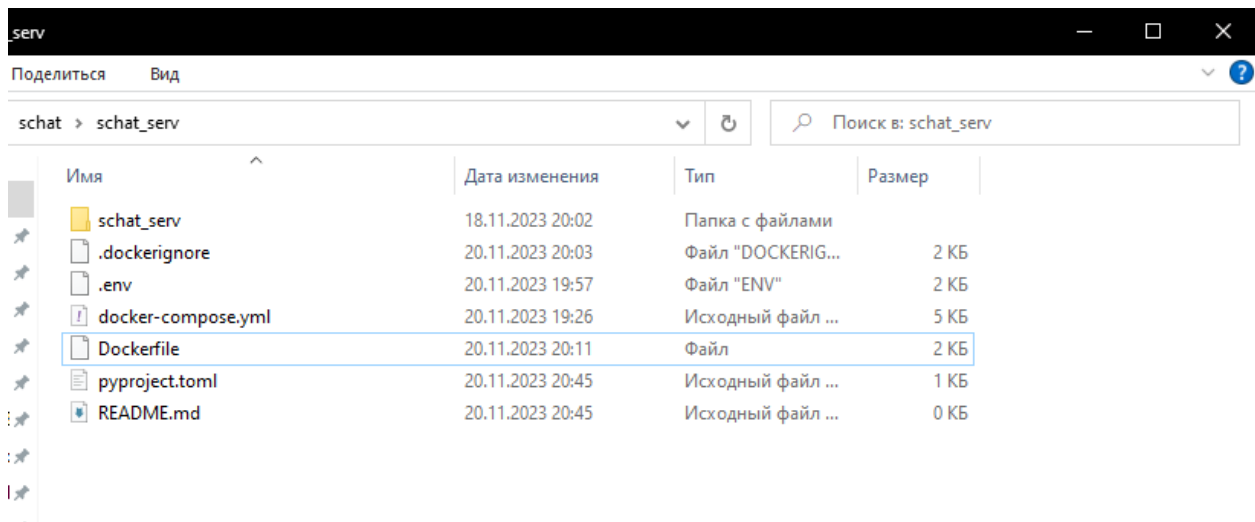
Мы подошли к концу. После длинного, изматывающего, но интересного пособия вы готовы захватить мир контейнеров! Если вы следовали пособию до самого конца, то можете заслуженно гордиться собой. Вы научились устанавливать Докер, запускать свои контейнеры, запускать статические и динамические веб-сайты и, самое главное, получили опыт деплоя приложений в облако.

Надеюсь, прохождение этого руководства помогло вам стать увереннее в своих способностях управляться с серверами. Когда у вас появится новая идея для сайта или приложения, можете быть уверены, что сможете показать его людям с минимальными усилиями.

Как же сделать запуск контейнера `schat_serv`?



В папке с сервером лежит файл с именем `Dockerfile` – это наша инструкция по запуску контейнера, напомним её.



Напишем следующий код для работы

```
Файл  Правка  Поиск  Вид  Кодировки  Синтаксисы  Опции  Инструменты  Макросы  Запуск  Плагины  Вкладки  ?
FROM python:3.11
# Базовый образ Docker, который будет использоваться для создания контейнера. В данном случае, используется образ Python версии 3.11.
RUN pip install poetry
# Выполняется команда "pip install poetry", чтобы установить инструмент управления зависимостями Poetry внутри Docker-образа.
COPY . /app
# Копируются все файлы и папки из текущего каталога (где находится Dockerfile) в папку "/app" внутри Docker-образа.
WORKDIR /app
# Устанавливается рабочая директория внутри Docker-образа как "/app". Все последующие команды будут выполняться относительно этой директории.
RUN poetry install
# Выполняется команда "poetry install" для установки зависимостей проекта с помощью инструмента Poetry внутри Docker-образа.
EXPOSE 3000
# Эта строка объявляет порт 3000 в Docker-контейнере, чтобы он был доступен извне при запуске контейнера.
CMD ["poetry", "run", "python", "schat_serv/app.py"]
# Эта строка задает команду, которая будет выполняться при запуске Docker-контейнера. Она использует инструмент Poetry для запуска команды "python schat_serv/app.py" внутри контейнера.
```

FROM python:3.11

Базовый образ Docker, который будет использоваться для создания контейнера. В данном случае, используется образ Python версии 3.11.

RUN pip install poetry

Выполняется команда "pip install poetry", чтобы установить инструмент управления зависимостями Poetry внутри Docker-образа.

COPY ./app

Копируются все файлы и папки из текущего каталога (где находится Dockerfile) в папку "/app" внутри Docker-образа.

WORKDIR /app

Устанавливается рабочая директория внутри Docker-образа как "/app". Все последующие команды будут выполняться относительно этой директории.

RUN poetry install

Выполняется команда "poetry install" для установки зависимостей проекта с помощью инструмента Poetry внутри Docker-образа.








EXPOSE 3000

Эта строка объявляет порт 3000 в Docker-контейнере, чтобы он был доступен извне при запуске контейнера.

CMD ["poetry", "run", "python", "schat_serv/app.py"]

Эта строка задает команду, которая будет выполняться при запуске Docker-контейнера. Она использует инструмент Poetry для запуска команды "python schat_serv/app.py" внутри контейнера.

Теперь нужно создать ещё один файл с названием `docker-compose.yml` он будет отвечать за конфигурацию для Docker Compose.

chat > schat_serv	▼	↺	🔍 Поиск в: schat_serv
Имя	Дата изменения	Тип	Размер
 schat_serv	18.11.2023 20:02	Папка с файлами	
 .dockerignore	20.11.2023 20:03	Файл "DOCKERIG...	2 КБ
 .env	20.11.2023 19:57	Файл "ENV"	2 КБ
 docker-compose.yml	20.11.2023 19:26	Исходный файл ...	5 КБ
 Dockerfile	20.11.2023 20:11	Файл	2 КБ
 pyproject.toml	20.11.2023 20:45	Исходный файл ...	1 КБ
 README.md	20.11.2023 20:45	Исходный файл ...	0 КБ

```
1 version: '3.3' # Версию формата файла конфигурации Docker Compose.
2 services: # Список сервисов, которые будут запущены.
3   api: #Сервис с именем "api".
4     build: . #Сборка контейнера будет выполнена из текущей директории.
5     container_name: schat_api #Устанавливает имя контейнера в "schat_api".
6     ports:
7       - "3000:3000" # Пропускает порт 3000 контейнера на порт 3000 хоста.
8     depends_on:
9       - db # Устанавливает зависимость от сервиса с именем "db", то есть контейнер "api" будет запущен только после запуска контейнера "db".
10    env_file:
11      - .env #файл с переменными окружения (.env), который будет использоваться контейнером "api".
12    networks:
13      - fnt # Подключает контейнер "api" к сети с именем "fnt".
14
15  db: # сервис с именем "db".
16    image: mysql # Использует официальный образ MySQL из Docker Hub.
17    container_name: schat_db # Устанавливает имя контейнера в "schat_db".
18    environment: #Устанавливает переменные окружения для контейнера MySQL.
19      MYSQL_ROOT_PASSWORD: ${DB_PASSWORD} # такие как пароль администратора (MYSQL_ROOT_PASSWORD),
20      MYSQL_DATABASE: ${DB_NAME} # имя базы данных (MYSQL_DATABASE),
21      MYSQL_USER: ${DB_USER} # имя пользователя (MYSQL_USER)
22      MYSQL_PASSWORD: ${DB_PASSWORD} # пароль пользователя (MYSQL_PASSWORD) || Значения переменных берутся из файла окружения.
23    ports:
24      - "3307:3306" # Пропускает порт 3306 контейнера на порт 3307 хоста.
25    volumes:
26      - dbdata:/var/lib/mysql # Создает именованный том "dbdata" и монтирует его в директорию /var/lib/mysql контейнера, чтобы сохранить данные MySQL между перезапусками контейнера.
27    env_file:
28      - ./schat_serv/.env # Указывает на файл с переменными окружения (./schat_serv/.env), который будет использоваться контейнером "db".
29    networks:
30      - fnt # Подключает контейнер "db" к сети с именем "fnt".
31
32  phpmyadmin: # Определяет сервис с именем "phpmyadmin".
33    image: phpmyadmin/phpmyadmin # Использует официальный образ phpMyAdmin из Docker Hub.
34    container_name: schat_pma # Устанавливает имя контейнера в "schat_pma".
35    links:
36      - db #Устанавливает связь с контейнером "db", чтобы phpMyAdmin мог подключиться к базе данных MySQL.
37    environment: #Устанавливает переменные окружения для контейнера phpMyAdmin,
38      PMA_HOST: schat_db #такие как хост базы данных (PMA_HOST)
39      PMA_PORT: 3306 #порт базы данных (PMA_PORT)
40      PMA_ARBITRARY: 1 #флаг PMA_ARBITRARY для отключения проверки подключения.
41    restart: always #командное перезапускание контейнера phpMyAdmin в случае его остановки.
42    ports: #Пропускает порт 80 контейнера на порт 8081 хоста.
43      - 8081:80
44    networks: #Подключает контейнер "phpmyadmin" к сети с именем "fnt".
45      - fnt
46    volumes: #Определяет именованный том "dbdata".
47      dbdata:
48
49  networks: #Определяет сеть с именем "fnt".
50    fnt:
51
52
```

version: '3.3' # Версию формата файла конфигурации Docker Compose.

services: # Список сервисов, которые будут запущены.

api: #Сервис с именем "api".

build: . #Сборка контейнера будет выполнена из текущей директории.

container_name: schat_api #Устанавливает имя контейнера в "schat_api".

ports:

- "3000:3000" # Пропускает порт 3000 контейнера на порт 3000 хоста.

depends_on:

- db # Устанавливает зависимость от сервиса с именем "db", то есть контейнер "api" будет запущен только после запуска контейнера "db".

env_file:

- .env #файл с переменными окружения (.env), который будет использоваться контейнером "api".

networks:

- fnt # Подключает контейнер "api" к сети с именем "fnt".

db: # сервис с именем "db".

image: mysql # Использует официальный образ MySQL из Docker Hub.

container_name: schat_db # Устанавливает имя контейнера в "schat_db".

environment: # Устанавливает переменные окружения для контейнера MySQL,

MYSQL_ROOT_PASSWORD: \${DB_PASSWORD} # такие как пароль администратора (MYSQL_ROOT_PASSWORD),

MYSQL_DATABASE: \${DB_NAME} # имя базы данных (MYSQL_DATABASE),

MYSQL_USER: \${DB_USER} # имя пользователя (MYSQL_USER)

MYSQL_PASSWORD: \${DB_PASSWORD} # пароль пользователя (MYSQL_PASSWORD) || Значения переменных берутся из файла окружения.

ports:

- "3307:3306" # Пробрасывает порт 3306 контейнера на порт 3307 хоста.

volumes:

- dbdata:/var/lib/mysql # Создает именованный том "dbdata" и монтирует его в директорию /var/lib/mysql контейнера, чтобы сохранить данные MySQL между перезапусками контейнера.

env_file:

- ./schat_serv/.env # Указывает на файл с переменными окружения (/schat_serv/.env), который будет использоваться контейнером "db".

networks:

- fnt # Подключает контейнер "db" к сети с именем "fnt".

phpmyadmin: # Определяет сервис с именем "phpmyadmin".

image: phpmyadmin/phpmyadmin # Использует официальный образ phpMyAdmin из Docker Hub.

container_name: schat_pma # Устанавливает имя контейнера в "schat_pma".

links:

- db # Устанавливает связь с контейнером "db", чтобы phpMyAdmin мог подключиться к базе данных MySQL.

environment: # Устанавливает переменные окружения для контейнера phpMyAdmin,

PMA_HOST: schat_db # такие как хост базы данных (PMA_HOST)

PMA_PORT: 3306 # порт базы данных (PMA_PORT)

PMA_ARBITRARY: 1 # флаг PMA_ARBITRARY для отключения проверки подключения.

restart: always # автоматическое перезапускание контейнера phpMyAdmin в случае его остановки.

ports: # Пробрасывает порт 80 контейнера на порт 8081 хоста.

- 8081:80

networks: # Подключает контейнер "phpmyadmin" к сети с именем "fnt".

- fnt

volumes: # Определяет именованный том "dbdata".

dbdata:

networks: # Определяет сеть с именем "fnt".

fnt:

Убедись, что файлы `.dockerignore`, `.env` и `pyproject.toml` размещены в той же папке, где находятся `Dockerfile` и `docker-compose.yml`. Файл `.dockerignore` используется для указания файлов и папок, которые необходимо игнорировать при копировании в контейнер.

Теперь перейди в директорию с проектом в консоли и выполните следующие команды:

docker-compose build # Создание образа контейнера

docker-compose up # Запуск контейнера

```
MINGW64/c/Users/sereg/OneDrive/Рабочий стол/snup/server
sereg@DESKTOP-1L5030N MINGW64 ~/OneDrive/Рабочий стол/snup/server
$ docker-compose build
#0 building with "default" instance using docker driver

#1 [api internal] load build definition from Dockerfile
#1 transferring dockerfile: 1.77kB done
#1 DONE 0.1s

#2 [api internal] load .dockerignore
#2 transferring context: 1.96kB 0.0s done
#2 DONE 0.1s

#3 [api internal] load metadata for docker.io/library/python:3.11
#3 ...

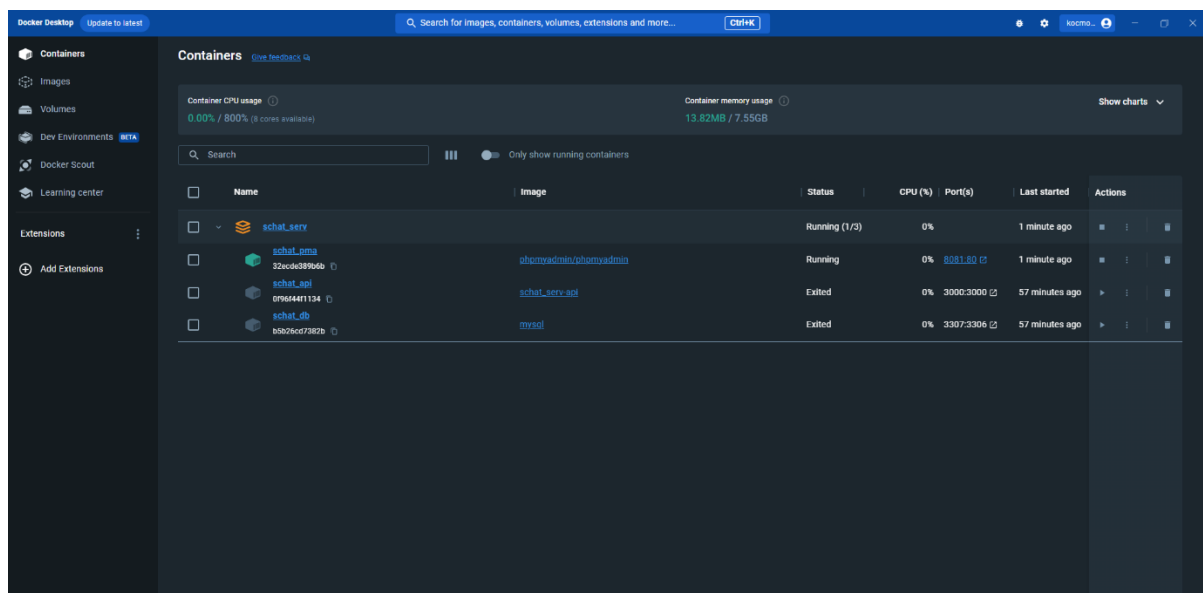
#4 [api auth] library/python:pull token for registry-1.docker.io
#4 DONE 0.0s

#3 [api internal] load metadata for docker.io/library/python:3.11
#3 DONE 2.6s

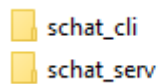
#5 [api 1/5] FROM docker.io/library/python:3.11@sha256:2a725c9721f737a2944244c98c714d24f8bcfadd9f5c15083cbaa024f7fce54
```

Теперь у нас появился контейнер.

Когда будут созданы все необходимые файлы, то увидим работу сервера.



Теперь займёмся разработкой клиентской части.



Заходим в папку `schat_cli` и создаём все необходимые файлы.

schat > schat_cli >					Поиск в: schat_cli	
	Имя	Дата изменения	Тип	Размер		
	schat_cli	19.11.2023 14:55	Папка с файлами			
	poetry.lock	18.11.2023 20:44	Файл "LOCK"	54 КБ		
	pyproject.toml	19.11.2023 14:55	Исходный файл ...	1 КБ		
	README.md	19.11.2023 14:55	Исходный файл ...	0 КБ		

Для начала опишу вот эти файлики, которые у нас лежат в папке, и для чего они.