

Ссылка на проект на **GitHub**:

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.

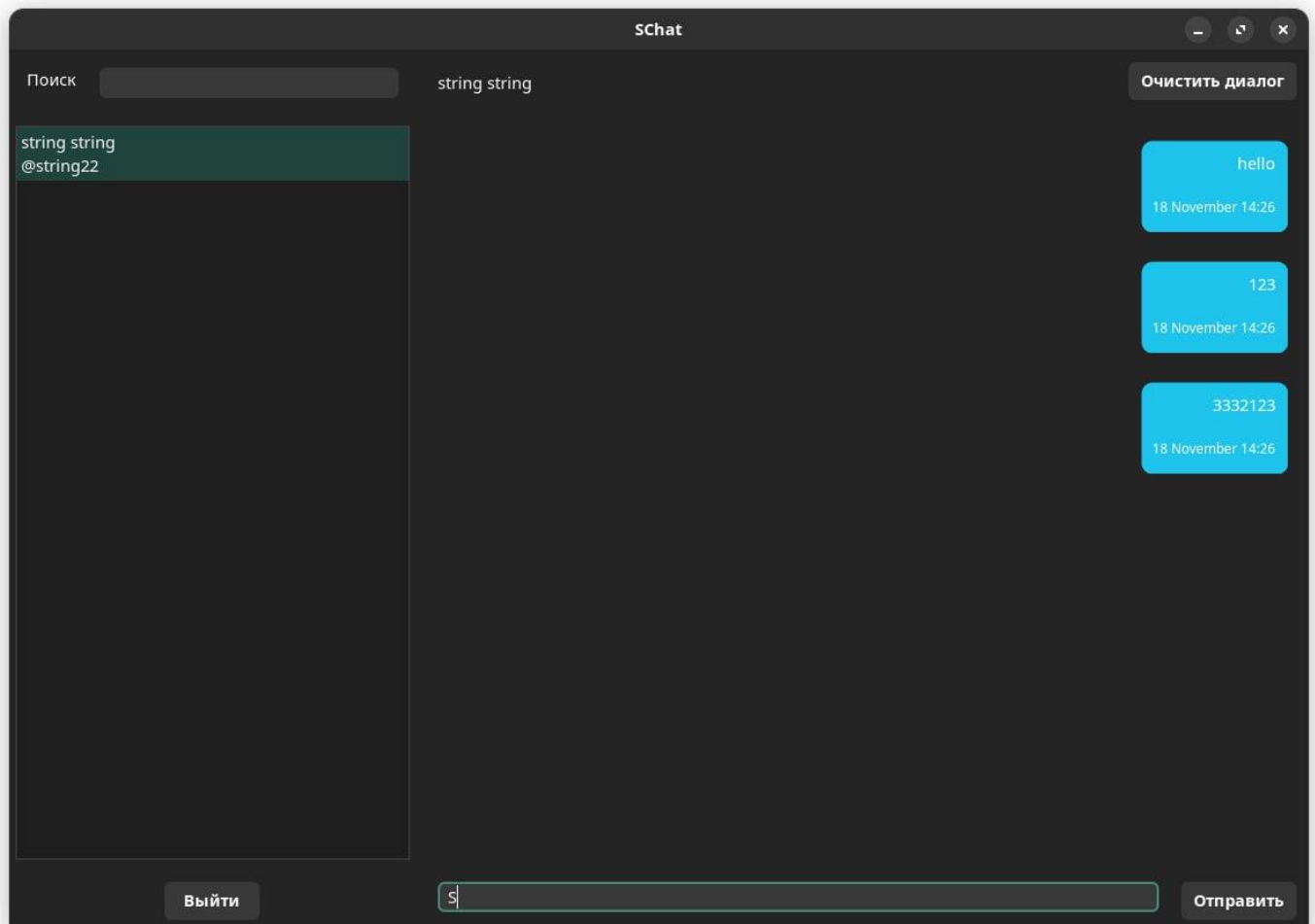
Чтобы реализовать переписку пользователей, вам нужно создать и настроить серверное приложение, которое будет обрабатывать запросы от клиентов, сохранять сообщения в базе данных и обеспечивать их передачу между пользователями.

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:

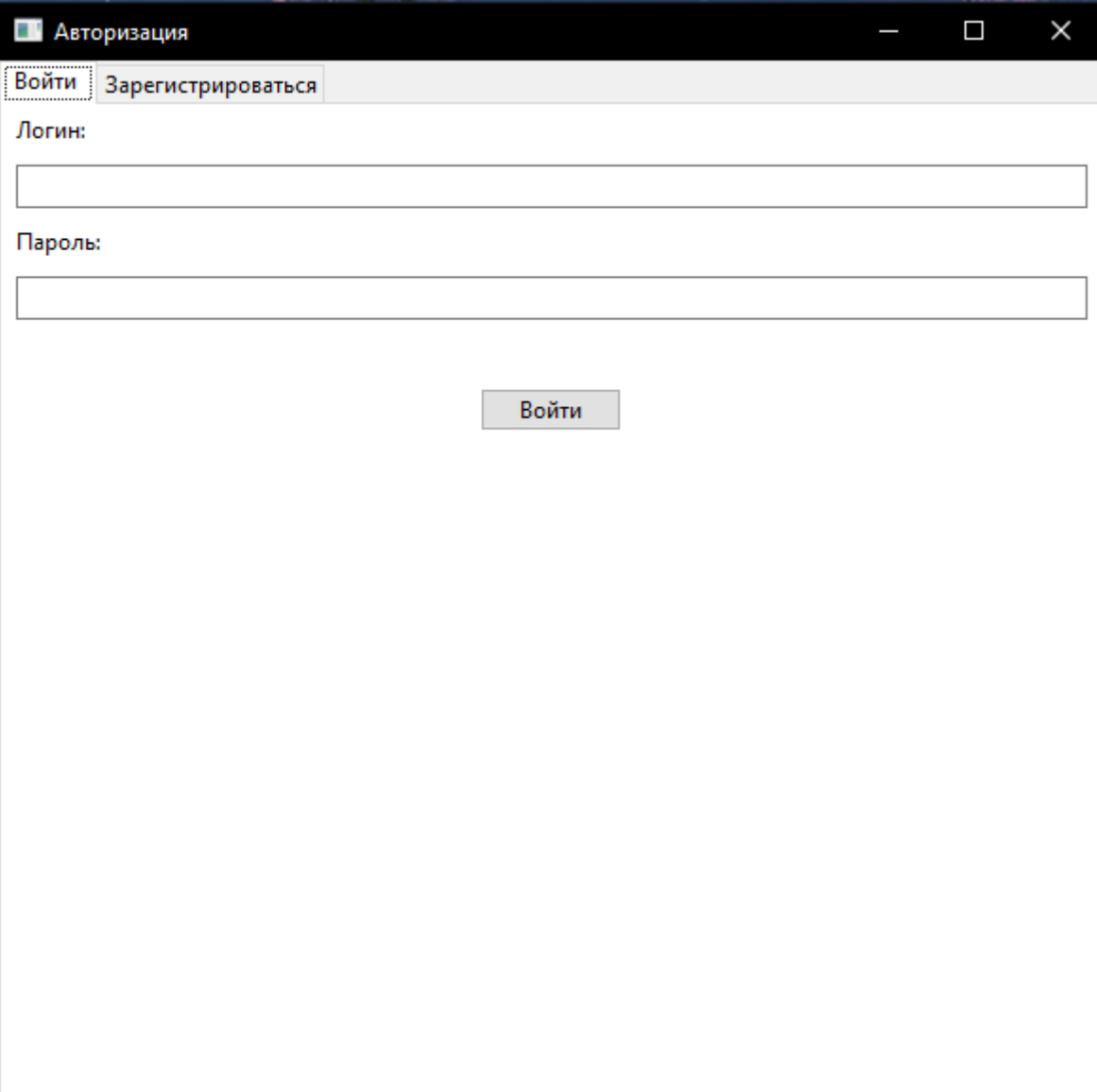


Как мы можем заметить у нас имеется следующий функционал:

- 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:

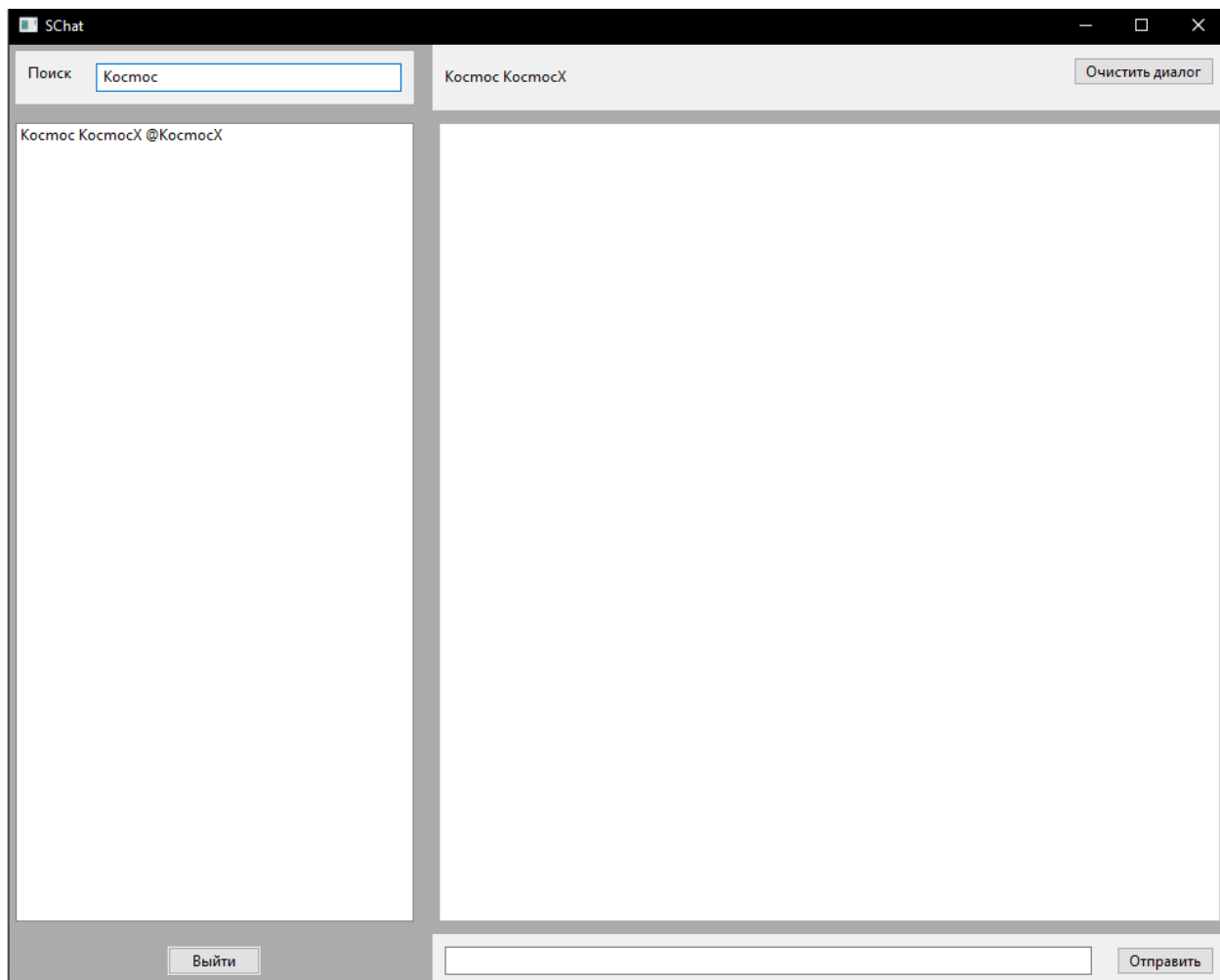
...

Повторите пароль:

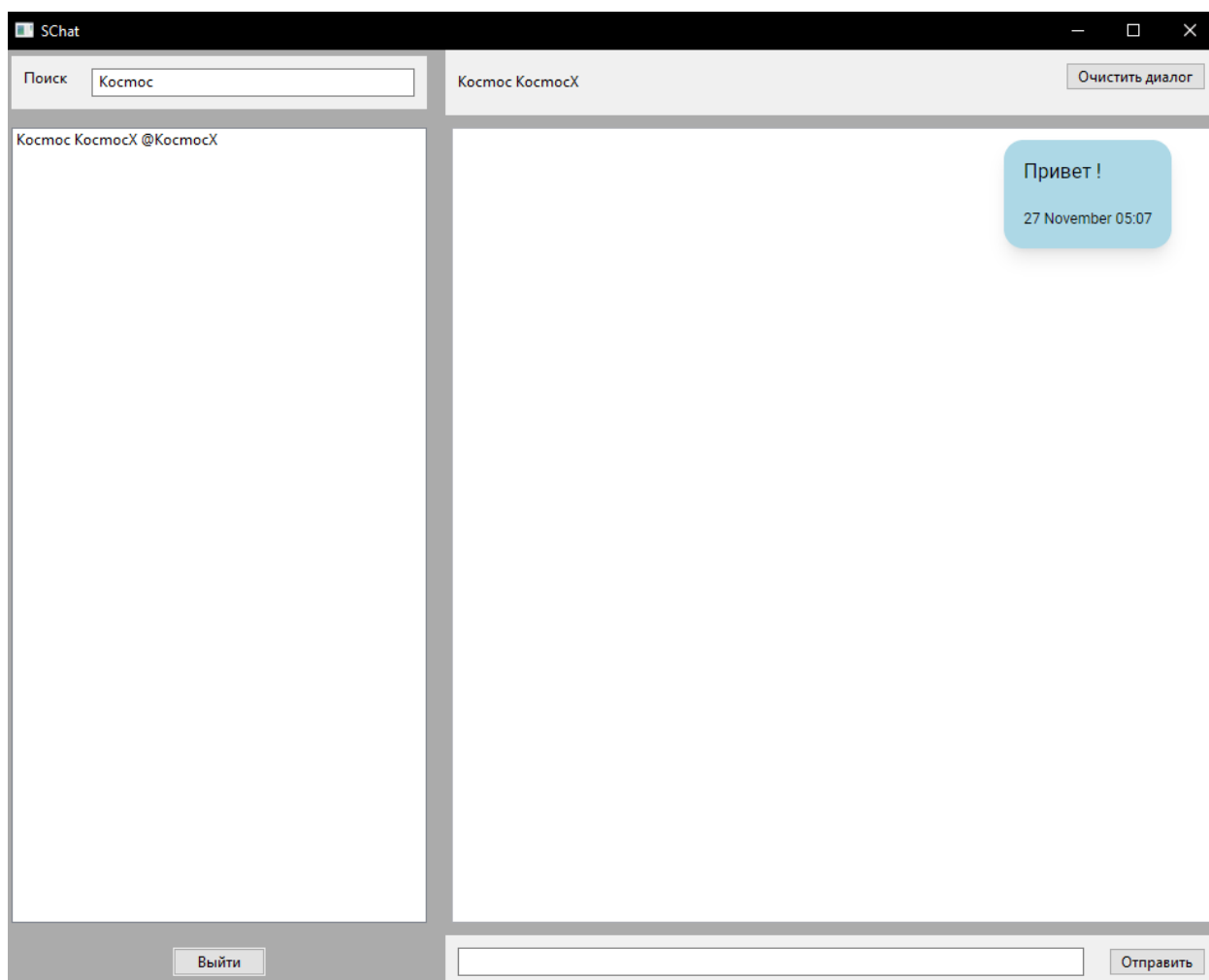
...

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!



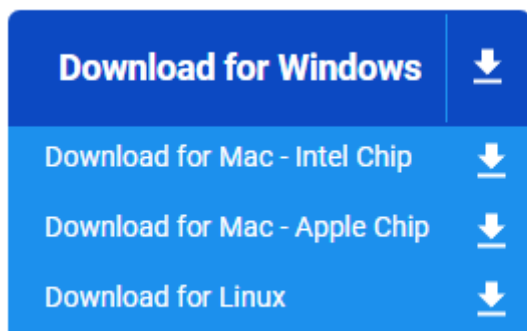
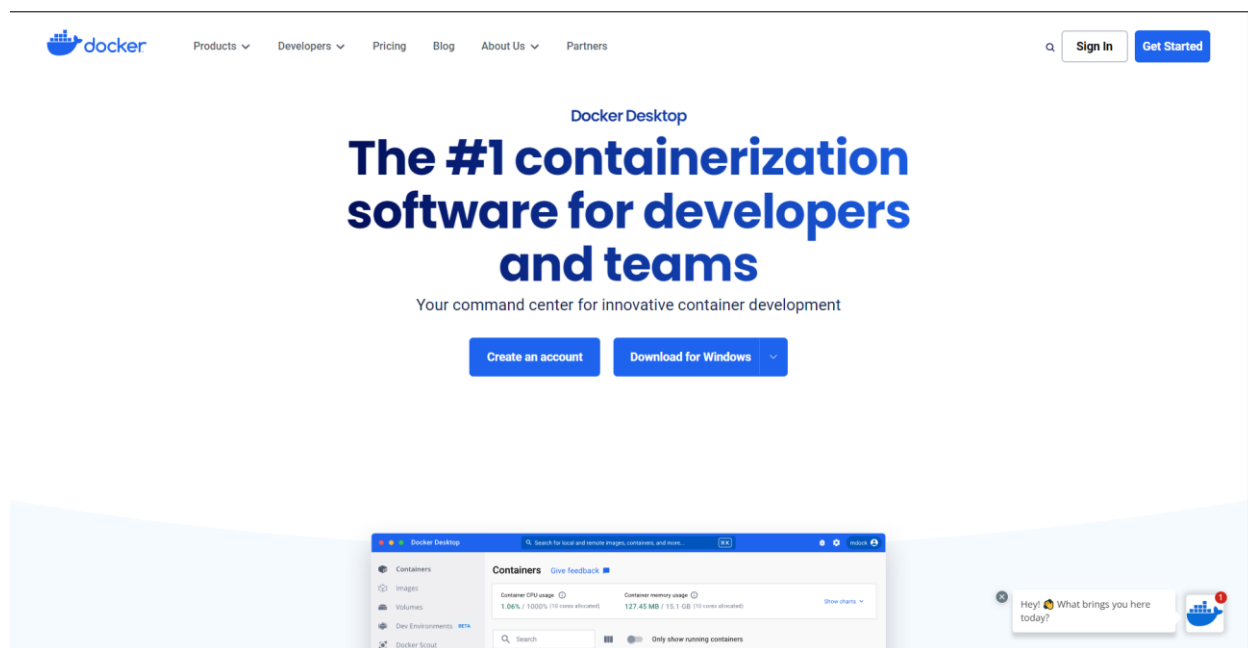
Далее я расскажу о том, как реализовать данный функционал, и что нужно использовать, чтобы всё получилось)

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

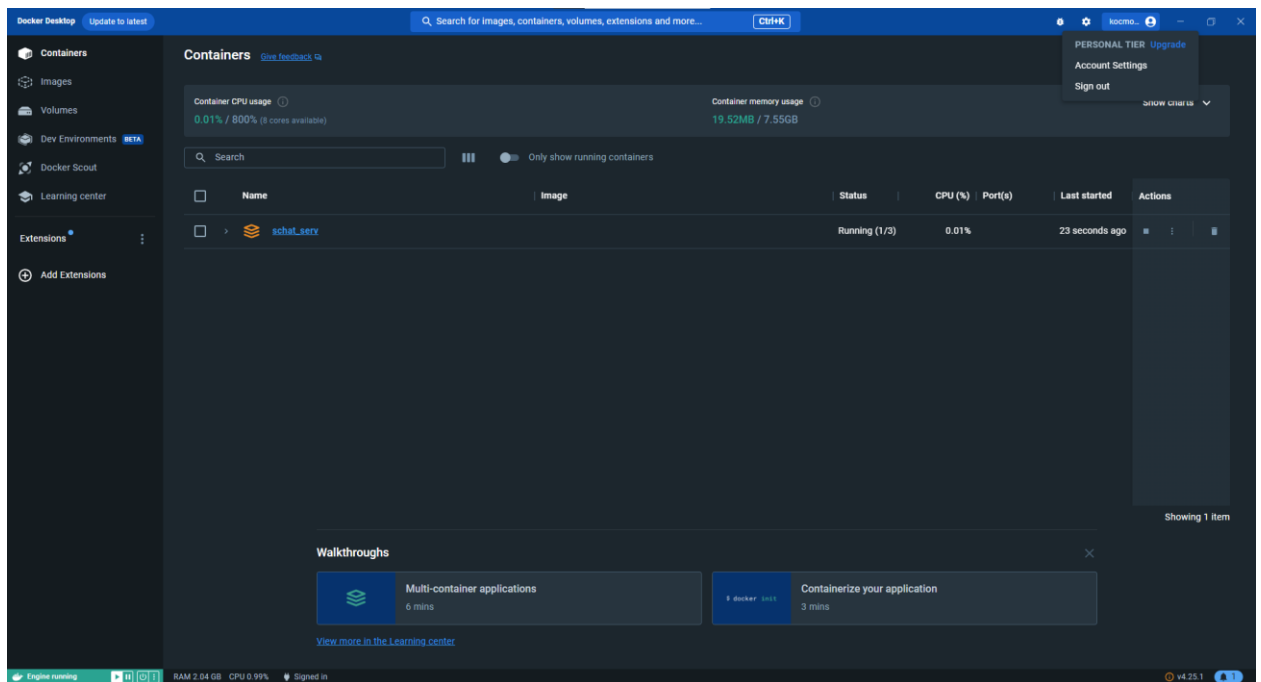
Как скачать Docker:

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 запуска 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 или другой реестр образов.

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 придётся 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. Также 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;
- **Обрабатывает 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

Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
schat_serv		Running (1/3)	0%		9 minutes ago	
schat_pma 32ecdc389b6b	phomyadmin/phomyadmin	Running	0%	8081:80	9 minutes ago	
schat_api 0f96f44f1134	schat_serv-api	Exited	0%	3000:3000	5 days ago	
schat_db b5b26cd7382b	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 PSF Документы PyPI Вакансии Сообщество

[Пожертвовать](#) [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	График выпусков
3.13	предварительный выпуск 2024-10-01 (планируется)	2029-10 PEP 719
3.12	исправлена ошибка 2023-10-02	2028-10, 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

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



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.  schat_cli  schat_serv

Начнём создание сервера.

Внутри папки schat_serv создадим папку schat_serv и в ней будем создавать следующие файлы.

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			

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.
```

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.

```
database.py
1 import os #строка импортирует модуль os, который предоставляет функции для работы с операционной системой.
2 from dotenv import load_dotenv #Здесь мы используем модуль dotenv для загрузки переменных окружения из файла .env в текущую среду.
3 from peewee import Model, MySQLDatabase #MySQLDatabase - это класс из библиотеки Peewee, который используется для подключения к базе данных MySQL.
4
5
6 load_dotenv() #Эта строка вызывает функцию load_dotenv(), которая загружает переменные окружения из файла .env в текущую среду.
7
8 db_name = os.getenv("DB_NAME") #Здесь мы используем модуль os для получения значения переменной окружения DB_NAME, которая должна содержать имя базы данных.
9 db_user = os.getenv("DB_USER") #Аналогично, мы получаем значение переменной окружения DB_USER, которая содержит имя пользователя базы данных.
10 db_password = os.getenv("DB_PASSWORD") #Здесь мы получаем значение переменной окружения DB_PASSWORD, которая содержит пароль для подключения к базе данных.
11 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.
12 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.
13
14 database = MySQLDatabase(db_name, user=db_user, password=db_password, host=db_host, port=db_port) #Мы создаем экземпляр класса MySQLDatabase из модуля peewee и передаем значения переменных окружения, полученные ранее, для настройки подключения к базе данных.
15
16
17
18
19 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.
20     class Meta: #Здесь мы определяем внутренний класс Meta внутри BaseModel. В этом классе можно указать дополнительные настройки модели, такие как указание используемой базы данных.
21         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.
22
```


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.

```
py   
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**