

Государственное образовательное учреждение высшего профессионального образования



**«Московский государственный технический университет  
имени Н.Э. Баумана»  
(МГТУ им. Н.Э. Баумана)»**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Системы обработки информации и управления» (ИУ-5)

## ДОМАШНЕЕ ЗАДАНИЕ

ChatFreely

Группа ИУ5-35Б

Студент Мушкарин 17.12.2024 /Д.Е. Мушкарин/

(Подпись, дата)

(И.О.Фамилия)

Преподаватель \_\_\_\_\_ /Ю. Е. Гапанюк/

(Подпись, дата)

(И.О.Фамилия)

2024

**Задание:**

Разработать программу на языке Rust.

**ChatFreely** - это анонимный чат бот, написанный на python с применением асинхронного подхода

В разработке применялись:

- aiogram 
- aiomysql
- pytest\_asyncio (для тестов)

Необходимый функционал:

- Механизм хранения записей всех пользователей и регулировка доступа к нему
- Соединение между любыми пользователями
- Система рейтинга, очереди соединений, профиль
- Поддержка всех типов вложений, включая стикеры, файлы
- Ответы на сообщения синхронизированы между пользователями

С целью непрерывной и надежной интеграции обновлений был реализован набор тестов для проверки основного функционала работы с базой данных.

**Текст программы:**

**//main.py**

**import os**

**env = os.getenv('ENV', 'default')**

**async def main():**

**await connect()**

**await create\_tables\_if\_not\_exist()**

**await start\_bot()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**if env == 'default':**

**from .database import connect, create\_tables\_if\_not\_exist**

**import asyncio**

**from .bot import start\_bot**

**asyncio.run(main())**

**if env == 'test':**

**import pytest**

**pytest.main()**

**//bot.py**

**from .configure import get\_key**

**from aiogram.utils.keyboard import InlineKeyboardBuilder**

**from aiogram import Router, Bot, Dispatcher, F**

**from aiogram.filters import Command**

**from aiogram.types import Message, CallbackQuery**

**from .keyboards import (**

**inline\_to\_menu\_buttons\_list,**

**inline\_banned\_buttons\_list,**

**inline\_connected\_buttons\_list,**

**inline\_dialogue\_end\_buttons\_list,**

**inline\_rate\_buttons\_list,**

**inline\_regular\_buttons\_list,**

```
    inline_search_buttons_list
)
```

```
from .database import *
```

```
API_TOKEN = get_key()
bot = Bot(token=API_TOKEN)
dp = Dispatcher()
router = Router()
dp.include_router(router)
```

```
async def start_bot():
    await dp.start_polling(bot)
```

```
@router.callback_query(F.data == 'profile')
@router.message(Command("profile"))
async def profile(call):
    if isinstance(call, CallbackQuery):
        await call.answer("", show_alert=False)
        data = await fetch_user(call.from_user.id)
        answer = f"""
```

```
Давай рассмотрим твой профиль:
```

```
Ваш рейтинг: {data.rating}
```

```
Всего диалогов: {data.total_connections}
```

```
Регистрация: {data.registration}
```

```
"""
```

```
    builder = InlineKeyboardBuilder()
    builder.add(*inline_to_menu_buttons_list)
    markup = builder.as_markup()
    await bot.send_message(chat_id = call.from_user.id, text=
answer, reply_markup=markup)
```

```
@router.message(Command("start"))
async def start(message: Message):
```

```
builder = InlineKeyboardBuilder()
builder.add(*inline_to_menu_buttons_list)
markup = builder.as_markup()
await log_user(message.from_user.id)
await message.answer(f"Привет,
{message.from_user.username}! Используй /menu, чтобы
попасть в основное меню.", reply_markup=markup)
```

```
@router.callback_query(F.data == 'stop')
@router.message(Command("stop"))
async def stop_search(call):
    print("Stop called.")
    if isinstance(call, CallbackQuery):
        await call.answer("", show_alert=False)
    is_searching = await is_in_search(call.from_user.id)
    if not is_searching:
        print(is_searching)
        return await menu(call)
```

```
await drop_from_search(call.from_user.id)
await bot.send_message(chat_id = call.from_user.id, text= f"Вы
остановили поиск собеседника!")
await update_status(call.from_user.id, "normal")
return await menu(call)
```

```
@router.callback_query(F.data == 'quit')
@router.message(Command("quit"))
async def quit_dialogue(call):
    if isinstance(call, CallbackQuery):
        await call.answer("", show_alert=False)
    builder = InlineKeyboardBuilder()
    builder.add(*inline_dialogue_end_buttons_list)
    markup = builder.as_markup()
    counterpart = await get_connected_user(call.from_user.id)
    if counterpart is None:
```

```
    return await bot.send_message(chat_id = call.from_user.id,
text= f"Вы не находитесь в диалоге. Используйте /menu,
чтобы попасть в меню, или /search, чтобы найти
собеседника.", reply_markup=markup)
    await bot.send_message(chat_id = call.from_user.id, text= f"Вы
остановили диалог с вашим собеседником. Используйте
/search, чтобы найти нового собеседника, или /menu для
возврата в меню.", reply_markup=markup)
    await drop_from_connections(call.from_user.id)
    await bot.send_message(chat_id = counterpart, text= f"Ваш
собеседник остановил диалог. Используйте /search, чтобы
найти нового собеседника, или /menu для возврата в меню.",
reply_markup=markup)
    await after_dialogue(call.from_user.id,counterpart)
    builder = InlineKeyboardBuilder()
    builder.add(*inline_rate_buttons_list)
    builder.adjust(*[2,1])
    markup = builder.as_markup()
    await bot.send_message(chat_id=counterpart, text="Как вы
оцените вашего последнего
собеседника?",reply_markup=markup)
    await bot.send_message(chat_id=call.from_user.id, text="Как
вы оцените вашего последнего
собеседника?",reply_markup=markup)
```

```
@router.callback_query(F.data == 'decrease_rating')
async def decrease_rating(call: CallbackQuery):
    usr = await fetch_user(call.from_user.id)
    if usr.last_connected is not None:
        await call.message.edit_text(text="Спасибо за отзыв!")
        await sub_rating(usr.last_connected, usr.telegram_uid)
    else:
        await call.message.edit_text(text="Устаревший отзыв.")
```

```
@router.callback_query(F.data == 'increase_rating')
```

```
async def increase_rating(call: CallbackQuery):
    usr = await fetch_user(call.from_user.id)
    if usr.last_connected is not None:
        await call.message.edit_text(text="Спасибо за отзыв!")
        await add_rating(usr.last_connected, usr.telegram_uid)
    else:
        await call.message.edit_text(text="Устаревший отзыв.")
```

```
@router.callback_query(F.data == 'report')
async def report(call: CallbackQuery):
    usr = await fetch_user(call.from_user.id)

    if usr.last_connected is not None:
        await call.message.edit_text(text="Спасибо за отзыв!")
        await add_report(usr.last_connected, usr.telegram_uid)
    else:
        await call.message.edit_text(text="Устаревший отзыв.")
```

```
@router.callback_query(F.data == 'appeal')
async def report(call: CallbackQuery):
    await call.message.edit_text(text="Отправьте заказным
письмом заявку на разблокировку с полной информацией о
себе по данному адресу:")
    await bot.send_location(call.from_user.id, latitude=55.766321,
longitude=37.686584)
```

```
@router.callback_query(F.data == 'about')
async def about(call: CallbackQuery):
    await call.answer(text="", show_alert=False)
    ab = await bot.get_me()
    await bot.send_message(chat_id=call.from_user.id,
text=str(ab)+" located at:")
    await bot.send_location(call.from_user.id, latitude=55.766321,
longitude=37.686584)
```

```

@router.callback_query(F.data == 'search')
@router.message(Command("search"))
async def search(call: CallbackQuery):
    usr = await fetch_user(call.from_user.id)
    if isinstance(call, CallbackQuery):
        await call.answer("", show_alert=False)
    if usr.user_status != 'normal':
        return await menu(call)
    await update_status(call.from_user.id, "search")
    await bot.send_dice(chat_id = call.from_user.id)
    builder = InlineKeyboardBuilder()
    builder.add(*inline_search_buttons_list)
    # builder.adjust([1])
    markup = builder.as_markup()
    await bot.send_message(chat_id = call.from_user.id, text=
f"Ищем для вас подходящего собеседника...",
reply_markup=markup)
    counterpart = await get_counterpart(call.from_user.id)
    if counterpart is None:
        await add_to_search(usr)
    else:
        await drop_from_search(counterpart.telegram_uid)
        await update_status(call.from_user.id, "connected")
        await update_status(counterpart.telegram_uid, "connected")
        print(f"Found match for user
{call.from_user.id}(@{call.from_user.username}),
{counterpart.telegram_uid}")

        builder = InlineKeyboardBuilder()
        builder.add(*inline_connected_buttons_list)
        markup = builder.as_markup()
        counterpart_answer_rating = f"{counterpart.rating} □" if
counterpart.rating >= 0 else f"{counterpart.rating} □"

```



```

    my_answer_rating = f"{usr.rating} □" if usr.rating >= 0 else
f"{usr.rating} □"
    await bot.send_message(chat_id = call.from_user.id, text=
f"Собеседник найден, рейтинг: {counterpart_answer_rating}
\nИспользуйте /quit, чтобы прекратить диалог.",
reply_markup=markup)
    await bot.send_message(chat_id = counterpart.telegram_uid,
text= f"Собеседник найден, рейтинг: {my_answer_rating}
\nИспользуйте /quit, чтобы прекратить диалог.",
reply_markup=markup)
    await
add_to_connections(call.from_user.id,counterpart.telegram_uid)

```

```

@router.callback_query(F.data == 'menu')
@router.message(Command("menu"))
async def menu(call):

```

```

    if isinstance(call, CallbackQuery):
        await call.answer("", show_alert=False)
        await log_user(call.from_user.id)
        user = await fetch_user(call.from_user.id)
        status = user.user_status
        answer = "None"
        if status == 'normal':
            answer = f"Добро пожаловать в главное меню,
{call.from_user.full_name[:50]}!"
            buttons_list = inline_regular_buttons_list
        elif status == 'banned':
            answer = "Вы были заблокированы. Нажмите ниже, если
хотите подать апелляцию."
            buttons_list = inline_banned_buttons_list
        elif status == 'search':
            buttons_list = inline_search_buttons_list

```

```
    answer = f"Вы в процессе поиска собеседника,  
{call.from_user.full_name[:50]}. Используйте /stop, чтобы  
прекратить поиск."
```

```
    elif status == 'connected':  
        buttons_list = inline_connected_buttons_list  
        answer = f"Вы находитесь в диалоге,  
{call.from_user.full_name[:50]}. Используйте /quit, чтобы  
прекратить диалог."
```

```
    keyboard_builder = InlineKeyboardBuilder()  
    keyboard_builder.add(*buttons_list)  
    keyboard_builder.adjust(*[1,2])  
    menu_markup = keyboard_builder.as_markup()  
    await bot.send_message(chat_id = call.from_user.id, text=  
answer, reply_markup=menu_markup)
```

```
@router.message()  
async def send_message(message : Message):  
    did_reply = False if message.reply_to_message is None else  
True  
    connected = await get_connected_user(message.from_user.id)  
    if connected is not None:  
        if not did_reply:  
            reply = await message.copy_to(chat_id=connected)  
        else:  
            new_reply_id = await  
get_reply_id(message.reply_to_message.message_id,  
message.from_user.id)  
            # print(new_reply_id, "\n\n\n\n")  
            reply = await  
message.copy_to(chat_id=connected,reply_to_message_id=new_  
reply_id)  
  
    pair = [message.message_id, reply.message_id]  
    await log_message(message.from_user.id, pair[0], pair[1])
```

```
    await log_message(message.from_user.id, pair[1], pair[0])
else:
    await basic(message)
```

```
async def basic(message : Message):
    builder = InlineKeyboardBuilder()
    builder.add(*inline_to_menu_buttons_list)
    markup = builder.as_markup()
    await message.answer(text="Неизвестная команда.
Используйте /menu, чтобы попасть в основное меню",
reply_markup=markup)
```

```
//__init__.py
version = "1.0.0"
```

```
//configure.py
import json
```

```
def read_json_contents(filename : str):
    try:
        with open(filename, "r") as file:
            try:
                contents = json.load(file)
            except json.JSONDecodeError as err:
                print(f"config.json format error in {err.doc}")
                print(f"Error: {err.msg}")
                print(f"At line: {err.lineno}, coloumn: {err.colno}")
                print(f"Pos: {err.pos}")
                file.seek(0)
                old_contents = file.readlines()
                with open(f"{filename}.old", "w") as old_file:
                    old_file.writelines(old_contents)
                contents = { "Users" : []}
```

```

        with open("config.json", "w") as file:
            json.dump(contents, file, indent=4)
except FileNotFoundError:
    print("Creating config.json...")
    contents = { "Users" : []}
    with open("config.json", "w") as file:
        json.dump(contents, file, indent=4)
except Exception:
    print("Unknown exception.")
    raise Exception("The end.")
return contents

```

```

def add_user(User):
    contents = read_json_contents("config.json")
    with open("config.json", "w") as file:
        contents["Users"].append(User)
        json.dump(contents, file, indent=4)

```

```

def list_users():
    contents = read_json_contents("config.json")
    if contents["Users"]:
        print("Here is the list of all the users (username@host):")

```

```

print("_____ \n"
)

```

```

    [print(i) for i in [_["username"]+"@"+_["host"] for _ in
contents["Users"]]]

```

```

print("_____ \n"
)

```

```

    else:
        print("Database is empty\n")

```

```

def is_user(username : str):
    contents = read_json_contents("config.json")

```

```
for i in contents["Users"]:
    if i["username"] == username:
        return True
return False
```

```
def remove_user(username):
    contents = read_json_contents("config.json")
    with open("config.json", "w") as file:
        index = 0
        for i in contents["Users"]:
            if i["username"] == username:
                break
            index+=1
        else:
            print(f"User '{username}' not found")
            json.dump(contents, file, indent=4)
            return
        del contents["Users"][index]
        json.dump(contents, file, indent=4)
```

```
def update_user(username):
    contents = read_json_contents("config.json")
    with open("config.json", "r") as file:
        index = 0
        for i in contents["Users"]:
            if i["username"] == username:
                break
            index +=1
        else:
            print(f"User '{username}' not found. ")
            return
    while True:
        print("What you would like to change?")
        print("1.username")
        print("2.host")
```

```
print("3.password")
print("4.database")
print("5.Save and exit edit menu")
```

```
a =input()
if not a.isdecimal():
    print("Please, enter number from 1 to 5. Try again")
    continue
a = int(a)
if (a < 1) or (a > 5):
    print("Incorrect choice. Try again")
    continue
```

```
if a == 1:
    new_username = str(input(f"Enter new username (Non-
empty) for {username}: "))
    if not new_username:
        print("Username can't be empty. Try again")
        continue
    contents["Users"][index]["username"] = new_username
if a == 2:
    new_host = str(input(f"Enter new host (default = localhost)
for {username}: "))
    if not new_host:
        new_host = "localhost"
    contents["Users"][index]["host"] = new_host
if a == 3:
    new_password = str(input(f"Enter new password (default =
None) for {username}: "))
    contents["Users"][index]["password"] = new_password
if a == 4:
    new_database = str(input(f"Enter new database (default =
None) for {username}: "))
    contents["Users"][index]["database"] = new_database
if a == 5:
```

```
        break
    with open("config.json", "w") as file:
        json.dump(contents, file, indent=4)
```

```
def get_credentials(username):
    contents = read_json_contents("config.json")
    index = 0
    for i in contents["Users"]:
        if i["username"] == username:
            break
        index +=1
    else:
        print(f"User {username} not found")
        return None
    return contents["Users"][index]
```

```
def get_key():
    contents = read_json_contents("config.json")
    return contents["API"]
```

```
def add_key(key : str):
    contents = read_json_contents("config.json")
    with open("config.json", "w") as file:
        contents["API"] = key
        json.dump(contents, file, indent=4)
```

```
def main():
    while (True):
        print(f"Welcome to the Manager control panel")
        print("-----")
        print("1.Add new user credentials")
        print("2.Remove user credentials")
        print("3.List users")
        print("4.Update user")
        print("5.Add API key")
```

```
print("6.Exit")

a =input()
if not a.isdecimal():
    print("Please, enter number from 1 to 5. Try again")
    continue
a = int(a)
if (a < 1) or (a > 5):
    print("Incorrect choice. Try again")
    continue
if a == 1:
    print("Please, enter your:")
    User = {
        "username" : str(input("username (Non empty): ")),
        "host"      : str(input("host    (default = localhost): ")),
        "password" : str(input("password (default = None): ")),
        "database" : str(input("database (default = None): "))
    }
    if not User["username"]:
        print("Incorrect username. Try again\n")
        continue
    if not User["host"]:
        User["host"] = "localhost"
    add_user(User)
if a == 2:
    remove_user(str(input("username(Non empty): ")))
if a == 3:
    list_users()
if a == 4:
    update_user(str(input("username (Non empty): ")))
if a == 5:
    add_key(str(input("API key: ")))
if a == 6:
    break
```



```
if __name__ == "__main__":  
    main()
```

```
    //database.py  
import aiomysql  
from .configure import get_credentials, add_user, is_user  
from .user import SearchUser, User  
import warnings  
import random  
import json
```

```
pool = None  
async def create_database_async_pool(user =  
"ChatFreelyAdmin"):  
    credentials = get_credentials(user)  
    if not credentials:  
        warnings.warn("Incorrect credentials")  
        return None  
    try:  
        global pool  
        pool = await aiomysql.create_pool(  
            user=credentials["username"],  
            db=credentials["database"],  
            password=credentials["password"],  
            host=credentials["host"],  
            minsize=1,  
            maxsize=10,  
            autocommit = True,  
            port=3306,  
        )  
    if pool is None:  
        raise Exception("Connection failed unexpectedly")  
    else:  
        print("Correct connetion acquired!")
```

```
except aiomysql.Error as err:  
    print("Error:", err.args)  
    return None
```

```
async def grace_close():  
    if pool:  
        pool.close() # Close the pool to avoid lingering connections  
        await pool.wait_closed() # Wait for all connections to close
```

```
async def connect(user : str = None):  
    if pool is not None:  
        pass  
    if user is None:  
        await create_database_async_pool()  
    else:  
        print("Logging in with custom user!\n")  
        await create_database_async_pool(user)
```

```
    if pool is None:  
        raise BaseException("Could not resolve mysql database  
connection. Maybe check credentials/start db.")
```

```
async def create_tables_if_not_exist():  
    warnings.filterwarnings("ignore", message="Table '.*' already  
exists")  
    async with pool.acquire() as conn:  
        async with conn.cursor() as cursor:  
            cursor: aiomysql.Cursor  
            await cursor.execute(  
                """  
                CREATE TABLE IF NOT EXISTS users (  
                    telegram_uid BIGINT PRIMARY KEY NOT NULL,  
                    user_status ENUM('normal', 'connected', 'search',  
'banned') DEFAULT 'normal',  
                    INDEX (user_status),
```

```
        rating INT DEFAULT 0,  
        registration TIMESTAMP DEFAULT  
CURRENT_TIMESTAMP,  
        total_connections INT DEFAULT 0,  
        last_update TIMESTAMP DEFAULT  
CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  
        last_connected BIGINT NULL DEFAULT NULL,  
        reports INT DEFAULT 0  
    );  
    """)
```

```
await cursor.execute(  
    """
```

```
    CREATE TABLE IF NOT EXISTS search (  
        telegram_uid BIGINT PRIMARY KEY NOT NULL,  
        rating INT DEFAULT 0,  
        FOREIGN KEY (telegram_uid) REFERENCES  
users(telegram_uid) ON DELETE CASCADE  
    );  
    """)
```

```
await cursor.execute(  
    """
```

```
    CREATE TABLE IF NOT EXISTS connections (  
        telegram_uid_1 BIGINT PRIMARY KEY NOT NULL,  
        FOREIGN KEY (telegram_uid_1) REFERENCES  
users(telegram_uid) ON DELETE CASCADE,  
        telegram_uid_2 BIGINT NOT NULL,  
        FOREIGN KEY (telegram_uid_2) REFERENCES  
users(telegram_uid) ON DELETE CASCADE,  
        UNIQUE INDEX T_UID_2(telegram_uid_2),  
        messages_table JSON DEFAULT '{}'  
    );  
    """)
```

```
async def drop_tables(): #ОCTOPOЖHO!!
```

```

async with pool.acquire() as conn:
    conn : aiomysql.Connection
    async with conn.cursor() as cursor:
        cursor: aiomysql.Cursor
        try:
            warnings.filterwarnings("ignore", message="Unknown
table '.*'")
            await cursor.execute("DROP TABLE IF EXISTS
connections;")
            await cursor.execute("DROP TABLE IF EXISTS search;")
            await cursor.execute("DROP TABLE IF EXISTS users;")
        except aiomysql.OperationalError as err:
            warnings.warn(f"Err: {err.args}")
            await conn.rollback()
            return False

        await conn.commit()
    return True

```

```

async def log_user(telegram_uid):
    # print(f"Message {context} logged.")
    async with pool.acquire() as conn:
        conn : aiomysql.Connection
        async with conn.cursor() as cursor:
            cursor: aiomysql.Cursor
            await cursor.execute(
                """
                INSERT INTO users
                (
                    telegram_uid
                )
                VALUES (%s)
                ON DUPLICATE KEY UPDATE
                last_update = CURRENT_TIMESTAMP
                """, (telegram_uid,))

```

```
await conn.commit()
```

```
async def drop_user(telegram_uid):  
    async with pool.acquire() as conn:  
        conn : aiomysql.Connection  
        async with conn.cursor() as cursor:  
            cursor: aiomysql.Cursor  
            await cursor.execute(  
                """  
  
                DELETE FROM users WHERE telegram_uid = %s  
                """, (telegram_uid,))
```

```
async def has_data():  
    res = {"users" : 0, "connections" : 0, "search" : 0}  
    async with pool.acquire() as conn:  
        conn : aiomysql.Connection  
        async with conn.cursor() as cursor:  
            cursor: aiomysql.Cursor  
            await cursor.execute(  
                """  
  
                SELECT * FROM users;  
                """)  
            if cursor.rowcount > 0:  
                res["users"] = 1  
  
            await cursor.execute(  
                """  
  
                SELECT * FROM connections;  
                """)  
            if cursor.rowcount > 0:  
                res["connections"] = 1  
  
            await cursor.execute(  
                """  
  
                SELECT * FROM search;
```

```
        """)
    if cursor.rowcount > 0:
        res["search"] = 1

    await conn.commit()
    return res
```

```
async def add_to_search(user : User):
    async with pool.acquire() as conn:
        conn : aiomysql.Connection
        async with conn.cursor() as cursor:
            cursor: aiomysql.Cursor
            await cursor.execute(
                """
                REPLACE INTO search
                (
                    telegram_uid,
                    rating
                )
                VALUES (%s,%s);
                """, (user.telegram_uid, user.rating))
            await conn.commit()
            print("User added!")
```

```
async def drop_from_search(telegram_uid):
    async with pool.acquire() as conn:
        conn : aiomysql.Connection
        async with conn.cursor() as cursor:
            cursor: aiomysql.Cursor

            await cursor.execute(
                """
                DELETE FROM search
                WHERE telegram_uid = %s
                """, (telegram_uid, ))
```

```
await conn.commit()
```

```
async def add_to_connections(recipient, counterpart):
```

```
    async with pool.acquire() as conn:
```

```
        conn : aiomysql.Connection
```

```
        async with conn.cursor() as cursor:
```

```
            cursor: aiomysql.Cursor
```

```
            await cursor.execute(  
                """
```

```
                
```

```
                REPLACE INTO connections
```

```
                (  
                    telegram_uid_1,
```

```
                    telegram_uid_2
```

```
                )
```

```
                VALUES (%s,%s)
```

```
                ;
```

```
                """, (recipient, counterpart))
```

```
            await conn.commit()
```

```
async def drop_from_connections(telegram_uid):
```

```
    async with pool.acquire() as conn:
```

```
        conn : aiomysql.Connection
```

```
        async with conn.cursor() as cursor:
```

```
            cursor: aiomysql.Cursor
```

```
            await cursor.execute(  
                """
```

```
                
```

```
                DELETE FROM connections
```

```
                WHERE telegram_uid_1 = %s
```

```
                OR telegram_uid_2 = %s
```

```
                """, (telegram_uid, telegram_uid))
```

```
            await conn.commit()
```

```
async def update_status(uid : str, status : str):
```

```
    async with pool.acquire() as conn:
```

```
        conn : aiomysql.Connection
```

```

async with conn.cursor() as cursor:
    cursor: aiomysql.Cursor
    await cursor.execute(
        """
        UPDATE users
        SET user_status = %s
        WHERE telegram_uid = %s
        """, (status, uid))
    await conn.commit()

```

```

async def after_dialogue(uid : str, uid2 : str):
    async with pool.acquire() as conn:
        conn : aiomysql.Connection
        async with conn.cursor() as cursor:
            cursor: aiomysql.Cursor
            await cursor.execute(
                """
                UPDATE users
                SET user_status = 'normal', total_connections =
total_connections + 1, last_connected = %s
                WHERE telegram_uid = %s;
                """, (uid, uid2))
            await cursor.execute(
                """
                UPDATE users
                SET user_status = 'normal', total_connections =
total_connections + 1, last_connected = %s
                WHERE telegram_uid = %s;
                """, (uid2, uid))
            await conn.commit()

```

```

async def fetch_user(user_id):
    async with pool.acquire() as conn:
        async with conn.cursor() as cursor:
            cursor: aiomysql.Cursor

```



```

await cursor.execute(
    """
    SELECT * FROM users WHERE telegram_uid = %s
    """, (user_id,))
if cursor.rowcount==0:

    print(f"An attempt to fetch unknown user: {user_id}")
    return None
data = await cursor.fetchone()
return User(data)

```

```

async def get_connected_user(telegram_uid : str):
    async with pool.acquire() as conn:
        async with conn.cursor() as cursor:
            await cursor.execute(
                """
                SELECT * FROM connections WHERE telegram_uid_1 =
                %s OR telegram_uid_2 = %s;
                """, (telegram_uid, telegram_uid))
            if cursor.rowcount > 0:
                data = await cursor.fetchone()
                if data[0]==telegram_uid:
                    return data[1]
                return data[0]
            else:
                print(f"No user {telegram_uid} in connections.")
                return None

```

```

async def is_in_search(telegram_uid):
    async with pool.acquire() as conn:
        async with conn.cursor() as cursor:

```

```

        cursor: aiomysql.Cursor

```

```

        await cursor.execute(

```

```

        """
        SELECT * FROM search
        WHERE telegram_uid = %s
        """, (telegram_uid,))

    # await conn.commit()
    if cursor.rowcount > 0:
        res = True
    else:
        res = False
    # await conn.commit()

    return res

async def get_counterpart(telegram_uid : str):
    async with pool.acquire() as conn:
        async with conn.cursor() as cursor:

            cursor: aiomysql.Cursor
            user = await fetch_user(telegram_uid)
            user : User
            await cursor.execute(
                """
                SELECT * FROM search
                WHERE telegram_uid != %s
                ORDER BY ABS(rating-%s);
                """, (user.telegram_uid, user.rating ))
            if cursor.rowcount > 0:
                data = await cursor.fetchone()
                return SearchUser(data)
            else:
                print(f"Match for {user.telegram_uid, user.rating } Not
Found.")
                return None

```

```
async def add_rating(telegram_uid, voter_uid):
    async with pool.acquire() as conn:
        async with conn.cursor() as cursor:
            cursor : aiomysql.Cursor
            await cursor.execute("""
                UPDATE users
                SET rating = rating + 1
                WHERE telegram_uid = %s;
            """, (telegram_uid,))
            await cursor.execute("""
                UPDATE users
                SET last_connected = NULL
                WHERE telegram_uid = %s;
            """, (voter_uid,))
            await conn.commit()
```

```
async def sub_rating(telegram_uid, voter_uid):
    async with pool.acquire() as conn:
        async with conn.cursor() as cursor:
            cursor : aiomysql.Cursor
            await cursor.execute("""
                UPDATE users
                SET rating = rating - 1
                WHERE telegram_uid = %s;
            """, (telegram_uid,))
            await cursor.execute("""
                UPDATE users
                SET last_connected = NULL
                WHERE telegram_uid = %s;
            """, (voter_uid,))
            await conn.commit()
```

```
async def add_report(telegram_uid, voter_uid):
    async with pool.acquire() as conn:
        async with conn.cursor() as cursor:
```

```

cursor : aiomysql.Cursor
await cursor.execute("""
    UPDATE users
    SET reports = reports + 1
    WHERE telegram_uid = %s;
    """, (telegram_uid,))
await cursor.execute("""
    UPDATE users
    SET last_connected = NULL
    WHERE telegram_uid = %s;
    """, (voter_uid,))
await conn.commit()

```

```

async def log_message(sender_uid, from_message_id,
bot_message_id):

```

```

    async with pool.acquire() as conn:

```

```

        conn : aiomysql.Connection

```

```

        async with conn.cursor() as cursor:

```

```

            cursor: aiomysql.Cursor

```

```

            await cursor.execute(

```

```

                """

```

```

                SELECT messages_table FROM connections WHERE
telegram_uid_1 = %s OR telegram_uid_2 = %s;

```

```

                """,(sender_uid, sender_uid))

```

```

            fetch = await cursor.fetchone()

```

```

            obj = json.loads(fetch[0])

```

```

            obj[from_message_id] = bot_message_id

```

```

            if len(obj) > 20:

```

```

                res = {}

```

```

                counter = 0

```

```

                for key, value in obj.items():

```

```

                    counter +=1

```

```

                    if counter > 10:

```

```

                        res[key] = value

```

```
obj = res
```

```
load = json.dumps(obj)
```

```
await cursor.execute(
```

```
"""
```

```
UPDATE connections SET messages_table = %s WHERE  
telegram_uid_1 = %s OR telegram_uid_2 = %s;
```

```
""",(load, sender_uid, sender_uid))
```

```
await conn.commit()
```

```
async def get_reply_id(message_id,from_message_id):
```

```
    async with pool.acquire() as conn:
```

```
        conn : aiomysql.Connection
```

```
        async with conn.cursor() as cursor:
```

```
            cursor: aiomysql.Cursor
```

```
            await cursor.execute(
```

```
                """
```

```
                SELECT messages_table FROM connections WHERE  
telegram_uid_1 = %s OR telegram_uid_2 = %s;
```

```
                """,(from_message_id, from_message_id))
```

```
            fetch = await cursor.fetchone()
```

```
            # if fetch is None:
```

```
                # print(f"Message reply not found for users  
{from_message_id}, {bot_message_id}")
```

```
            res = json.loads(fetch[0])
```

```
        res : dict
```

```
        reply_id = res.get(str(message_id))
```

```
        return reply_id
```

```
async def get_two_unique():
```

```
    unique = []
```

```
    while True:
```

```
        again = False
```

```
        unique = [random.randint(0, 1000000) for i in range(2)]
```

```
for item in unique:
    if unique.count(item) > 1:
        again = True
        break
if not again:
    break
return unique
```

```
async def prepare_test_env():
    if not is_user("test_user"):
        add_user({"username" : "test_user", "password" :
"test_password", "host" : "localhost", "database" : "test_db"})
    await connect("test_user")
```

//keyboards.py

```
from aiogram.types import InlineKeyboardButton
```

```
inline_to_menu_buttons_list = [
    InlineKeyboardButton(text="Перейти в меню",
callback_data="menu")
]
inline_dialogue_end_buttons_list = [
    InlineKeyboardButton(text="Перейти в меню",
callback_data="menu"),
    InlineKeyboardButton(text="Найти собеседника",
callback_data="search")
]
inline_search_buttons_list = [
    InlineKeyboardButton(text="Прекратить поиск",
callback_data="stop")
]
inline_connected_buttons_list = [
    InlineKeyboardButton(text="Прекратить диалог",
callback_data="quit")
]
```

```
inline_regular_buttons_list = [  
  
    InlineKeyboardButton(text="Найти собеседника",  
callback_data="search"),  
    InlineKeyboardButton(text="Мой профиль",  
callback_data="profile"),  
    InlineKeyboardButton(text="О боте", callback_data="about")  
]  
inline_banned_buttons_list = [  
    InlineKeyboardButton(text="Подать апелляцию",  
callback_data="appeal")  
]  
inline_rate_buttons_list = [  
    InlineKeyboardButton(text="□",  
callback_data="increase_rating"),  
    InlineKeyboardButton(text="□",  
callback_data="decrease_rating"),  
    InlineKeyboardButton(text="Пожаловаться",  
callback_data="report")  
]
```

//user.py

```
class BaseUser():  
    def __init__(self, data):  
        self._telegram_uid = data[0]  
  
    @property  
    def telegram_uid(self):  
        return self._telegram_uid  
  
    @telegram_uid.setter
```

```
def telegram_uid(self, uid):  
    self._telegram_uid = uid
```

```
class SearchUser(BaseUser):  
    def __init__(self, data):  
        self._telegram_uid = data[0]  
        self._rating = data[1]
```

```
@property  
def rating(self):  
    return self._rating
```

```
@rating.setter  
def rating(self, rating):  
    self._rating = rating
```

```
class User(SearchUser):  
    def __init__(self, data):  
        self._telegram_uid = data[0]  
        self._user_status = data[1]  
        self._rating = data[2]  
        self._registration = data[3]  
        self._total_connections = data[4]  
        self._last_update = data[5]  
        self._last_connected = data[6]  
        self._reports = data[7]
```

```
@property  
def connected_uid(self):  
    return self._connected_uid
```

```
@connected_uid.setter  
def connected_uid(self, uid):  
    self._connected_uid = uid
```



```
@property  
def registration(self):  
    return self._registration
```

```
@registration.setter  
def registration(self, registration):  
    self._registration = registration
```

```
@property  
def user_status(self):  
    return self._user_status
```

```
@user_status.setter  
def user_status(self, status):  
    self._user_status = status
```

```
@property  
def last_update(self):  
    return self._last_update
```

```
@last_update.setter  
def last_update(self, update):  
    self._last_update = update
```

```
@property  
def total_connections(self):  
    return self._total_connections
```

```
@total_connections.setter  
def total_connections(self, total_connections):  
    self._total_connections = total_connections
```

```
@property
```

```
def last_connected(self):  
    return self._last_connected
```

```
@last_connected.setter  
def last_connected(self, last_connected):  
    self._last_connected = last_connected
```

```
@property  
def reports(self):  
    return self._reports
```

```
@reports.setter  
def reports(self, reports):  
    self._reports = reports
```

```
class ConnectedUser(BaseUser):  
    def __init__(self, data):  
        super().__init__(data[0])  
        self._telegram_uid = data[1]
```

```
@property  
def telegram_uid_2(self):  
    return self._telegram_uid
```

```
@telegram_uid_2.setter  
def telegram_uid_2(self, uid):  
    self._telegram_uid = uid
```

```
//conftest.py  
import pytest_asyncio  
from ChatFreelyBot.database import grace_close, drop_tables,  
create_tables_if_not_exist, connect, prepare_test_env
```

```
@pytest_asyncio.fixture(loop_scope="function")  
async def module_setup_teardown():  
    await connect("test_user")  
    await drop_tables()  
    await create_tables_if_not_exist()  
    await prepare_test_env()  
    yield True  
    await drop_tables()  
    await grace_close()
```

```
//test_module_1.py  
# test_module_1.py  
import pytest  
import warnings  
from ChatFreelyBot.database import (  
    log_user, fetch_user,  
    drop_user, get_two_unique,  
    has_data, add_to_search,  
    add_to_connections, drop_tables,  
    drop_from_search, get_counterpart)
```

```
@pytest.mark.usefixtures("module_setup_teardown")  
class TestClass:  
    @pytest.mark.asyncio  
    async def test_add_drop_user(module_setup_teardown): #  
тест добавления и удаления пользователя  
        test_uids = await get_two_unique()  
        test_uid = test_uids[0]  
        await log_user(test_uid)  
        usr = await fetch_user(test_uid)  
        assert usr.telegram_uid == test_uid \  
            and usr.reports == 0 \  
            and usr.total_connections == 0 \  
            and usr.rating == 0 \
```

```
        and usr.user_status == 'normal'    # пользователь
создается нормальным
        await drop_user(test_uid)
        usr = await fetch_user(test_uid)
        assert usr is None    # пользователь действительно
удаляется
```

```
@pytest.mark.asyncio    # продвинутый тест
множественного добавления и удаления
async def test_add_drop_user_two(module_setup_teardown):
    test_uids = await get_two_unique()
    test_uid = test_uids[0]
    await log_user(test_uid)
    await log_user(test_uid)
    usr = await fetch_user(test_uid)
    assert usr is not None    # идентичное
создание
    usr1 = await fetch_user(test_uid)
    usr2 = await fetch_user(test_uid)
    assert (usr1.registration == usr2.registration)    # идентичное
получение
    await drop_user(test_uid)
    usr = await fetch_user(test_uid)
    assert usr is None    # нет двух записей
```

```
@pytest.mark.asyncio
async def test_is_empty(module_setup_teardown):
    res = await has_data()
    for value in res.values():
        assert not value
    test_uids = await get_two_unique()
    for uid in test_uids:
        await log_user(uid)
        await add_to_search(await fetch_user(uid))
    await add_to_connections(test_uids[0], test_uids[1])
```

```
res = await has_data()
for value in res.values():
    assert value
```

```
@pytest.mark.asyncio
async def test_drop_tables(module_setup_teardown):      #
тест очистки таблиц
    warnings.filterwarnings(message="Dropping tables from the
empty database", action='ignore')
    code = await drop_tables()
    assert code
```

```
@pytest.mark.asyncio
async def
test_add_drop_user_multiple(module_setup_teardown):    #
нагрузочный тест
    for entry in range(125):
        await log_user(entry)
        await drop_user(entry)
    res = await has_data()
    for value in res.values():
        assert not value
```

```
@pytest.mark.asyncio
async def test_add_drop_search(module_setup_teardown):
# тест корректного соединения
    test_uids = await get_two_unique()
    for uid in test_uids:
        await log_user(uid)
        usr = await fetch_user(uid)
        await add_to_search(usr)

uid1_counterpart = await get_counterpart(test_uids[0])
assert (uid1_counterpart.telegram_uid == test_uids[1])
```

```
uid2_counterpart = await get_counterpart(test_uids[1])
assert (uid2_counterpart.telegram_uid == test_uids[0]) #
верное соединение
for uid in test_uids:
    await drop_from_search(uid)
    await drop_user(uid)    # убираем из бд
res = await has_data()
for value in res.values():
    assert not value        # очищение работает верно
```

Пример работы программы:

