**Аннотация**

*В уроке рассказывается про реализацию 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.*

**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](https://ru.wikipedia.org/wiki/Telegram).

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!)

**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](https://core.telegram.org/bots#6-botfather). Очень рекомендуем предварительно с ней ознакомиться. Если кратко:

* Необходимо связаться с [@BotFather](https://telegram.me/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.

**API python-telegram-bot**

Полное описание Telegram Bot API находится [здесь](https://core.telegram.org/bots/api).

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

У этой библиотеки есть неплохая документация, которая находится [тут](https://python-telegram-bot.readthedocs.io/en/stable/). Кроме того, в репозитории библиотеки есть ряд [примеров программ](https://github.com/python-telegram-bot/python-telegram-bot/blob/master/examples/README.md), изучая которые можно понять, как ей правильно пользоваться и как выглядят типовые операции.

**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:

Updater → Dispatсher → Handlers → start → wait\_for\_the\_end

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

Полученные сообщения Updater передает Dispatcher’у, который автоматически создается внутри Updater’а.

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

Диспетчер хранит обработчики сообщений — handler’ы. Это функции, имеющие определенную сигнатуру, получающие на вход сообщения уже определенных типов с информацией о том, от каких пользователей пришли данные сообщения.

Внутри Updater’а реализован цикл приема-передачи сообщений, который запускается в отдельном потоке специальным вызовом.

Таким образом, программа-бот должна:

1. Создать объект Updater, передав ему токен, полученный от @BotFather.
2. Получить от Updater’а Dispatcher.
3. Получить функции-обработчики для сообщений и команд и зарегистрировать их в диспетчере.
4. Запустить цикл приема и обработки сообщений.
5. 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 Updater, MessageHandler, Filters

# Запускаем логгирование

logging.basicConfig(

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

)

logger = logging.getLogger(\_\_name\_\_)

TOKEN = 'BOT\_TOKEN'

# Определяем функцию-обработчик сообщений.

# У неё два параметра, сам бот и класс updater, принявший сообщение.

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.

update.message.reply\_text(update.message.text)

def main():

# Создаём объект updater.

# Вместо слова "TOKEN" надо разместить полученный от @BotFather токен

updater = Updater(TOKEN)

# Получаем из него диспетчер сообщений.

dp = updater.dispatcher

# Создаём обработчик сообщений типа Filters.text

# из описанной выше функции echo()

# После регистрации обработчика в диспетчере

# эта функция будет вызываться при получении сообщения

# с типом "текст", т. е. текстовых сообщений.

text\_handler = MessageHandler(Filters.text, echo)

# Регистрируем обработчик в диспетчере.

dp.add\_handler(text\_handler)

# Запускаем цикл приема и обработки сообщений.

updater.start\_polling()

# Ждём завершения приложения.

# (например, получения сигнала SIG\_TERM при нажатии клавиш Ctrl+C)

updater.idle()

# Запускаем функцию main() в случае запуска скрипта.

if \_\_name\_\_ == '\_\_main\_\_':

main()

Каждый из использованных классов подробно описан в документации:

* [Updater](https://python-telegram-bot.readthedocs.io/en/stable/telegram.ext.updater.html)
* [Dispatcher](https://python-telegram-bot.readthedocs.io/en/stable/telegram.ext.dispatcher.html)
* [MessageHandler](https://python-telegram-bot.readthedocs.io/en/stable/telegram.ext.messagehandler.html)
* [Message](https://python-telegram-bot.readthedocs.io/en/stable/telegram.message.html)
* [Bot](https://python-telegram-bot.readthedocs.io/en/stable/telegram.bot.html)

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.

**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 Updater в параметре request\_kwargs:

REQUEST\_KWARGS = {

'proxy\_url': 'socks5://ip:port', # Адрес прокси сервера

# Опционально, если требуется аутентификация:

# 'urllib3\_proxy\_kwargs': {

# 'assert\_hostname': 'False',

# 'cert\_reqs': 'CERT\_NONE'

# 'username': 'user',

# 'password': 'password'

# }

}

updater = Updater(TOKEN, use\_context=True,

request\_kwargs=REQUEST\_KWARGS)

**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](https://python-telegram-bot.readthedocs.io/en/stable/telegram.ext.html#handlers).

**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 диспетчере.

# Добавим необходимый объект из модуля telegram.ext

from telegram.ext import CommandHandler

# Напишем соответствующие функции.

# Их сигнатура и поведение аналогичны обработчикам текстовых сообщений.

def start(update, context):

update.message.reply\_text(

"Привет! Я эхо-бот. Напишите мне что-нибудь, и я пришлю это назад!")

def help(update, context):

update.message.reply\_text(

"Я пока не умею помогать... Я только ваше эхо.")

# Зарегистрируем их в диспетчере рядом

# с регистрацией обработчиков текстовых сообщений.

# Первым параметром конструктора CommandHandler я

# вляется название команды.

dp.add\_handler(CommandHandler("start", start))

dp.add\_handler(CommandHandler("help", help))

Также откорректируем строку text\_handler = MessageHandler(Filters.text, echo) запишем ее так: text\_handler = MessageHandler(Filters.text & ~Filters.command, echo), таким способом мы укажем, то обработчик **echo** будет использован только для обработки текстовых сообщений, но не для обработки команд.

**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.

def help(update, context):

update.message.reply\_text(

"Я бот справочник.")

def address(update, context):

update.message.reply\_text(

"Адрес: г. Москва, ул. Льва Толстого, 16")

def phone(update, context):

update.message.reply\_text("Телефон: +7(495)776-3030")

def site(update, context):

update.message.reply\_text(

"Сайт: http://www.yandex.ru/company")

def work\_time(update, context):

update.message.reply\_text(

"Время работы: круглосуточно.")

def main():

updater = Updater(TOKEN)

dp = updater.dispatcher

dp.add\_handler(CommandHandler("address", address))

dp.add\_handler(CommandHandler("phone", phone))

dp.add\_handler(CommandHandler("site", site))

dp.add\_handler(CommandHandler("work\_time", work\_time))

dp.add\_handler(CommandHandler("help", help))

updater.start\_polling()

updater.idle()

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.

def start(update, context):

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

def close\_keyboard(update, context):

update.message.reply\_text(

"Ok",

reply\_markup=ReplyKeyboardRemove()

)

dp.add\_handler(CommandHandler("close", close\_keyboard))

**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:

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.

def set\_timer(update, context):

"""Добавляем задачу в очередь"""

chat\_id = update.message.chat\_id

try:

# args[0] должен содержать значение аргумента

# (секунды таймера)

due = int(context.args[0])

if due < 0:

update.message.reply\_text('Извините, не умеем возвращаться в прошлое')

return

# 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, due, context=chat\_id, name=str(chat\_id))

text = f'Вернусь через {due} секунд!'

if job\_removed:

text += ' Старая задача удалена.'

update.message.reply\_text(text)

except (IndexError, ValueError):

update.message.reply\_text('Использование: /set <секунд>')

Задача — это функция с одним параметром — контекстом.

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

def task(context):

"""Выводит сообщение"""

job = context.job

context.bot.send\_message(job.context, text='КУКУ!')

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

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'

update.message.reply\_text(text)

Регистрируем обработчики. Обратите внимание на параметры pass\_job\_queue и pass\_chat\_data.

По умолчанию они равны False и соответствующие параметры не передаются в обработчики. Сейчас же они нам нужны, мы явно задаем их значения и прописываем соответствующие параметры в прототипы обработчиков.

dp.add\_handler(CommandHandler("set", set\_timer,

pass\_args=True,

pass\_job\_queue=True,

pass\_chat\_data=True))

dp.add\_handler(CommandHandler("unset", unset,

pass\_chat\_data=True)

)

**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)]

)

dp.add\_handler(conv\_handler)

В обычных обработчиках есть одна-единственная функция. Диспетчер 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 диспетчере, как делали ранее.

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

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

В свою очередь, first\_response() вернет значение 2. После этого к следующим сообщениям диспетчер применит обработчик из states[2].

Для окончания диалога нужно вернуть специальное значение ConversationHandler.END. После этого диспетчер 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.

def start(update, context):

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'а.

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.

def first\_response(update, context):

# Это ответ на первый вопрос.

# Мы можем использовать его во втором вопросе.

locality = update.message.text

update.message.reply\_text(

f"Какая погода в городе {locality}?")

# Следующее текстовое сообщение будет обработано

# обработчиком states[2]

return 2

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)

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.

def stop(update, context):

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.

**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. Для получения доступа к нему необходимо при создании соответствующего обработчика указать параметр pass\_user\_data=True.

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.

def first\_response(update, context):

# Сохраняем ответ в словаре.

context.user\_data['locality'] = update.message.text

update.message.reply\_text(

f"Какая погода в городе {context.user\_data['locality']}?")

return 2

# Добавили словарь user\_data в параметры.

def second\_response(update, context):

weather = update.message.text

logger.info(weather)

# Используем user\_data в ответе.

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

< ... >

conv\_handler = ConversationHandler(

entry\_points=[CommandHandler('start', start)],

states={

# Добавили user\_data для сохранения ответа.

1: [MessageHandler(Filters.text & ~Filters.command, first\_response, pass\_user\_data=True)],

# ...и для его использования.

2: [MessageHandler(Filters.text & ~Filters.command, second\_response, pass\_user\_data=True)]

},

fallbacks=[CommandHandler('stop', stop)]

)

**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](https://python-telegram-bot.readthedocs.io/en/stable/telegram.bot.html#telegram.Bot.sendPhoto).

Функция, обрабатывающая сообщения, получится примерно следующая:

def geocoder(update, context):

geocoder\_uri = geocoder\_request\_template = "http://geocode-maps.yandex.ru/1.x/"

response = requests.get(geocoder\_uri, params={

"apikey": "40d1649f-0493-4b70-98ba-98533de7710b",

"format": "json",

"geocode": update.message.text

})

toponym = response.json()["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"

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:"

)