Сообщить об ошибке.

Библиотека python-telegram-bot в Python

Создание Telegram каналов и ботов

Пакет python-telegram-bot содержит ряд высокоуровневых классов, которые делают разработку ботов простой и понятной. Эти классы содержатся в модуле telegram.ext. Он совместим с версиями Python 3.7+. Пакет python-telegram-bot также может работать с PyPy3 (официально не поддерживается), хотя раньше было много проблем.

> ***Внимание***! *Пакеты python-telegram-bot версии 13.x будут придерживаться многопоточной парадигмы программирования (на данный момент актуальна версия 13.15). Пакеты версий 20.x и новее предоставляют чистый асинхронный Python интерфейс для Telegram Bot API. Дополнительно смотрите основные изменения в пакете python-telegram-bot версии 20.x.*

Установка пакета python-telegram-bot в виртуальное окружение:

Пакет python-telegram-bot пытается использовать как можно меньше сторонних зависимостей. Однако для некоторых функций использование сторонней библиотеки более разумно, чем повторная реализация функциональности. Поскольку эти функции являются необязательными, соответствующие сторонние зависимости не устанавливаются по умолчанию. При установке пакета их можно перечислены как необязательные зависимости. Это позволяет избежать ненужных конфликтов зависимостей для пользователей, которым не нужны дополнительные функции.

Единственная необходимая зависимость - это httpx ~= 0.23.3 для telegram.request.HTTPXRequest, сетевого бэкенда по умолчанию.

```
# создаем виртуальное окружение, если нет
$ python3 -m venv .telegram --prompt TelegramBot
# активируем виртуальное окружение
$ source .telegram/bin/activate
# ставим последнюю многопоточную версию python-telegram-bot
(TelegramBot):~$ python3 -m pip install python-telegram-bot==13.15 -U
# или установка асинхронной версии со всеми зависимостями
(TelegramBot):~$ python3 -m pip install python-telegram-bot[all] -U
# устанавливает все необязательные зависимости расширения `telegram.ext`
(TelegramBot):~$ python3 -m pip install python-telegram-bot[ext] -U
# При установке зависимости можно перечислить отдельно как:
# python-telegram-bot[rate-limiter,webhooks,callback-data,job-queue]
```

Дополнительные зависимости:

- pip install python-telegram-bot[passport] - устанавливает библиотеку cryptography. Используется для функции, связанных с Telegram Passport.
- pip install python-telegram-bot[socks] - устанавливает httpx[socks]. Используется для работы с сервером Socks5.
- pip install python-telegram-bot[http2] - устанавливает httpx[http2]. Используется для работы с сервером HTTP/2.
- pip install python-telegram-bot[rate-limiter] - устанавливает aiolimiter. Используется для работы с telegram.ext.AIORateLimiter.
- pip install python-telegram-bot[webhooks] - устанавливает библиотеку tornado. Используется для работы с telegram.ext.Updater.start_webhook/telegram.ext.Application.run_webhook.
- pip install python-telegram-bot[callback-data] - устанавливает библиотеку cachetools. Используется для работы с вольными callback_data.

Вверх

- pip install python-telegram-bot[job-queue] - устанавливает библиотеку APScheduler и применяет pytz, где pytz является зависимостью APScheduler. Используется для работы с telegram.ext.JobQueue.

Чтобы установить несколько необязательных зависимостей, разделите их запятыми, например. pip install python-telegram-bot[socks,webhooks].

Пакет python-telegram-bot в основном будет разбираться на примерах. Содержание, обзорного/вводного материала по библиотеке ниже. Меню с материалами по всему разделу - справа.

Высокоуровневый интерфейс пакета python-telegram-bot построен поверх чистой реализации Telegram Bot API и находится в подмодуле расширений telegram.ext. Он предоставляет простой в использовании интерфейс и снимает с программиста некоторую работу.

> *Для выполнения примеров требуется сгенерировать токен доступа к API. Для этого необходимо пообщаться с @BotFather и выполнить несколько простых шагов, описанных в разделе Команды и оповещения @BotFather в Telegram.*

Содержание:

Многопоточный модуль расширения telegram.ext (*версия 13.x*).

Модуль расширений telegram.ext состоит из нескольких классов, но два наиболее важных - это telegram.ext.Updater и telegram.ext.Dispatcher.

Класс Updater постоянно слушает сервер Telegram, получает новые сообщения и передает их классу Dispatcher. Если создать объект Updater, то он автоматически создаст Dispatcher и свяжет их вместе с очередью. Затем в объекте Dispatcher можно зарегистрировать обработчики разных типов, которые будут сортировать полученные объектом Updater сообщения. Поступающие сообщения будут обрабатываться в соответствии с зарегистрированными обработчиками и передавать их в функцию обратного вызова, которую необходимо определить.

Еще нужно знать и понимать, что экземпляр Updater реализует все методы класса telegram.Bot (API Telegram), которые будут связаны с данным Updater. У экземпляра Dispatcher, в свою очередь, есть так называемый контекст context, который, при регистрации любого обработчика сообщений передается в функцию обратного вызова этого обработчика (кстати в нее так же передается updater). Так вот, у этого контекста то же есть экземпляр класса telegram.Bot, только он связан с конкретным сообщением, которое попало в эту функцию обратного вызова.

Каждый обработчик является экземпляром подкласса класса telegram.ext.Handler. Пакет python-telegram-bot предоставляет классы обработчиков почти на все стандартные случаи, но если нужно что-то конкретное, то можно создать собственный обработчик, наследуясь от класса Handler.

Создание Telegram bot, шаг за шагом.

Во-первых, нужно создать объект Updater. В коде ниже замените константу TOKEN на API-токен вашего бота. Для более быстрого доступа к Dispatcher, в который Updater посылает сообщение, можно создать его отдельно:

```
from telegram.ext import Updater
TOKEN = 'Замените эту строку на token, полученный от @BotFather'
# получаем экземпляр `Updater`
updater = Updater(token=TOKEN, use_context=True)
# получаем экземпляр `Dispatcher`
dispatcher = updater.dispatcher
```

> *Примечание. Аргумент use_context=True (по умолчанию False) - это специальный аргумент, необходимый только для python-telegram-bot меньше версии 12.x. Это обеспечивает лучшую обратную совместимость со старыми версиями и дает пользователям время для обновления. Начиная с версии 13.x, значение аргумента use_context=True используется по умолчанию (указывать его не нужно).*

Чтобы знать, когда и почему что-то не работает должным образом, настроим модуль ведения журнала логов:

```
import logging
logging.basicConfig(format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
```

```
                    level=logging.INFO)
```

Примечание. если хотите узнать больше об обработке исключений с python-telegram-bot, прочтите подраздел об "Обработка исключений".

Теперь определим функцию, которая должна обрабатывать определенный тип сообщения, отправленных боту:

```
# Обратите внимание, что из обработчика в функцию
# передаются экземпляры `update` и `context`
def start(update, context):
    # `bot.send_message` это метод Telegram API
    # `update.effective_chat.id` - определяем `id` чата,
    # откуда прилетело сообщение
    context.bot.send_message(chat_id=update.effective_chat.id,
                             text="I'm a bot, please talk to me!")
```

> *Примечание. Обратите внимание, аргументы update и context функции обратного вызова start():*
>
> - *аргументы update и context передаются автоматически;*
> - *update - это объект связанный с экземпляром Update который присылает и отправляет **все сообщения.** Через него можно получить доступ к экземпляру telegram.Bot() как update.bot;*
> - *context - это объект связанный с контекстом **обработанного сообщения.** Через него также можно получить доступ к экземпляру telegram.Bot() как context.bot.*

Цель состоит в том, чтобы эта функция вызывалась каждый раз, когда бот получает сообщение с серверов Telegram, содержащее команду /start. Для этого можно использовать класс CommandHandler (один из предоставленных подклассов Handler) и зарегистрировать его в Dispatcher:

```
# импортируем обработчик CommandHandler,
# который фильтрует сообщения с командами
from telegram.ext import CommandHandler
# говорим обработчику, если увидишь команду `/start`,
# то вызови функцию `start()`
start_handler = CommandHandler('start', start)
# добавляем этот обработчик в `dispatcher`
dispatcher.add_handler(start_handler)
```

И это все, что нужно! Для запуска бота дописываем команду:

```
# говорим экземпляру `Updater`,
# слушай сервера Telegram.
updater.start_polling()
```

Начните чат со своим ботом и введите команду /start - если все пойдет хорошо, он ответит.

Созданный бот может отвечать только на команду /start. Добавим еще один обработчик, который прослушивает обычные сообщения. Для этого используем класс MessageHandler - другой подкласс Handler, для вывода всех текстовых сообщений:

```
# функция обратного вызова
def echo(update, context):
    # добавим в начало полученного сообщения строку 'ECHO: '
    text = 'ECHO: ' + update.message.text
    # `update.effective_chat.id` - определяем `id` чата,
    # откуда прилетело сообщение
    context.bot.send_message(chat_id=update.effective_chat.id,
                             text=text)

# импортируем обработчик `MessageHandler` и класс с фильтрами
from telegram.ext import MessageHandler, Filters
# говорим обработчику `MessageHandler`, если увидишь текстовое
# сообщение (фильтр `Filters.text`)  и это будет не команда
# (фильтр ~Filters.command), то вызови функцию `echo()`
echo_handler = MessageHandler(Filters.text & (~Filters.command), echo)
# регистрируем обработчик `echo_handler` в экземпляре `dispatcher`
dispatcher.add_handler(echo_handler)
```

С этого момента создаваемый бот должен обрабатывать все получаемые текстовые сообщения, а так же работать с командой /start, но не будет реагировать на любые другие команды (например, /your_command) .

Примечание: как только новые обработчики добавляются в диспетчер, они сразу вступают в силу.

Вверх

Примечание. Класс telegram.ext.Filters содержит ряд так называемых фильтров, которые фильтруют входящие сообщения по тексту, изображениям, обновлениям статуса и т. д. Любое сообщение, которое возвращает True хотя бы для одного из фильтров, переданных в MessageHandler, будет принято. Если необходимо, то можно написать свои собственные фильтры. Подробнее смотрите раздел "Все о фильтрации сообщений python-telegram-bot в Python".

Добавим боту другую функциональность и реализуем команду /caps, которая будет принимать какой-то текст в качестве аргумента и отвечать на него тем же текстом, только в верхнем регистре. Аргументы команды (например /caps any args) будут поступать в функцию обратного вызова в виде списка ['any', 'args'], разделенного по пробелам:

```
def caps(update, context):
    # если аргументы присутствуют
    if context.args:
        # объединяем список в строку и
        # переводим ее в верхний регистр
        text_caps = ' '.join(context.args).upper()
        # `update.effective_chat.id` - определяем `id` чата,
        # откуда прилетело сообщение
        context.bot.send_message(chat_id=update.effective_chat.id,
                                 text=text_caps)
    else:
        # если в команде не указан аргумент
        context.bot.send_message(chat_id=update.effective_chat.id,
                                 text='No command argument')
        context.bot.send_message(chat_id=update.effective_chat.id,
                                 text='send: /caps argument')

# обработчик команды '/caps'
caps_handler = CommandHandler('caps', caps)
# регистрируем обработчик в диспетчере
dispatcher.add_handler(caps_handler)
```

Примечание. Обратите внимание на использование context.args. Объект CallbackContext будет иметь много разных атрибутов в зависимости от того, какой обработчик используется.

Режим встроенных запросов.

Еще одна интересная особенность официального Telegram Bot API - это режим встроенных запросов к ботам. Помимо отправки команд в личных сообщениях или группах, пользователи могут взаимодействовать с ботом с помощью встроенных запросов. Если встроенные запросы включены, то пользователи могут вызвать бота, введя его имя @bot_username и запрос в поле ввода текста в любом чате. Запрос отправляется боту в обновлении. Таким образом, люди могут запрашивать контент у ботов в любом из своих чатов, групп или каналов, вообще не отправляя им никаких отдельных сообщений.

Если необходимо реализовать такую функциональность для своего бота, то сначала необходимо изменить конфигурацию в @BotFather, включив этот режим при помощи команды /setinline. Иногда требуется какое-то время, пока бот не зарегистрируется в качестве встроенного бота на вашем клиенте. Можно ускорить процесс, перезапустив приложение Telegram или иногда просто нужно немного подождать.

Здесь используется ряд новых типов:

```
from telegram import InlineQueryResultArticle, InputTextMessageContent
def inline_caps(update, context):
    query = update.inline_query.query
    if not query:
        return
    results = list()
    results.append(
        InlineQueryResultArticle(
            id=query.upper(),
            title='Convert to UPPER TEXT',
            input_message_content=InputTextMessageContent(query.upper())
        )
    )
    context.bot.answer_inline_query(update.inline_query.id, results)

from telegram.ext import InlineQueryHandler
inline_caps_handler = InlineQueryHandler(inline_caps)
dispatcher.add_handler(inline_caps_handler)
```

Теперь бот может работать и через режим встроенных запросов.

Вверх

Пользователи могут попытаться отправить боту команды, которые он не понимает, поэтому можно использовать обработчик MessageHandler с фильтром Filters.command, чтобы отвечать на все команды, которые не были распознаны предыдущими обработчиками.

```
def unknown(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id,
                             text="Sorry, I didn't understand that command.")

unknown_handler = MessageHandler(Filters.command, unknown)
dispatcher.add_handler(unknown_handler)
```

> *Примечание. Этот обработчик должен быть добавлен последним. Если его поставить первым, то он будет срабатывать до того, как обработчик CommandHandlers для команды /start увидит обновление. После обработки обновления функцией unknown() все дальнейшие обработчики будут игнорироваться.*
>
> *Чтобы обойти такое поведение, можно передать в метод dispatcher.add_handler(handler, group), помимо самой функции обработчика аргумент group со значением, отличным от 0. Аргумент group можно воспринимать как число, которое указывает приоритет обновления обработчика. Более низкая группа означает более высокий приоритет. Обновление может обрабатываться (максимум) одним обработчиком в каждой группе.*

Остановить бота можно командой updater.stop().

Примечание. Объект Updater запускается в отдельном потоке, это хорошо, если вы запускаете команды в интерпретаторе Python. Но если запустить скрипт с написанным ботом, то вероятно, удобнее будет останавливать бота, нажатием Ctrl + C , отправив сигнал процессу бота. Для этого, после запуска бота командой updater.start_polling() допишите в коде следующей строкой команду updater.idle().

Весь код созданного бота:

```
# sample-bot.py
from telegram import InlineQueryResultArticle, InputTextMessageContent
from telegram.ext import Updater, CommandHandler
from telegram.ext import MessageHandler, Filters, InlineQueryHandler

TOKEN = 'Замените эту строку на token, полученный от @BotFather'
updater = Updater(token=TOKEN)
dispatcher = updater.dispatcher

# функция обработки команды '/start'
def start(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id,
                             text="I'm a bot, please talk to me!")

# функция обработки текстовых сообщений
def echo(update, context):
    text = 'ECHO: ' + update.message.text
    context.bot.send_message(chat_id=update.effective_chat.id,
                             text=text)

# функция обработки команды '/caps'
def caps(update, context):
    if context.args:
        text_caps = ' '.join(context.args).upper()
        context.bot.send_message(chat_id=update.effective_chat.id,
                                 text=text_caps)
    else:
        context.bot.send_message(chat_id=update.effective_chat.id,
                                 text='No command argument')
        context.bot.send_message(chat_id=update.effective_chat.id,
                                 text='send: /caps argument')

# функция обработки встроенного запроса
def inline_caps(update, context):
    query = update.inline_query.query
    if not query:
        return
    results = list()
    results.append(
        InlineQueryResultArticle(
            id=query.upper(),
            title='Convert to UPPER TEXT',
```

Вверх

```
            input_message_content=InputTextMessageContent(query.upper())
        )
    )
    context.bot.answer_inline_query(update.inline_query.id, results)

# функция обработки не распознных команд
def unknown(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id,
                             text="Sorry, I didn't understand that command.")

# обработчик команды '/start'
start_handler = CommandHandler('start', start)
dispatcher.add_handler(start_handler)

# обработчик текстовых сообщений
echo_handler = MessageHandler(Filters.text & (~Filters.command), echo)
dispatcher.add_handler(echo_handler)

# обработчик команды '/caps'
caps_handler = CommandHandler('caps', caps)
dispatcher.add_handler(caps_handler)

# обработчик встроенных запросов
inline_caps_handler = InlineQueryHandler(inline_caps)
dispatcher.add_handler(inline_caps_handler)

# обработчик не распознанных команд
unknown_handler = MessageHandler(Filters.command, unknown)
dispatcher.add_handler(unknown_handler)

# запуск прослушивания сообщений
updater.start_polling()
# обработчик нажатия Ctrl+C
updater.idle()
```

Асинхронный модуль расширения telegram.ext (*версия 20.x*).

Асинхронный модуль расширений telegram.ext состоит из нескольких классов, но здесь самый важный - это telegram.ext.Application.

Класс Application отвечает за получение обновлений из очереди update_queue, где класс Updater постоянно получает новые обновления из Telegram и добавляет их в эту очередь. Если создать объект приложения с помощью ApplicationBuilder, то он автоматически создаст средство обновления и свяжет их вместе с помощью asyncio.Queues. Затем можно зарегистрировать обработчики различных типов в приложении, которое будут сортировать обновления, полученные средством обновления, в соответствии с зарегистрированными обработчиками и доставлять их в определенную функцию обратного вызова.

Каждый обработчик является экземпляром любого подкласса класса telegram.ext.BaseHandler. Библиотека предоставляет классы-обработчики практически для всех случаев использования, но если нужно что-то очень конкретное, то можете сами создать подкласс Handler.

Создание асинхронного Telegram bot, шаг за шагом.

Первое. В описании работы асинхронного бота, в коде будут комментироваться только отличительные моменты. Если что-то непонятно - смотрим сначала создание многопоточного бота.

Второе. для выполнения примеров создадим новый файл, например sync-bot.py. По ходу материала в этот файл будем добавлять новый код несколько раз. Другими словами, для краткости не будем повторять весь код создаваемого бота каждый раз, когда что-то добавляем.

И третье, самое важное!!! Для создания асинхронного бота (без наворотов) достаточно понимать и учитывать в коде следующие вещи:

- Все функции-обработчики становятся сопрограммами, т.е. добавляется асинхронный оператор async перед определением функции-обработчика.
- Внутри функции-обработчика появляется асинхронный оператор await, который ставится перед методами объектов context или update, ожидающие своей очереди для каких-то *сетевых операций* (например, отправить/ответить/изменить сообщение и т.д.). Объекты context или update передаются в качестве аргументов функциям-обработчикам.

Вверх

Важно! Здесь нужно четко понимать какие методы ожидают подключения к сети. Например, метод context.bot.send_message() или update.message.reply_text() явно хотят что-то отправить по сети в чат, следовательно перед ним ставиться await. А вот свойство update.message.chat_id или update.effective_chat.id просто извлекают значение chat_id из словаря, полученного во время *фоновой операции Update* и не являются сопрограммами - перед ними оператор await НЕ СТАВИТЬСЯ.

- Создание экземпляра приложения Telegram осуществляется вызовом одного класса ApplicationBuilder() или Application.builder(), а настройка (параметров по умолчанию) происходит вызовом цепочки методов этого экземпляра.

И так, вставим в файл следующий код:

```
import logging
from telegram import Update
from telegram.ext import ApplicationBuilder, ContextTypes, CommandHandler

# настроим модуль ведения журнала логов
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    level=logging.INFO
)

# определяем асинхронную функцию
async def start(update, context):
    # ожидание отправки сообщения по сети - нужен `await`
    await context.bot.send_message(chat_id=update.effective_chat.id,
                                   text="I'm a bot, please talk to me!")

if __name__ == '__main__':
    TOKEN = 'Замените эту строку на token, полученный от @BotFather'
    # создание экземпляра бота через `ApplicationBuilder`
    application = ApplicationBuilder().token(TOKEN).build()

    # создаем обработчик для команды '/start'
    start_handler = CommandHandler('start', start)
    # регистрируем обработчик в приложение
    application.add_handler(start_handler)
    # запускаем приложение
    application.run_polling()
```

Переварим написанный код, пройдемся по нему шаг за шагом.

```
import logging
# настройки модуля ведения журнала
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    level=logging.INFO
)
```

Вышеуказанный код предназначена для настройки модуля ведения журнала, чтобы знать, когда (и почему) что-то не работает должным образом:

```
TOKEN = 'Замените эту строку на token, полученный от @BotFather'
# создание экземпляра бота через `ApplicationBuilder`
application = ApplicationBuilder().token(TOKEN).build()
```

Здесь создается объект Application. Константу TOKEN необходимо заменить API-токеном своего бота.

Приложение само по себе ничего не делает. Чтобы добавить функциональность, необходимо сделать две вещи. Во-первых, определить функцию, которая должна обрабатывать определенный тип обновления:

```
async def start(update, context):
    await context.bot.send_message(
        chat_id=update.effective_chat.id,
        text="I'm a bot, please talk to me!"
    )
```

> *Примечание. Обратите внимание, аргументы update и context функции обратного вызова start():*
>
> - *аргументы update и context передаются автоматически;*
> - *update - это объект связанный с экземпляром Update который присылает и отправляет **все сообщения**. Через него можно получить доступ к экземпляру telegram.Bot() как update.bot;*
> - *context - это объект связанный с контекстом **обработанного сообщения**. Через него также можно получить доступ к экземпляру telegram.Bot() как context.bot.*

Вверх

Цель состоит в том, чтобы асинхронная функция start() вызывалась каждый раз, когда бот получает сообщение от сервера Telegram, содержащее команду /start. Для этого можно использовать один из обработчиков CommandHandler (подклассов Handler) и зарегистрировать его в приложении:

```
from telegram.ext import CommandHandler
# регистрируем обработчик команды 'start'
start_handler = CommandHandler('start', start)
# добавляем обработчик в приложение
application.add_handler(start_handler)
```

И это все, что на данном этапе нужно. И наконец, строка application.run_polling() запускает бота, пока не поступит нажмете CTRL+C .

Теперь попробуем. В чат со своим ботом введем команду /start - если все прошло правильно, он ответит.

Теперь бот умеет отвечать только на команду /start. Добавим еще один обработчик, который прослушивает обычные сообщения. Для этого используем класс обработчика MessageHandler (подкласс Handler) для отображения всех текстовых сообщений. Остановим запущенного бота (CTRL+C), определим новую функцию echo(), добавим ее в обработчик MessageHandler, а созданный обработчик зарегистрируем в приложении:

```
from telegram import Update
from telegram.ext import filters, MessageHandler, ApplicationBuilder, CommandHandler, ContextTypes

async def start(update, context):
    ...

async def echo(update, context):
    await context.bot.send_message(chat_id=update.effective_chat.id, text=update.message.text)

if __name__ == '__main__':
    ...

    # создаем обработчик текстовых сообщений,
    # которые будут поступать в функцию `echo()`
    echo_handler = MessageHandler(filters.TEXT & (~filters.COMMAND), echo)

    # регистрируем обработчик для `start()`
    application.add_handler(start_handler)
    # регистрируем обработчик для `echo()`
    application.add_handler(echo_handler)
    # запускаем приложение
    application.run_polling()
```

С этого момента бот должен повторять все получаемые им текстовые (некомандные) сообщения.

> *Примечание. Модуль filters содержит ряд так называемых фильтров, которые фильтруют входящие сообщения по тексту, изображениям, обновлениям статуса и многому другому. Любое сообщение, возвращающее True хотя бы для одного из фильтров, переданных в MessageHandler, будет принято. Можно написать свои собственные фильтры, если это необходимо.*

Добавим боту некоторые реальные функции. Реализуем команду /caps, которая будет принимать некоторый текст в качестве аргумента и отвечать на него копией сообщения в верхнем регистре. Чтобы упростить задачу, будем получать аргументы команды в виде списка, разделенного пробелами, которые передаются команде /caps в функцию обратного вызова caps():

```
...

async def caps(update, context):
    text_caps = ' '.join(context.args).upper()
    await context.bot.send_message(chat_id=update.effective_chat.id, text=text_caps)

if __name__ == '__main__':
    ...
    # создаем новый обработчик для функции `caps()`
    caps_handler = CommandHandler('caps', caps)
    # здесь регистрируются созданные обработчики
    application.add_handler(start_handler)
    application.add_handler(echo_handler)
    # вот регистрация для функции `caps()`
    application.add_handler(caps_handler)
    # запускаем приложение
    application.run_polling()
```

Вверх

> *Примечание. Обратите внимание на использование context.args. CallbackContext будет иметь несколько атрибутов, в зависимости от того, какой обработчик используется.*

Режим встроенных запросов.

Еще одна интересная функция Telegram Bot API - режим встроенных запросов. Если необходимо реализовать такую функциональность для своего бота, то сначала необходимо изменить конфигурацию в @BotFather, включив этот режим при помощи команды /setinline. Иногда требуется какое-то время, пока бот не зарегистрируется в качестве встроенного бота на вашем клиенте. Можно ускорить процесс, перезапустив приложение Telegram или иногда просто нужно немного подождать.

Здесь используется ряд новых типов, так что смотрим внимательно (комментировать код не имеет смысла, т.к. просматривается аналогия с предыдущим кодом):

```
from telegram import InlineQueryResultArticle, InputTextMessageContent
from telegram.ext import InlineQueryHandler

...

async def inline_caps(update, context):
    query = update.inline_query.query
    if not query:
        return
    results = []
    results.append(
        InlineQueryResultArticle(
            id=query.upper(),
            title='Caps',
            input_message_content=InputTextMessageContent(query.upper())
        )
    )
    await context.bot.answer_inline_query(update.inline_query.id, results)

if __name__ == '__main__':
    ...

    # создаем обработчик для функции `inline_caps()`
    inline_caps_handler = InlineQueryHandler(inline_caps)
    # регистрируем обработчик
    application.add_handler(inline_caps_handler)

    application.run_polling()
```

Теперь бот может отвечать копией сообщения в верхнем регистре и через режим встроенных запросов.

Некоторые сбитые с толку пользователи могут попытаться отправить созданному боту команды, которые он не понимает, поэтому можно использовать MessageHandler с фильтром filters.COMMAND, чтобы отвечать на все команды, которые не были распознаны предыдущими обработчиками.

```
...

async def unknown(update, context):
    await context.bot.send_message(chat_id=update.effective_chat.id,
                                   text="Sorry, I didn't understand that command.")

if __name__ == '__main__':
    ...

    # создаем обработчик для функции `unknown()`
    unknown_handler = MessageHandler(filters.COMMAND, unknown)
    # регистрируем обработчик
    application.add_handler(unknown_handler)

    application.run_polling()
```

> *Примечание. Этот обработчик должен быть добавлен последним. Если его поставить первым, то он будет срабатывать до того, как обработчик CommandHandlers увидит обновление. После обработки обновления функцией unknown() все дальнейшие обработчики будут игнорируются.*
>
> *Чтобы обойти такое поведение, можно передать в метод dispatcher.add_handler(handler, group), помимо самой функции обработчика аргумент group со значением, отличным от 0. Аргумент group можно воспринимать как число, которое указывает приоритет обновления обработчика. Более низкая группа означает более высокий приоритет. Обновление может обрабатываться*

Вверх

> *(максимум) одним обработчиком в каждой группе.*

Содержание раздела: