Урок Чат-боты 2 Tlg

Чат-боты 2. Telegram

Аннотация

В уроке рассказывается про реализацию telegram API в виде библиотеки на примере telegram-bot-API. Мы будем учиться на конкретных задачах: разберем устройство telegram-бота и узнаем об основных возможностях API, создав несколько простых ботов.

1. Мессенджер Telegram

Для начала разберемся, что такое Telegram, что такое telegram-bot и какие задачи можно решать с помощью этой технологии.

Телеграм

Телеграм — это мессенджер (программа для обмена сообщениями), реализованный по клиент-серверной архитектуре. Используя сервер для создания диалога между двумя клиентами, Телеграм пересылает через него или напрямую текстовые сообщения, а также изображения, видео или документы других форматов.

Подробнее о нем можно почитать **тут**.

Что же такое **телеграм-бот**?

Представьте, что у нас есть зарегистрированный аккаунт. Все сообщения, присылаемые на него, попадают на вход некоторой программе, а выход этой программы отправляется в виде сообщения тому, от кого пришло исходное сообщение. Это и есть бот. Иными словами, Телеграм-бот — это специальный пользователь, поведением которого управляет некоторая программа. Технически для сервера нет разницы, является данный пользователь человеком или ботом: для сервера оба клиента выглядят одинаково. Эта идеология очень похожа на пользователей-сообществ в vk.com.

API, которое мы изучим, позволит удобным образом получать сообщения от пользователей, обрабатывать их и отсылать ответы. Вся системная часть (получение ключей, шифрование, маршрутизация и т. д.) остается зоной ответственности библиотеки и архитектуры Telegram. Разработчику же дается удобный интерфейс, поэтому создателю нужно только реализовать логику поведения бота.

Какие задачи можно решать Телеграм-ботом?

Все ограничивается только фантазией. Приведем несколько примеров:

- Автоответчики. Все ситуации, когда требуется однозначный ответ на запрос. Например, бот может сообщать телефоны и другие контакты организации, ее рабочее время или предоставлять другую справочную информацию по запросу
- Интерфейс доступа к веб-сервисам. Бот может выполнять запросы к различным API и отдавать ответы в виде телеграм-сообщений
- Сценарии действий. Бот может пройти по какому-либо сценарию, задать пользователю определенные вопросы и собрать ответы на них. Например, при регистрации в каком-либо сервисе или при заявке на услугу
- Игры. Бот умеет пересылать картинки, поэтому можно создавать любые игры, не требующие мгновенной реакции. Например, подумайте, как реализовать игру в шахматы?
- Куда фантазия заведет (умные дома, управление автомобильной сигнализацией и т. д. Но помним о безопасности!)

2. Как завести пользователя в Телеграме?

Прежде чем мы сможем писать программу для бота, надо его зарегистрировать. Для этого нужно присвоить ему номер, который раздает «отец всех ботов». Тоже, кстати, бот, с именем **@BotFather** (такая вот рекурсия).

В режиме диалога он задаст несколько вопросов о назначении бота, поможет выбрать для него имя и выдаст специальный токен — уникальный ключ, с помощью которого в дальнейшем система будет идентифицировать нашего бота.

Подробная информация находится **здесь**. Очень рекомендуем предварительно с ней ознакомиться. Если кратко:

- Необходимо связаться с **@BotFather**, дать ему команду `/newbot` и ответить на его вопросы
- Придумать имя (name) для бота (это имя будет указано в чатах с ботом)
- Придумать системное имя (username) (оно будет использоваться для логина бота на сервер. Это имя должно быть уникальным и обязано оканчиваться на «bot»)

А дальше надо написать собственно бота. Давайте разберемся, как это делать.

3. API python-telegram-bot

Полное описание Telegram Bot API находится **здесь**.

Над API реализована обертка в виде библиотеки python-telegram-bot. Она делает написание кода более удобным. Как и в случае с vk.com, telegram предоставляет API в виде доступа к базовым примитивам системы. Это делается для того, чтобы дать пользователям API возможность максимально свободно и гибко применять возможности системы.

Однако на практике такая свобода и гибкость нужна далеко не всем. Подавляющее большинство программ, использующих API, повторяют одни и те же сценарии. При этом программисты вынуждены многократно повторять один и тот же код. Чтобы освободить разработку от этого, выпускается обертка над базовым API, реализующая большинство общих сценариев. При этом можно использовать и сам базовый API.

В итоге для обычных сценариев не приходится писать лишний код, но при этом можно сделать и то, что в эти базовые сценарии не вписывается.

Установим библиотеку:

```
pip install python-telegram-bot[ext] --upgrade
```

У этой библиотеки есть неплохая документация, которая находится **тут**. Кроме того, в репозитории библиотеки есть ряд **примеров программ**, изучая которые можно понять, как ей правильно пользоваться и как выглядят типовые операции.

Библиотека **python-telegram-bot версии 20** написана с использованием `asyncio.` Чтобы материал, изложенный в уроке был понятнее рекомендуем сначала изучить материалы урока **"Введение в асинхронное программирование"**.

4. Эхо-бот

Давайте разберем простейший пример.

Это **Эхо-бот**, то есть бот, который просто присылает полученное текстовое сообщение назад. Обычно такие боты используются системой для проверки связи.

Устройство телеграм-бота очень похоже на Flask веб-приложение (да и вообще на все то, чем мы занимаемся на втором году обучения): мы создаем обработчики для различных действий пользователя, а потом запускаем приложение, которое ждет этих действий и реагирует соответственно. В терминах библиотеки это выглядит следующим образом:

`Applicaton` → `Handlers` → `start` → `wait_for_the_end`

Входом в программу-бота является `Application`. Это объект, получающий на вход сообщения от telegram-сервера. Задача этого объекта — организация сетевого взаимодействия между клиентом и сервером.

Полученные сообщения `Application` передает `Dispatcher`'у, который автоматически создается внутри `Application`'а.

Диспечер отвечает за вызов обработчиков сообщений. Как и следует из названия объекта, он перенаправляет сообщения внутри программы (от англ. dispatch — пересылать).

Диспечер хранит обработчики сообщений — `handler`'ы. Это функции, а точнее, корутины, имеющие

определенную сигнатуру, получающие на вход сообщения уже определенных типов с информацией о том, от каких пользователей пришли данные сообщения.

Внутри `Application`'а реализован цикл приема-передачи сообщений.

Таким образом, программа-бот должна:

1. Создать объект `Application`, передав ему токен, полученный от @BotFather.
2. Получить корутины-обработчики для сообщений и команд и зарегистрировать их.
3. Запустить цикл приема и обработки сообщений.
4. Дожидаться окончания/прерывания работы программы.

Приведем пример кода, реализующего простейший эхо-бот, который объяснит эту теорию.

```
# Импортируем необходимые классы.
import logging
from telegram.ext import Application, MessageHandler, filters
from config import BOT_TOKEN

# Запускаем логгирование
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s', level=logging.DEBUG
)

logger = logging.getLogger(__name__)


# Определяем функцию-обработчик сообщений.
# У неё два параметра, updater, принявший сообщение и контекст - дополнительная информация о соо
async def echo(update, context):
    # У объекта класса Updater есть поле message,
    # являющееся объектом сообщения.
    # У message есть поле text, содержащее текст полученного сообщения,
    # а также метод reply_text(str),
    # отсылающий ответ пользователю, от которого получено сообщение.
    await update.message.reply_text(update.message.text)


def main():
    # Создаём объект Application.
    # Вместо слова "TOKEN" надо разместить полученный от @BotFather токен
    application = Application.builder().token(BOT_TOKEN).build()

    # Создаём обработчик сообщений типа filters.TEXT
    # из описанной выше асинхронной функции echo()
    # После регистрации обработчика в приложении
    # эта асинхронная функция будет вызываться при получении сообщения
    # с типом "текст", т. е. текстовых сообщений.
    text_handler = MessageHandler(filters.TEXT, echo)

    # Регистрируем обработчик в приложении.
    application.add_handler(text_handler)
```

```
    # Запускаем приложение.
    application.run_polling()


# Запускаем функцию main() в случае запуска скрипта.
if __name__ == '__main__':
    main()
```

Каждый из использованных классов подробно описан в документации:

- **Application**
- **MessageHandler**
- **Message**

Теперь бота можно запустить как обычный python-скрипт.

Найдите его по имени или системному имени из любого Telegram-клиента и попробуйте с ним пообщаться. В ответ на любое текстовое сообщение бот пришлет его назад.

5. Подключение через прокси-сервер

К сожалению, у некоторых провайдеров заблокированы интернет-адреса серверов, на которых размещено API Telegram. В этом случае необходимо либо разместить бота на ресурсе, с которого есть доступ к серверам, либо использовать подключение через промежуточный прокси-сервер. Сначала установим дополнительный модуль:

```
pip install python-telegram-bot[socks]
```

Теперь для того, чтобы подключиться через прокси, необходимо указать настройки сервера при создании `Application`:

```
from telegram.ext import ApplicationBuilder

proxy_url = "socks5://user:pass@host:port"

app = ApplicationBuilder().token("TOKEN").proxy_url(proxy_url).build()
```

6. Обработка команд

У внимательного ученика при виде строчки:

```
text_handler = MessageHandler(filters.TEXT, echo)
```

должно возникнуть два вопроса:

1. Какие еще типы обработчиков существуют?
2. Какие еще фильтры сообщений существуют?

Ответим сначала на первый вопрос.

Кроме сообщений бот может получать **команды**, **inline-запросы** и некоторые другие объекты. Все возможные варианты перечислены **здесь**.

Команда

Команда — это особое сообщение, начинающееся с символа «/». Команда предполагает какое-либо конкретное действие бота.

Есть несколько стандартных команд. Например:

- **/start** (команда начала общения с ботом, обычно она присылает сообщение о возможностях бота и как с ним общаться)
- **/help** (в ответ на команду бот присылает инструкцию по работе с собой)

Команды, на которые отвечает бот, могут быть любыми, в зависимости от назначения бота.

Добавим обработчики команд **/start** и **/help**.

По аналогии с обработкой сообщений, обработчик команд нужно создать из функции и зарегистрировать в приложении.

```
# Добавим необходимый объект из модуля telegram.ext
from telegram.ext import CommandHandler


# Напишем соответствующие функции.
# Их сигнатура и поведение аналогичны обработчикам текстовых сообщений.
async def start(update, context):
    """Отправляет сообщение когда получена команда /start"""
    user = update.effective_user
    await update.message.reply_html(
        rf"Привет {user.mention_html()}! Я эхо-бот. Напишите мне что-нибудь, и я пришлю это наза
    )


async def help_command(update, context):
    """Отправляет сообщение когда получена команда /help"""
    await update.message.reply_text("Я пока не умею помогать... Я только ваше эхо.")




# Зарегистрируем их в приложении перед
# регистрацией обработчика текстовых сообщений.
# Первым параметром конструктора CommandHandler я
# вляется название команды.
application.add_handler(CommandHandler("start", start))
application.add_handler(CommandHandler("help", help_command))
```

Также откорректируем строку `text_handler = MessageHandler(filters.TEXT, echo)` запишем ее так: `text_handler = MessageHandler(filters.TEXT & ~filters.COMMAND, echo)`, таким способом мы укажем, что обработчик **echo** будет использован только для обработки текстовых сообщений, но не для обработки команд.

7. Создание клавиатуры в диалоге пользователя

Одна из возможных задач бота — предоставлять разную справочную информацию. Мы уже знаем, что для реализации такого поведения можно завести специальные команды. Однако для пользователя вводить команды — дело сравнительно долгое. А если он пришел к боту с мобильного телефона, то и просто неудобное.

Для того чтобы справиться с этой трудностью, в API сделан механизм предразмеченных ответов. Если предполагается, что собеседник будет пользоваться какими-либо командами бота, можно вывести набор кнопок, каждая из которых присылает боту определенную команду. Это делает взаимодействие с ботом быстрее, удобнее и понятнее.

Давайте рассмотрим пример. Разработаем бот-справочник, который будет сообщать некоторую справочную информацию о фирме. Для начала: адрес, телефон, сайт и время работы.

Заведем четыре команды: /address, /phone, /site, /work_time, каждая из которых будет просто присылать пользователю текстовое сообщение с нужной информацией.

```
# Напишем соответствующие функции.
async def help(update, context):
    await update.message.reply_text(
        "Я бот справочник.")


async def address(update, context):
    await update.message.reply_text(
        "Адрес: г. Москва, ул. Льва Толстого, 16")


async def phone(update, context):
    await update.message.reply_text("Телефон: +7(495)776-3030")


async def site(update, context):
    await update.message.reply_text(
        "Сайт: http://www.yandex.ru/company")


async def work_time(update, context):
    await update.message.reply_text(
        "Время работы: круглосуточно.")


def main():
    application = Application.builder().token(BOT_TOKEN).build()
    application.add_handler(CommandHandler("address", address))
    application.add_handler(CommandHandler("phone", phone))
    application.add_handler(CommandHandler("site", site))
    application.add_handler(CommandHandler("work_time", work_time))
    application.add_handler(CommandHandler("help", help))
    application.run_polling()
```

Теперь создадим клавиатуру с четырьмя кнопками для этих команд. Для этого воспользуемся классом `ReplyKeyboardMarkup`. Чтобы им воспользоваться, нужно импортировать его из модуля telegram:

```
from telegram import ReplyKeyboardMarkup
```

Первым параметром конструктора `ReplyKeyboardMarkup` является список кнопок. Обратите внимание: в примере список состоит из двух подсписков, каждый из которых определяет строчку кнопок.

```
reply_keyboard = [['/address', '/phone'],
                  ['/site', '/work_time']]
markup = ReplyKeyboardMarkup(reply_keyboard, one_time_keyboard=False)
```

Если передать все четыре строчки в виде одного списка, то получим клавиатуру с четырьмя кнопками в одну строку.

Параметр `one_time_keyboard` указывает, нужно ли скрыть клавиатуру после нажатия на одну из кнопок (`one_time_keyboard = True`, клавиатура получается одноразовой) или не нужно (`one_time_keyboard = False`, как у нас в примере).

Для того чтобы клавиатура появилась в диалоге у пользователя, необходимо добавить ее в качестве параметра `reply_markup` в функцию `reply_text`. Тогда, помимо текста, API перешлет и размеченную клавиатуру.

```
async def start(update, context):
    await update.message.reply_text(
        "Я бот-справочник. Какая информация вам нужна?",
        reply_markup=markup
    )
```

Переданная однажды клавиатура будет оставаться в диалоге в свернутом или развернутом виде до тех пор, пока клиенту не перешлют новую или не укажут явно, что переданную клавиатуру надо удалить.

Для удаления нужно в качестве значения параметра `reply_markup` передать объект специального класса: `ReplyKeyboardRemove`.

```
from telegram import ReplyKeyboardRemove


async def close_keyboard(update, context):
    await update.message.reply_text(
        "Ok",
        reply_markup=ReplyKeyboardRemove()
    )


application.add_handler(CommandHandler("close", close_keyboard))
```

8. Установка и удаление таймера

Рассмотрим еще одну возможность API python-telegram-bot — установку таймера.

Таймер позволяет выполнить действие не сразу, а через некоторое время. Например, периодически проверять доступность какого-либо сайта, напомнить о чем-нибудь и т. д.

Для этих целей у бота есть очередь задач, в которую мы можем добавлять свои задачи. Делается это следующим образом:

```
TIMER = 5  # таймер на 5 секунд
```

```
def remove_job_if_exists(name, context):
    """Удаляем задачу по имени.
    Возвращаем True если задача была успешно удалена."""
    current_jobs = context.job_queue.get_jobs_by_name(name)
    if not current_jobs:
        return False
    for job in current_jobs:
        job.schedule_removal()
    return True


# Обычный обработчик, как и те, которыми мы пользовались раньше.
async def set_timer(update, context):
    """Добавляем задачу в очередь"""
    chat_id = update.effective_message.chat_id
    # Добавляем задачу в очередь
    # и останавливаем предыдущую (если она была)
    job_removed = remove_job_if_exists(str(chat_id), context)
    context.job_queue.run_once(task, TIMER, chat_id=chat_id, name=str(chat_id), data=TIMER)

    text = f'Вернусь через 5 с.!'
    if job_removed:
        text += ' Старая задача удалена.'
    await update.effective_message.reply_text(text)
```

Задача — это корутина с одним параметром — контекстом.

Обратите внимание: сообщение мы отправляем, используя объект `bot` внутри контекста, а не `updater`. Это обращение к базовому API.

```
async def task(context):
    """Выводит сообщение"""
    await context.bot.send_message(context.job.chat_id, text=f'КУКУ! 5с. прошли!')
```

Задачу из очереди можно отменить. Добавим для этого специальную команду:

```
async def unset(update, context):
    """Удаляет задачу, если пользователь передумал"""
    chat_id = update.message.chat_id
    job_removed = remove_job_if_exists(str(chat_id), context)
    text = 'Таймер отменен!' if job_removed else 'У вас нет активных таймеров'
    await update.message.reply_text(text)
```

Регистрируем обработчики.

```
    application.add_handler(CommandHandler("set", set_timer))
    application.add_handler(CommandHandler("unset", unset))
```

9. Создание сценариев диалогов

Обсуждая сферу применения ботов, мы говорили о разных сценариях, в которых пользователю последовательно задаются вопросы, а бот собирает ответы и что-то дальше с ними делает. Чем это отличается

от того, что мы делали раньше? Главным образом тем, что в предыдущем примере мы **не хранили контекст** «разговора». То есть бот «отвечал» на один вопрос и тут же «забывал», кто и о чем его спрашивал.

Сценарий — это серия вопросов или реплик, в которой бот «помнит», какие вопросы он уже задавал пользователю, какие ответы получил и что спрашивать дальше.

Для создания сценариев в telegram-bot-python есть специальный обработчик диалога: `ConversationHandler`.

Рассмотрим пример его использования.

```
    conv_handler = ConversationHandler(
        # Точка входа в диалог.
        # В данном случае — команда /start. Она задаёт первый вопрос.
        entry_points=[CommandHandler('start', start)],

        # Состояние внутри диалога.
        # Вариант с двумя обработчиками, фильтрующими текстовые сообщения.
        states={
            # Функция читает ответ на первый вопрос и задаёт второй.
            1: [MessageHandler(filters.TEXT & ~filters.COMMAND, first_response)],
            # Функция читает ответ на второй вопрос и завершает диалог.
            2: [MessageHandler(filters.TEXT & ~filters.COMMAND, second_response)]
        },

        # Точка прерывания диалога. В данном случае — команда /stop.
        fallbacks=[CommandHandler('stop', stop)]
    )

    application.add_handler(conv_handler)
```

В обычных обработчиках есть одна-единственная функция. Диспечер вызывает ее, если выполняется условие фильтра в обработчике. Сама же функция возвращает None, или, другими словами, не возвращает никакого значения.

Здесь мы регистрируем обработчик, состоящий из других обработчиков. Как же такой механизм работает?

В самом начале работы программы активен только обработчик `start`, описанный в параметре `entry_points`. Ровно так же, как если бы мы без создания `ConversationHandler` просто зарегистрировали бы его в Диспечере, как делали ранее.

Однако, в отличие от предыдущих случаев, обработчик `start` должен будет вернуть значение. И это значение укажет Диспечеру, какой обработчик выбрать для **последующих** сообщений.

В нашем случае мы вернем 1, и это укажет Диспечеру, что к следующему сообщению надо применить обработчик из параметра `states` с индексом 1 — `states[1]`. То есть тот, что связан с функцией `first_response()`. Получается, что, помимо обработки самой команды /start, мы еще и указываем Диспечеру, как работать дальше.

В свою очередь, `first_response()` вернет значение 2. После этого к следующим сообщениям Диспечер применит обработчик из `states[2]`.

Для окончания диалога нужно вернуть специальное значение `ConversationHandler.END`. После этого Диспечер будет, как и в самом начале, пробовать применить обработчик `entry_points`.

Обработчик из параметра `fallbacks` активен все время работы диалога и деактивируется после выхода из него. Он служит для прерывания диалога.

Давайте напишем упомянутые выше функции.

```
async def start(update, context):
    await update.message.reply_text(
        "Привет. Пройдите небольшой опрос, пожалуйста!\n"
        "Вы можете прервать опрос, послав команду /stop.\n"
        "В каком городе вы живёте?")

    # Число-ключ в словаре states —
    # втором параметре ConversationHandler'a.
    return 1
    # Оно указывает, что дальше на сообщения от этого пользователя
    # должен отвечать обработчик states[1].
    # До этого момента обработчиков текстовых сообщений
    # для этого пользователя не существовало,
    # поэтому текстовые сообщения игнорировались.


async def first_response(update, context):
    # Это ответ на первый вопрос.
    # Мы можем использовать его во втором вопросе.
    locality = update.message.text
    await update.message.reply_text(
        f"Какая погода в городе {locality}?")
    # Следующее текстовое сообщение будет обработано
    # обработчиком states[2]
    return 2


async def second_response(update, context):
    # Ответ на второй вопрос.
    # Мы можем его сохранить в базе данных или переслать куда-либо.
    weather = update.message.text
    logger.info(weather)
    await update.message.reply_text("Спасибо за участие в опросе! Всего доброго!")
    return ConversationHandler.END  # Константа, означающая конец диалога.
    # Все обработчики из states и fallbacks становятся неактивными.


async def stop(update, context):
    await update.message.reply_text("Всего доброго!")
    return ConversationHandler.END
```

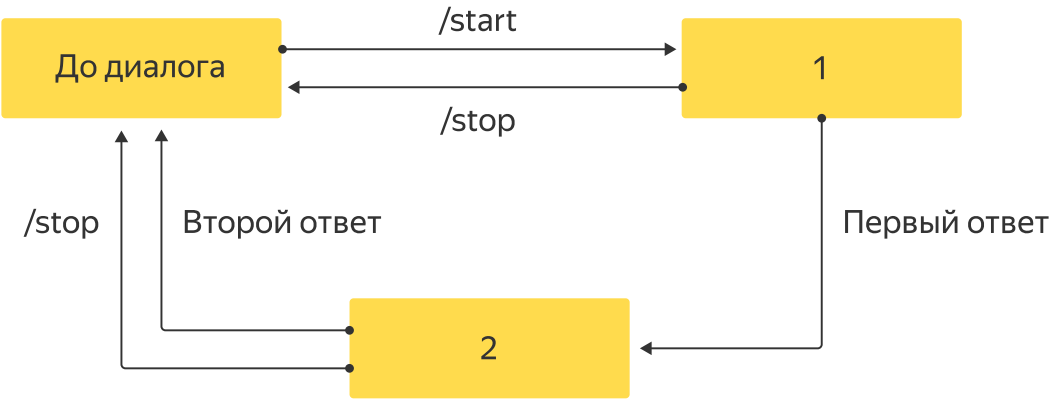
Рассмотрим таблицу состояний созданного сценария:

	До диалога	После /start	После first_response	После second_response	После /stop
/start	Активна	—	—	Активна	Активна
first_response	—	Активна	—	—	—
second_response	—	—	Активна	—	—
/stop	—	Активна	Активна	—	—

1. До начала диалога активна только команда /start. Остальные части диалога игнорируются.
2. После выполнения команда /start перестает работать, а активной становится команда /stop и обработчик `first_response`.
3. После выполнения обработчика `first_response` он деактивируется, а его место занимает обработчик `second_response`.
4. После выполнения обработчика `second_response` деактивируются он и команда /stop, и снова включается команда /start.

Пускай наш бот находится в каком-то состоянии и может перейти в другие состояния. Описывать возможности перехода можно либо таблицей состояний, как мы делали выше, либо диаграммой состояний. Диаграмма состояний — это схема, на которой изображены состояния (например, в виде прямоугольников или кружков) и стрелками отмечены доступные переходы в из одних состояний в другие. Часто над стрелками пишут условия, при которых осуществляется тот или иной переход.

Диаграмма состояний нашего бота изображена на рисунке ниже.



10. Передача пользовательских данных в сценарии

Часто нужно хранить не только предыдущий ответ пользователя, но и большее количество данных, полученных в ходе диалога.

Для хранения и передачи таких данных телеграм поддерживает специальный словарь `context.user_data`.

Модифицируем нашего бота так, чтобы он в конце диалога мог *передать привет* в город, который пользователь указал в первом ответе.

```
# Добавили словарь user_data в параметры.
async def first_response(update, context):
    # Сохраняем ответ в словаре.
    context.user_data['locality'] = update.message.text
    await update.message.reply_text(
        f"Какая погода в городе {context.user_data['locality']}?")
    return 2


# Добавили словарь user_data в параметры.
async def second_response(update, context):
    weather = update.message.text
```

```
    logger.info(weather)
    # Используем user_data в ответе.
    await update.message.reply_text(
        f"Спасибо за участие в опросе! Привет, {context.user_data['locality']}!")
    context.user_data.clear()  # очищаем словарь с пользовательскими данными
    return ConversationHandler.END
```

11. Использование HTTP-API в телеграм-ботах

Как вы понимаете, чат-боты могут работать не только с теми данными, которые находятся в их непосредственной доступности, но и запрашивать информацию у API сторонних сервисов. Идея проста: научим телеграм-бота «ходить» в HTTP API, превращая запросы пользователей в http-запросы к API, и транслировать ответы API в удобной и понятной для пользователя форме.

Давайте создадим бота, который по запросу пользователя присылает ему карту с запрошенным объектом. Например, пользователь запрашивает: «г. Москва, Тверская улица». В ответ высылается картинка с картой улицы.

Алгоритм работы может быть таким:

1. Бот «идет» с адресом в геокодер, получает координаты и размеры окна карты с нужным объектом.

2. С полученными координатами бот «идет» в StaticAPI и запрашивает по ним картинку карты.

Необходимо разобраться, как отправлять изображения (фотографии, в нотации телеграм-бота). Об этом можно почитать **здесь**.

Поскольку наш телеграм бот работает в асинхронном режиме, то применять стандартную библиотеку `requests` для обращения к сторонним api неправильно. Будем применять асинхронную библиотеку `aiohttp`, примеры ее использования есть в уроке **"Введение в асинхронное программирование"**.

Иногда получение ответа от стороннего API может требовать времени, и в синхронном режим эта операция становится блокирующей, т.е. пока не будет обработан запрос от одного пользователя, запросы и команды от остальных пользователей будут проигнорированы, точнее, будут ждать своей очереди на обработку. В асинхронном режиме пока наш бот ждет ответа от стороннего сервиса, он продолжает принимать и обрабатывать команды от пользователей в обычном режиме.

Функция, обрабатывающая сообщения, получится примерно следующая:

```
async def geocoder(update, context):
    geocoder_uri = "http://geocode-maps.yandex.ru/1.x/"
    response = await get_response(geocoder_uri, params={
        "apikey": "40d1649f-0493-4b70-98ba-98533de7710b",
        "format": "json",
        "geocode": update.message.text
    })

    toponym = response["response"]["GeoObjectCollection"][
        "featureMember"][0]["GeoObject"]
    ll, spn = get_ll_spn(toponym)
    # Можно воспользоваться готовой функцией,
    # которую предлагалось сделать на уроках, посвящённых HTTP-геокодеру.

    static_api_request = f"http://static-maps.yandex.ru/1.x/?ll={ll}&spn={spn}&l=map"
```

```
    await context.bot.send_photo(
        update.message.chat_id,  # Идентификатор чата. Куда посылать картинку.
        # Ссылка на static API, по сути, ссылка на картинку.
        # Телеграму можно передать прямо её, не скачивая предварительно карту.
        static_api_request,
        caption="Нашёл:"
    )


async def get_response(url, params):
    logger.info(f"getting {url}")
    async with aiohttp.ClientSession() as session:
        async with session.get(url, params=params) as resp:
            return await resp.json()
```