
Содержание

Предисловие	1.1
Урок 1. Введение, простой echo-бот	1.2
Урок 2. “Угадай мелодию”. Подготовка	1.3
Урок 3. “Угадай мелодию”. Завершаем бота	1.4
Урок 4. Вебхуки	1.5
Урок 5. Автопостинг в каналы	1.6
Урок 6. Собираем аналитику при помощи Botan	1.7
Урок 7. Встраиваемые боты (Inline)	1.8
Урок 8. Bot API v2: Кнопки и редактирование сообщений	1.9
Урок 9. Bot API v2: Специальные кнопки, опять редактирование сообщений, кэшированный инлайн.	1.10
Дополнительное чтение	1.11
Работа с библиотекой pyTelegramBotAPI	1.11.1
Отправляем сообщения через Telegram Bot API из Powershell	1.11.2
Запускаем несколько ботов на одной машине: CherryPy only	1.11.3
Запускаем несколько ботов на одной машине: nginx + CherryPy	1.11.4
Упаковываем ботов правильно: файлы .ini и .pyz	1.11.5

Пишем бота для Telegram на языке Python

Здравствуйте, этот "учебник" для тех, кто хочет научиться писать ботов для [Telegram](#).

Связаться с автором можно по адресу groosha (собака) protonmail.com.

Все исходные коды к урокам расположены в [этом репозитории](#).

Оглавление

- [Урок 1. Введение, простой echo-бот](#)
- [Урок 2. “Угадай мелодию”. Подготовка](#)
- [Урок 3. “Угадай мелодию”. Завершаем бота](#)
- [Урок 4. Вебхуки](#)
- [Урок 5. Автопостинг в каналы](#)
- [Урок 6. Собираем аналитику при помощи Botan](#)
- [Урок 7. Встраиваемые боты \(Inline\)](#)
- [Урок 8. Bot API v2: Кнопки и редактирование сообщений](#)
- [Урок 9. Bot API v2: Специальные кнопки, опять редактирование сообщений, кэшированный инлайн \(NEW!\)](#)
- [Дополнительное чтение:](#)
 - [Работа с библиотекой pyTelegramBotAPI](#)
 - [Отправляем сообщения через Telegram Bot API из Powershell](#)
 - [Запускаем несколько ботов на одной машине: CherryPy only](#)
 - [Запускаем несколько ботов на одной машине: nginx + CherryPy](#)
 - [Упаковываем ботов правильно: файлы *.ini и *.pyz](#)

Это не окончательный формат "учебника", со временем что-то может быть добавлено, изменено или удалено.

Урок 1. Введение, простой echo-bot.

Введение

Приветствую тебя, читатель! Судя по количеству вопросов, возникших в последнее время в группе "[Боты и каналы Telegram](#)", народ стал активно интересоваться этой темой. Накопилось достаточно много различных вопросов, что, собственно, и стало причиной появления этого "курса".

Сразу оговорюсь: тот бот, который получится в итоге - это лишь прототип, цель всех этих постов - рассказать об основах ботостроения, показать, как можно за короткое время написать простого бота для своих нужд.

Язык программирования будет Python 3, но это не означает, что любители PHP, Ruby и т.д. в пролёте; все основные принципы совпадают. Я не буду особо останавливаться на описании самого языка, желающие могут ознакомиться с документацией по Python [здесь](#).

Важно!!!

Со временем мне, как автору, стало понятно, что материал из первой версии этого урока сильно устарел и вызывает больше вопросов, чем ответов. Так что, отныне в этом уроке сразу используются хэндлеры (декораторы). Что это и зачем, можно прочитать в соответствующем [доп. материале](#)

Подготовка к запуску

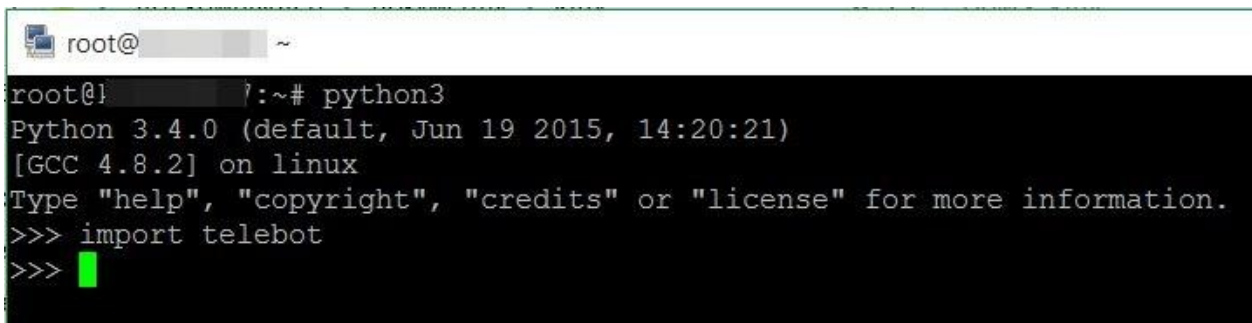
Взаимодействие ботов с людьми основано на HTTP-запросах. С первых дней появления API ботов, я использую библиотеку [pyTelegramBotAPI](#), которая берет на себя все нюансы отправки и получения запросов, позволяя сконцентрироваться непосредственно на логике. Установка библиотеки предельно простая:

```
pip install pytelegrambotapi
```

Для того, чтобы установить менеджер пакетов `pip`, обратитесь к документации по своей операционной системе. Если у Вас установлено несколько интерпретаторов Python, то лучше `pip` заменить на `pip3` для установки правильной версии. Здесь и далее подразумевается использование ОС Ubuntu. Чтобы проверить, правильно ли всё установилось, выполните следующую команду:

```
python3
```

Когда появится окно ввода (вида `>>>`) впишите `import telebot` и нажмите Enter. Если ничего не произошло - значит, библиотека установлена корректно. Как это выглядит, можно увидеть на рисунке 1.



```
root@l ~# python3
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import telebot
>>>
```

Теперь можно выйти из режима Python-консоли (Ctrl+Z или Ctrl+D, или `exit()`)

Пишем простой echo-бот

Ну, довольно слов, перейдем к делу. В качестве практики к первому уроку, напомним бота, повторяющему присланное текстовое сообщение. Создадим каталог, а внутри него создадим 2 файла: `bot.py` и `config.py` . Я рекомендую выносить различные константы и настройки в файл `config.py` , дабы не загромождать другие. В файл `config.py` впишем:

```
# -*- coding: utf-8 -*-
# Этот токен невалидный, можете даже не пробовать :)
token = '110831855:AAE_GbIeVAUwk11012vq4UeMn120iADUtM'
```

token - это то, что вернул Вам [@BotFather](#) при регистрации бота

Теперь откроем файл `bot.py` и создадим объект нашего бота:

```
# -*- coding: utf-8 -*-
import config
import telebot

bot = telebot.TeleBot(config.token)
```

Теперь надо научить бота реагировать на сообщения. Напишем обработчик, который будет реагировать на все текстовые сообщения. (напоминаю: чтобы понять хэндлеры, прочтите этот [доп.материал](#))

```
@bot.message_handler(content_types=["text"])
def repeat_all_messages(message): # Название функции не играет никакой роли, в принципе

    bot.send_message(message.chat.id, message.text)
```

Теперь запустим бесконечный цикл получения новых записей со стороны Telegram:

```
if __name__ == '__main__':
    bot.polling(none_stop=True)
```

Функция `polling` запускает т.н. [Long Polling](#), а параметр `none_stop=True` говорит, что бот должен стараться не прекращать работу при возникновении каких-либо ошибок. При этом, само собой, за ботом нужно следить, ибо сервера Telegram периодически перестают отвечать на запросы или делают это с большой задержкой приводя к [ошибкам 5xx](#)

Итак, полный код файла `bot.py` выглядит следующим образом:

```
# -*- coding: utf-8 -*-
import config
import telebot

bot = telebot.TeleBot(config.token)

@bot.message_handler(content_types=["text"])
def repeat_all_messages(message): # Название функции не играет никакой роли, в принципе

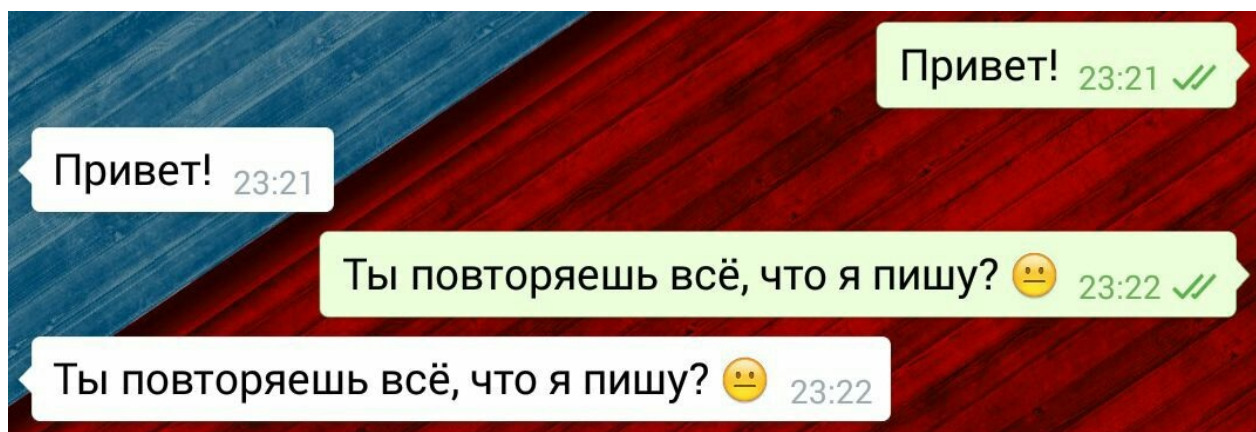
    bot.send_message(message.chat.id, message.text)

if __name__ == '__main__':
    bot.polling(none_stop=True)
```

Готово! Осталось запустить бота:

```
python3 bot.py
```

Теперь мы можем убедиться, что всё работает и бот действительно повторяет наши сообщения.



Сообщение



На этом первый урок окончен.

Урок 2. “Угадай мелодию”. Подготовка.

Нашей целью будет создание простой игры из серии “Угадай мелодию”. В первом уроке мы договорились, что мы делаем прототип, поэтому в том боте, который мы получим, не будет ни таблицы рекордов, ни какой-либо статистики, ни поддержки групповых чатов. Зато мы научимся создавать кастомные клавиатуры, отправлять голосовые заметки и создавать мультишаговые команды.

Целью данного урока будет подготовка базы данных для нашего бота.

Учимся уважать серверы Telegram.

Итак, для начала, подготовим аудиофайлы для отправки. Чтобы не усложнять никому жизнь, будем отправлять аудио как голосовые заметки в формате OGG, а не как музыку. Я взял 5 никому не известных песен, сделал из них 15-20-секундные нарезки, сконвертировал в *.ogg и положил в папку “music”. А теперь делаем финт ушами. Смотрите: мы будем отправлять юзерам одни и те же файлы много-много раз, давайте же побережем свой трафик и дисковое пространство на серверах Telegram, благо в документации написано, что можно отправлять различные объекты не как файлы, а по `file_id` (если файлы уже предварительно загружены). Прекрасно! Попросим нашего бота прислать нам наши аудиофайлы и написать их `file_id`:

```
# -*- coding: utf-8 -*-
import telebot

bot = telebot.TeleBot(config.token)

@bot.message_handler(commands=['test'])
def find_file_ids(message):
    for file in os.listdir('music/'):
        if file.split('.')[1] == 'ogg':
            f = open('music/'+file, 'rb')
            res = bot.send_voice(message.chat.id, f, None)
            print(res)
            time.sleep(3)

if __name__ == '__main__':
    bot.polling(none_stop=True)
```

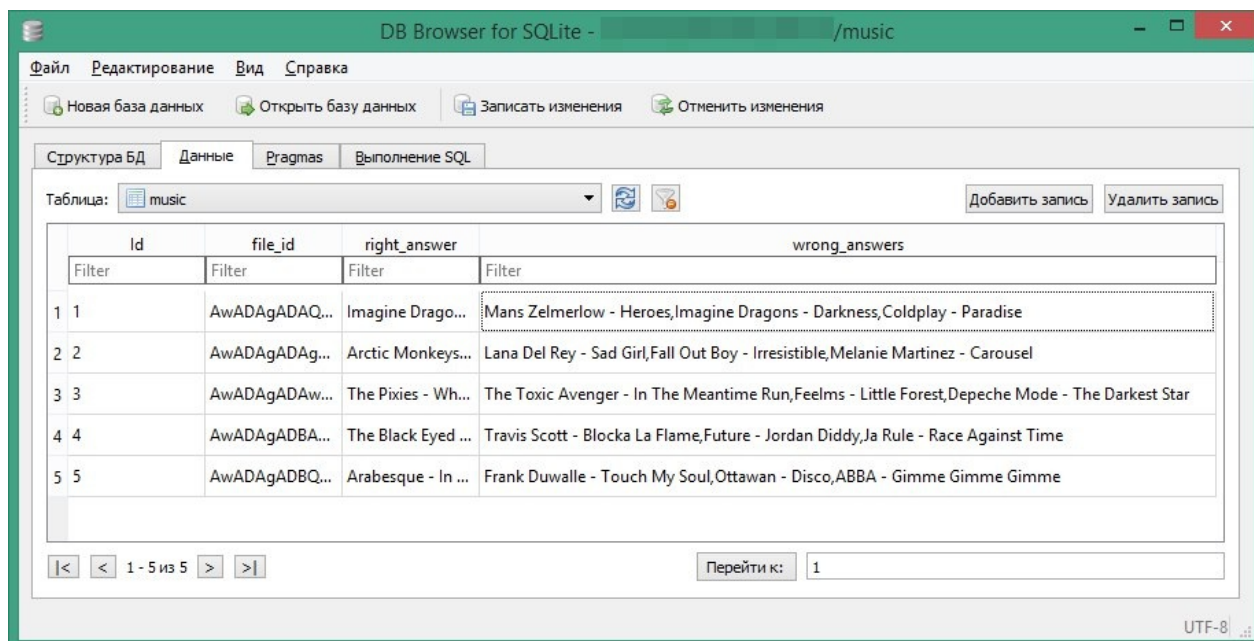
Обратите внимание, в последней строке мы больше не используем бесконечный цикл `While`, из-за изменений в используемой библиотеке. В данном случае по команде `/test` бот будет отправлять наши файлы и выводить в консоль информацию об отправленных медиа, в т.ч. и `file_id`. Записываем их куда-нибудь.

ВАЖНО! Идентификаторы `file_id` уникальны для каждого бота по отдельности! То есть, если Вы хотите, чтобы бот А сохранил `file_id` файла X, то именно этому боту и надо отправлять файлы. Если Вы попытаетесь воспользоваться для этого ботом В, идентификаторы станут невалидными.

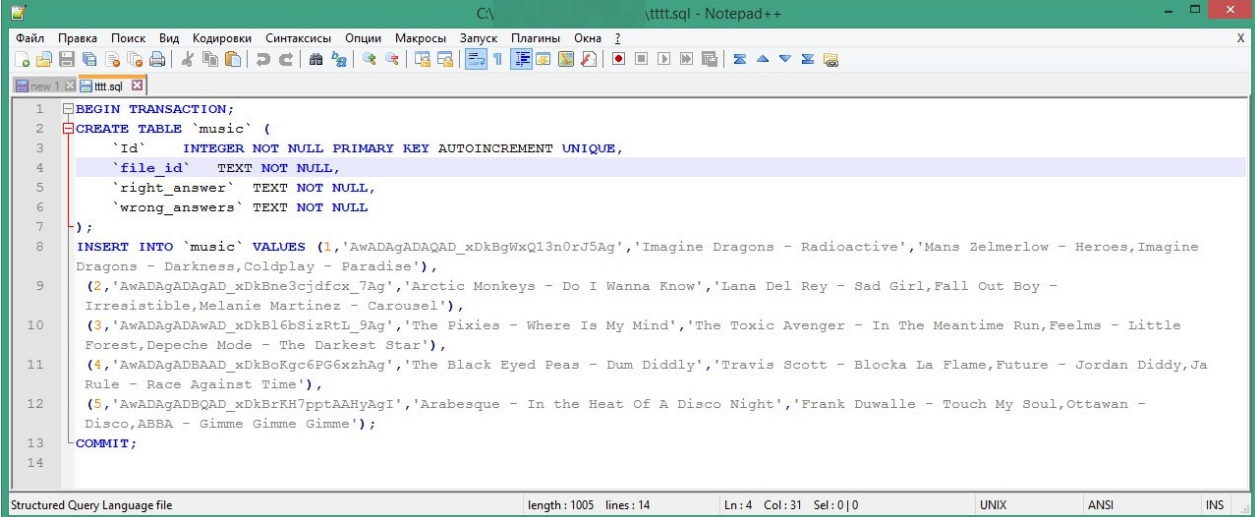
ВАЖНО 2: При отправке медиафайлов большого размера вы можете столкнуться с ошибкой `ConnectionError: ('Connection aborted.', timeout('The write operation timed out',))`. Чтобы её избежать, при вызове методов для медиа `send_audio`, `send_video` и остальных аргумент `timeout=ЧИСЛО`, где значение ЧИСЛО укажите в соответствии с вашими потребностями (например, 5, 10 или что-то ещё, в зависимости от размера файла)

База, приём!

Раз уж мы имеем дело с перманентными данными, нам нужно где-то их хранить. В стандартной библиотеке Python есть 2 чудесных способа: при помощи БД **SQLite3** и при помощи хранилищ типа "ключ-значение" **shelve**. Будем использовать оба варианта. Начнём с БД. Здесь и далее под "БД" или "Базой Данных" я буду понимать именно SQLite3, а под словом "хранилище" - shelve. Итак, при помощи бесплатной Windows-утилиты **DB Browser for SQLite** я создал базу данных с одной-единственной таблицей `music` и заполнил её сведениями о моих аудиофайлах. Чтобы была понятна примерная структура БД, посмотрите на скриншот:



Столбец `file_id` содержит идентификатор аудиозаписи, `right_answer` и `wrong_answer` - правильный и неправильные ответы соответственно. Для чего мне нужно это разделение, объясню позднее. Итак, наша тестовая база создана, при помощи команды **экспорт** я сгенерировал файл с чудесным названием `tttt.sql` следующего содержания:



```
1 BEGIN TRANSACTION;
2 CREATE TABLE `music` (
3   `id` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
4   `file_id` TEXT NOT NULL,
5   `right_answer` TEXT NOT NULL,
6   `wrong_answers` TEXT NOT NULL
7 );
8 INSERT INTO `music` VALUES (1,'AwADAgADAQAD_xDkBgWxQ13nOrU5Ag','Imagine Dragons - Radioactive','Mans Zelmerlow - Heroes,Imagine
9 Dragons - Darkness,Coldplay - Paradise'),
10 (2,'AwADAgADAgAD_xDkBne3cjdfcx_7Ag','Arctic Monkeys - Do I Wanna Know','Lana Del Rey - Sad Girl,Fall Out Boy -
11 Irresistible,Melanie Martinez - Carousel'),
12 (3,'AwADAgADAwAD_xDkBl6bsizRtL_9Ag','The Pixies - Where Is My Mind','The Toxic Avenger - In The Meantime Run,Feelms - Little
13 Forest,Depeche Mode - The Darkest Star'),
14 (4,'AwADAgADBAAD_xDkBoKgc6PG6kzhAg','The Black Eyed Peas - Dum Diddly','Travis Scott - Blocka La Flame,Future - Jordan Diddy,Ja
15 Rule - Race Against Time'),
16 (5,'AwADAgADBQAD_xDkBrKH7pptAAHyAgI','Arabesque - In the Heat Of A Disco Night','Frank Duwalle - Touch My Soul,Ottawan -
17 Disco,ABBA - Gimme Gimme Gimme');
18 COMMIT;
```

Затем я залил этот файл на свой Linux-сервер, в терминале которого выполнил команду `sqlite3 music.db < tttt.sql`, которая привела к созданию файла `music.db`, являющимся базой данных наших аудиозаписей.

Теперь создадим файл `sqlighter.py`. Т.к. Python изначально объектно-ориентированный язык, мне захотелось оформить работу с БД в виде класса. Пусть умные люди меня поправят, если я что-то сделал не так. Вот как выглядит наш класс:

```
# -*- coding: utf-8 -*-
import sqlite3

class SQLighter:

    def __init__(self, database):
        self.connection = sqlite3.connect(database)
        self.cursor = self.connection.cursor()

    def select_all(self):
        """ Получаем все строки """
        with self.connection:
            return self.cursor.execute('SELECT * FROM music').fetchall()

    def select_single(self, rownum):
        """ Получаем одну строку с номером rownum """
        with self.connection:
            return self.cursor.execute('SELECT * FROM music WHERE id = ?', (rownum,)).
fetchall()[0]

    def count_rows(self):
        """ Считаем количество строк """
        with self.connection:
            result = self.cursor.execute('SELECT * FROM music').fetchall()
            return len(result)

    def close(self):
        """ Закрываем текущее соединение с БД """
        self.connection.close()
```

При каждом создании объекта SQLighter будет открываться отдельное соединение с БД и впоследствии закрываться. Мне кажется, это правильный подход, тем более, что бот изначально многопоточный (особенность библиотеки).

Хранилище

Наверняка у кого-то возникнет справедливый вопрос: "А зачем нам нужно простое хранилище, если у нас уже есть полноценная база данных?". Ответ: я просто не хочу лишний раз дёргать БД.

Идея с key-value хранилищем ложится здесь идеально. В чём состоит моя идея: когда юзер начинает игру, вместе с вопросом я сохраняю себе правильный ответ, и при выборе ответа пользователем я сравниваю его ответ с правильным. Совпало - молодец. Не совпало - не молодец. Затем запись удаляется из хранилища, чтобы не занимать лишнее место.

Создадим файл `utils.py`, в котором опишем методы для сохранения правильного ответа, удаления правильного ответа, получения правильного ответа (или `None`, если юзер решил просто так что-то написать боту) и сохранении количества строк в основной БД. Количество строк будет пересчитываться при каждом запуске бота, тем самым, нам не надо думать, по какому правилу выбирать вопросы.

```
# -*- coding: utf-8 -*-
import shelve
from SQLighter import SQLighter
from config import shelve_name, database_name

def count_rows():
    """
    Данный метод считает общее количество строк в базе данных и сохраняет в хранилище.
    Потом из этого количества будем выбирать музыку.
    """
    db = SQLighter(database_name)
    rowsnum = db.count_rows()
    with shelve.open(shelve_name) as storage:
        storage['rows_count'] = rowsnum

def get_rows_count():
    """
    Получает из хранилища количество строк в БД
    :return: (int) Число строк
    """
    with shelve.open(shelve_name) as storage:
        rowsnum = storage['rows_count']
    return rowsnum

def set_user_game(chat_id, estimated_answer):
    """
    Записываем юзера в игроки и запоминаем, что он должен ответить.
    :param chat_id: id юзера
    :param estimated_answer: правильный ответ (из БД)
    """
    with shelve.open(shelve_name) as storage:
        storage[str(chat_id)] = estimated_answer

def finish_user_game(chat_id):
    """
    Заканчиваем игру текущего пользователя и удаляем правильный ответ из хранилища
    :param chat_id: id юзера
    """
    with shelve.open(shelve_name) as storage:
        del storage[str(chat_id)]
```

```
def get_answer_for_user(chat_id):
    """
    Получаем правильный ответ для текущего юзера.
    В случае, если человек просто ввёл какие-то символы, не начав игру, возвращаем None
    """
    :param chat_id: id юзера
    :return: (str) Правильный ответ / None
    """
    with shelve.open(shelve_name) as storage:
        try:
            answer = storage[str(chat_id)]
            return answer
        # Если человек не играет, ничего не возвращаем
        except KeyError:
            return None
```

Не вижу смысла подробно комментировать данный код, поясню лишь использование ключевого слова `with` : оно позволяет не заморачиваться о закрытии хранилища, Python сам возьмет на себя управление. Подробнее можно прочесть в [официальной документации](#).

Ах да, и не забудьте создать файл `config.py` , содержащий следующие строки:

```
# -*- coding: utf-8 -*-
token = 'YOUR_TOKEN' # Токен бота
database_name = 'music.db' # Файл с базой данных
shelve_name = 'shelve.db' # Файл с хранилищем
```

На следующем занятии мы закончим нашего бота-угадайку.

Урок 3. “Угадай мелодию”. Завершаем бота.

Что ж, настало время завершить начатое. В [уроке №2](#) мы подготовили базу данных и хранилище для бота, осталось написать логику для него.

Начнем с того, что добавим в файл `utils.py` функцию, создающую кастомную клавиатуру с вариантами ответа.

```
def generate_markup(right_answer, wrong_answers):
    """
    Создаем кастомную клавиатуру для выбора ответа
    :param right_answer: Правильный ответ
    :param wrong_answers: Набор неправильных ответов
    :return: Объект кастомной клавиатуры
    """

    markup = types.ReplyKeyboardMarkup(one_time_keyboard=True, resize_keyboard=True)
    # Склеиваем правильный ответ с неправильными
    all_answers = '{}{}'.format(right_answer, wrong_answers)
    # Создаем лист (массив) и записываем в него все элементы
    list_items = []
    for item in all_answers.split(','):
        list_items.append(item)
    # Хорошенько перемешаем все элементы
    shuffle(list_items)
    # Заполняем разметку перемешанными элементами
    for item in list_items:
        markup.add(item)
    return markup
```

Возможно, код немного перегружен комментариями, так он совсем небольшой. Сперва создаем объект клавиатуры `ReplyKeyboardMarkup`, о нём можно прочесть в [дополнительной секции](#), затем объединяем в единую строку правильный и неправильный варианты ответа, создаем объект типа `list` (по сути, обычный массив), куда вносим по очереди все варианты ответов, дробя их по запятым.

Наконец, при помощи команды `shuffle` (не забудьте написать сверху `from random import shuffle`) перемешиваем все варианты ответов и заносим в клавиатуру, возвращая её. Это делается для того, чтобы правильный ответ не был всегда первым.

Переходим к основной части. Создаем, если не сделали этого раньше, файл `bot.py`. Объявим нашего бота: `bot = telebot.TeleBot(config.token)`.

Команда будет всего одна: **/game**. По этой команде мы должны обратиться к БД, выдернуть оттуда случайную аудиозапись с вариантами ответа, загнать варианты ответа в клавиатуру, отправить аудиофайл с вариантами ответа и включить "игровой режим".

```
@bot.message_handler(commands=['game'])
def game(message):
    # Подключаемся к БД
    db_worker = SQLighter(config.database_name)
    # Получаем случайную строку из БД
    row = db_worker.select_single(random.randint(1, utils.get_rows_count()))
    # Формируем разметку
    markup = utils.generate_markup(row[2], row[3])
    # Отправляем аудиофайл с вариантами ответа
    bot.send_voice(message.chat.id, row[1], reply_markup=markup)
    # Включаем "игровой режим"
    utils.set_user_game(message.chat.id, row[2])
    # Отсоединяемся от БД
    db_worker.close()
```

Что значит "игровой режим"? Как только юзер нажимает на **/game**, мы заносим его ID в хранилище (именно поэтому его и используем, каждый раз дергать БД некошерно) и предполагаем, что следующее сообщение - ответ на вопрос. Всё, что юзер введёт с клавиатуры или любая кнопка в разметке (суть отправка сообщения) считается ответом. Значит, напомним хэндлер, который реагирует на все текстовые сообщения, кроме **/game** (т.е. следующий код надо расположить в файле `bot.py` ниже предыдущего)

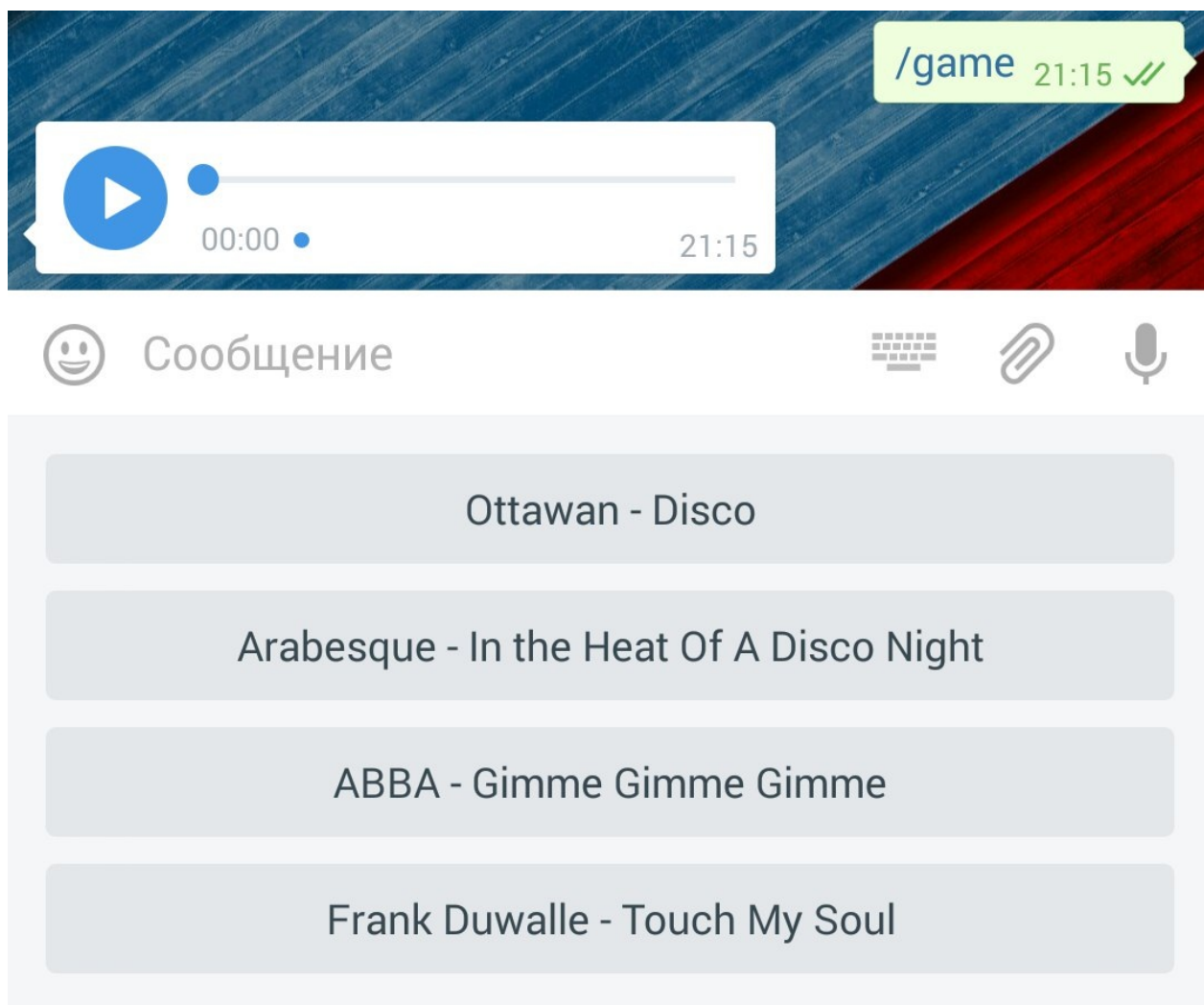
```
@bot.message_handler(func=lambda message: True, content_types=['text'])
def check_answer(message):
    # Если функция возвращает None -> Человек не в игре
    answer = utils.get_answer_for_user(message.chat.id)
    # Как Вы помните, answer может быть либо текст, либо None
    # Если None:
    if not answer:
        bot.send_message(message.chat.id, 'Чтобы начать игру, выберите команду /game')
    else:
        # Уберем клавиатуру с вариантами ответа.
        keyboard_hider = types.ReplyKeyboardRemove()
        # Если ответ правильный/неправильный
        if message.text == answer:
            bot.send_message(message.chat.id, 'Верно!', reply_markup=keyboard_hider)
        else:
            bot.send_message(message.chat.id, 'Увы, Вы не угадали. Попробуйте ещё раз!'
                              , reply_markup=keyboard_hider)
        # Удаляем юзера из хранилища (игра закончена)
        utils.finish_user_game(message.chat.id)
```

Наконец, включаем бота на постоянную проверку входящих сообщений (дописываем следующий код в конец файла):

```
if __name__ == '__main__':
    utils.count_rows()
    random.seed()
    bot.polling(none_stop=True)
```

Собственно, всё! Исходные коды можно посмотреть [вот тут](#). Если будете брать исходники, не забудьте проверить импорты (убрать в некоторых местах lesson_01 или lesson_02, например)

Запустим бота из нужного каталога командой `python3.4 bot.py`, выставив этому файлу права на исполнение владельцем (например, `chmod u+x bot.py`):



Поздравляю, теперь вы умеете писать ботов для Telegram как минимум лёгкой и средней сложности. Наверное, будет ещё пара уроков, посвященная различным плюшкам, типа вебхуков (и трюкам при их использовании), работе с Unix из-под ботов (может быть, напишем свой файловый менеджер), ну и по мелочам.

P.S. Домашнее задание:

1. Наш бот сейчас может некорректно работать в групповых чатах. Что нужно добавить, чтобы он правильно определял игрока и предлагал клавиатуру / проверял ответы только от одного человека?
2. Что нужно сделать, чтобы велась статистика по конкретному человеку?

Урок 4. Вебхуки.

С простым ботом наконец-то разобрались, теперь будем осваивать различные "плюшки". Первая из них и, пожалуй, самая главная, — вебхуки.

А в чём, собственно, разница?

Давайте для начала разберемся, как боты принимают сообщения. Первый и наиболее простой вариант заключается в периодическом опросе серверов Telegram на предмет наличия новой информации. Всё это осуществляется через т.н. [Long Polling](#), т.е. открывается соединение на непродолжительное время и все обновления тут же прилетают боту. Просто, но не очень надежно. Во-первых, серверы Telegram периодически начинают возвращать [ошибку 504](#) (Gateway Timeout), из-за чего некоторые боты впадают в ступор. Даже [pyTelegramBotAPI](#), используемый мной, не всегда может пережить такое.

Во-вторых, если одновременно запущено несколько ботов, вероятность столкнуться с ошибками возрастает. Это вдвойне обидно, если сами боты используются не очень часто.

Вебхуки работают несколько иначе. Устанавливая вебхук, вы как бы говорите серверам Telegram: "Слышь, если кто мне напишет, стукни сюда — (ссылка)". Отпадает необходимость периодически самому опрашивать серверы, тем самым, исчезает неприятная причина падений ботов. Однако за это приходится платить необходимостью установки полноценного веб-сервера на ту машину, на которой планируется запускать ботов. Что ещё неприятно, надо иметь собственный SSL-сертификат, т.к. вебхуки в телеграме работают только по [HTTPS](#). К счастью, в один прекрасный день появилась поддержка [самоподписанных сертификатов](#). Вот об их применении я и расскажу.

Создаем сертификат

Повторяю: я не считаю себя супер-мега-крутым специалистом в айти, возможно, я что-то делаю неправильно, тем не менее, это работает и выглядит вполне прилично.

Ладно, приступим.

Для начала, установим пакет openssl (для Linux): `sudo apt-get install openssl` .

Затем сгенерируем приватный ключ: `openssl genrsa -out webhook_pkey.pem 2048`

Теперь, внимание, генерируем самоподписанный сертификат вот этой вот длинной командой: `openssl req -new -x509 -days 3650 -key webhook_pkey.pem -out webhook_cert.pem`
После этой команды нам предложат ввести некоторую информацию о себе:

двухбуквенный код страны, имя организации и т.д. Если не хотите ничего вводить, ставьте точку. **НО! ВАЖНО!** Когда дойдете до предложения ввести Common Name, следует написать IP адрес сервера, на котором будет запущен бот.

```

root@ /home/ /telegram-lesson-04-webhook# openssl genrsa -out webhook_pkey.pem 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
root@ /home/ /telegram-lesson-04-webhook# openssl req -new -x509 -days 3650 -key webhook_pkey.pem -out w
ebhook_cert.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:RU
State or Province Name (full name) [Some-State]:Russia
Locality Name (eg, city) []:Moscow
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Groosha
Organizational Unit Name (eg, section) []:.
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:.
root@ /home/ /telegram-lesson-04-webhook#
  
```

ВАЖНО!!!
Здесь указываем
IP-адрес
нашего сервера

В результате получим файлы `webhook_cert.pem` и `webhook_pkey.pem`, положим их в какой-нибудь пустой каталог, в котором потом будем создавать бота. Сертификаты готовы, теперь займемся ботом. Чтобы не сильно загружать себе мозги, напишем простой echo-bot из [урока №1](#), только теперь с использованием сертификата.

Наш вишнёвый сервер

Выше я упомянул необходимость наличия веб-сервера, для работы с вебхуками. Те, кто умело владеет Apache или Nginx, можете дальше не читать. Лично я никак не мог (и не могу до сих пор) понять, как обрабатывать входящие сообщения от этих серверов в Python. Поэтому, было принято простое и довольно эффективное решение - используем веб-фреймворк [CherryPy](#). Это не самый простой фреймворк по сравнению, например, с [Flask](#), но мы будем использовать именно его.

Итак, установим CherryPy простой командой `python3.4 -m pip install cherrypy`.

Новый старый бот

Перейдем в каталог с нашими сертификатами и создадим файлы `bot.py` и `config.py`. В последнем создадим переменную `token`, в которую передадим токен нашего бота. Открываем `bot.py`.

Импортируем 2 библиотеки, зададим необходимые константы и создадим экземпляр бота:

```
#!/usr/bin/python3.4
# -*- coding: utf-8 -*-
import telebot
import cherrypy
import config

WEBHOOK_HOST = 'IP-адрес сервера, на котором запущен бот'
WEBHOOK_PORT = 443 # 443, 80, 88 или 8443 (порт должен быть открыт!)
WEBHOOK_LISTEN = '0.0.0.0' # На некоторых серверах придется указывать такой же IP, что
# о и выше

WEBHOOK_SSL_CERT = './webhook_cert.pem' # Путь к сертификату
WEBHOOK_SSL_PRIV = './webhook_pkey.pem' # Путь к приватному ключу

WEBHOOK_URL_BASE = "https://%s:%s" % (WEBHOOK_HOST, WEBHOOK_PORT)
WEBHOOK_URL_PATH = "%s/" % (config.token)

bot = telebot.TeleBot(config.token)
```

Обратите внимание, что Telegram поддерживает всего 4 различных порта при работе с самоподписанными сертификатами. Теоретически, это означает, что на одной машине может быть запущено не больше 4 ботов на вебхуках. Практически, это поправимо, но об этом - в следующий раз.

Создадим класс, реализующий экземпляр веб-сервера. Это, в принципе, стандартный код, который от бота к боту сильно меняться не будет:

```
# Наш вебхук-сервер
class WebhookServer(object):
    @cherrypy.expose
    def index(self):
        if 'content-length' in cherrypy.request.headers and \
            'content-type' in cherrypy.request.headers and \
            cherrypy.request.headers['content-type'] == 'application/json':
            :
            length = int(cherrypy.request.headers['content-length'])
            json_string = cherrypy.request.body.read(length).decode("utf-8")
            update = telebot.types.Update.de_json(json_string)
            # Эта функция обеспечивает проверку входящего сообщения
            bot.process_new_updates([update])
            return ''
        else:
            raise cherrypy.HTTPError(403)
```

Обратите внимание на название функции: `index`. Это, по сути, обозначает последнюю часть url. Поясню на примере: если бы мы хотели получать обновления на адрес `80.100.95.20/webhookbot`, то функцию выше мы бы назвали `webhookbot`. `index` - это аналог отсутствия какой-либо дополнительной маршрутизации. Зачем менять это

значение на другое, рассказано [здесь](#), сейчас это не нужно.

Итак, что мы видим в коде выше? Принимаем входящие запросы по URL

наш. ip. адрес/ , получаем содержимое и прогоняем через набор хэндлеров. Кстати, о них. Т.к. мы реализуем простейших echo-бот, хэндлер нам нужен всего один (такой же, как и в [уроке 1.5](#)):

```
# Хэндлер на все текстовые сообщения
@bot.message_handler(func=lambda message: True, content_types=['text'])
def echo_message(message):
    bot.reply_to(message, message.text)
```

Внимательный читатель всё же заметит одно отличие, о котором я говорить не буду ;) Заодно ещё один повод открыть документацию.

Далее, отправим серверу наш самоподписанный сертификат и "обратный адрес", по которому просим сообщать обо всех новых сообщениях:

```
# Снимаем вебхук перед повторной установкой (избавляет от некоторых проблем)
bot.remove_webhook()

# Ставим заново вебхук
bot.set_webhook(url=WEBHOOK_URL_BASE + WEBHOOK_URL_PATH,
               certificate=open(WEBHOOK_SSL_CERT, 'r'))
```

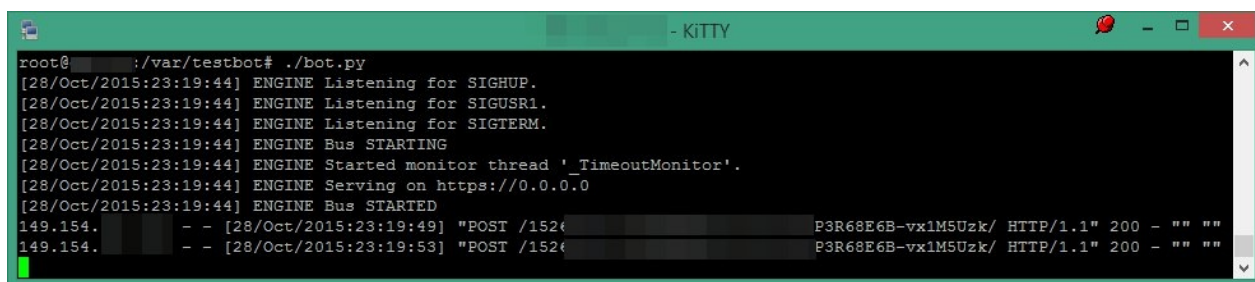
Наконец, укажем настройки нашего сервера и запустим его!

```
# Указываем настройки сервера CherryPy
cherry.py.config.update({
    'server.socket_host': WEBHOOK_LISTEN,
    'server.socket_port': WEBHOOK_PORT,
    'server.ssl_module': 'builtin',
    'server.ssl_certificate': WEBHOOK_SSL_CERT,
    'server.ssl_private_key': WEBHOOK_SSL_PRIV
})

# Собственно, запуск!
cherry.py.quickstart(WebhookServer(), WEBHOOK_URL_PATH, {'/': {}})
```

Обратите внимание на последнюю строку. Наш сервер в качестве "корня" будет прослушивать адрес вида "ip-адрес/токен_бота", относительно которого index - это и есть этот адрес. Может, немного криво пояснил, но позднее вам всё станет предельно ясно, сейчас не нужно загромождать голову лишней информацией.

Запустим бота и напишем ему парочку сообщений. Затем посмотрим в окно терминала:



```
root@:/var/testbot# ./bot.py
[28/Oct/2015:23:19:44] ENGINE Listening for SIGHUP.
[28/Oct/2015:23:19:44] ENGINE Listening for SIGUSR1.
[28/Oct/2015:23:19:44] ENGINE Listening for SIGTERM.
[28/Oct/2015:23:19:44] ENGINE Bus STARTING
[28/Oct/2015:23:19:44] ENGINE Started monitor thread '_TimeoutMonitor'.
[28/Oct/2015:23:19:44] ENGINE Serving on https://0.0.0.0
[28/Oct/2015:23:19:44] ENGINE Bus STARTED
149.154.154.154 - - [28/Oct/2015:23:19:49] "POST /1526 P3R68E6B-vx1M5Uzk/ HTTP/1.1" 200 - "" ""
149.154.154.154 - - [28/Oct/2015:23:19:53] "POST /1526 P3R68E6B-vx1M5Uzk/ HTTP/1.1" 200 - "" ""
```

Если код статуса равен **200 (OK)**, значит, всё в порядке и бот получил сообщения от сервера.

На сегодня всё.

Урок 5. Автопостинг в каналы.

Для начала — небольшое лирическое отступление.

Когда в Telegram появились [каналы](#), поначалу я к ним относился как к неудачной попытке клонировать Twitter. Сразу появились тупые каналчики с тупыми картиночками, что только усугубляло положение дел. Однако сейчас, спустя больше месяца, моё мнение изменилось на диаметрально противоположное. Призванные заменить списки рассылки (ими вообще кто-то пользовался?), каналы дали возможность получать огромное количество контента, которым можно делиться, который можно сохранять.

В [Bot FAQ](#) есть интересная фраза: "Мы будем внимательно смотреть на то, как люди пользуются ботами и развивать их в том направлении". С учетом того, что многие боты занимались именно рассылкой информации, каналы - это очень логичный и правильный шаг в развитии автоматизированных средств.

С точки зрения программиста, каналы решают сразу несколько ключевых проблем:

1. Не надо зависеть от серверов Telegram, т.к. нет входящих сообщений.
2. Анонимная рассылка сообщений (некоторые люди просили сделать возможность отправлять свои сообщения через "безликого" бота) максимально упростилась.
3. Не надо вести списки пользователей, которым нужно отправить информацию и внедрять паузы между отправкой сообщения по очереди всем, эту заботу берёт на себя сам Telegram.

Лично я заметил, что некоторые паблики и группы ВКонтакте стали дублировать свои записи в каналы Telegram. Сегодня мы научимся делать то же самое без помощи каких-либо конструкторов или сторонних веб-сайтов. Чтобы не сильно заморачиваться, будем постить только ссылку на пост, превью к ней и так даст необходимый минимум информации о посте.

Получаем записи

В качестве "подопытного кролика" я выбрал свою маленькую, но очень гордую группу с музыкой [C:\Music](#). Получать новые записи будем при помощи VK API, при этом нам не нужно заморачиваться с созданием приложений, т.к. получить записи со стены можно даже без авторизации ВКонтакте (если в настройках сообщества не указано иначе). Итак, сформируем ссылку, которая будет нам возвращать последние 10 записей от имени сообщества из группы C:\Music: [https://api.vk.com/method/wall.get?](https://api.vk.com/method/wall.get?domain=c.music&count=10&filter=owner)

[domain=c.music&count=10&filter=owner](https://api.vk.com/method/wall.get?domain=c.music&count=10&filter=owner)

Что здесь что? `domain` - короткое имя сообщества. Если его нет, то меняем

`domain=xxx` на `owner_id=-yyy` (обратите внимание на минус перед числом, это важно). `count` - число выводимых записей. Чем дольше пауза между проверками и чем чаще в сообществе появляются записи, тем большее число нужно выставить, но не более 100. `filter=owner` просит сервер выводить записи только от имени группы, полезно, если стена открыта.

Давайте теперь создадим файл `bot.py`, в котором зададим основные константы и импорты:

```
# -*- coding: utf-8 -*-

import time
import eventlet
import requests
import logging
import telebot
from time import sleep

# Каждый раз получаем по 10 последних записей со стены
URL_VK = 'https://api.vk.com/method/wall.get?domain=c.music&count=10&filter=owner'
FILENAME_VK = 'last_known_id.txt'
BASE_POST_URL = 'https://vk.com/wall-39270586_'

BOT_TOKEN = 'токен бота, постящего в канал'
CHANNEL_NAME = '@канал'

bot = telebot.TeleBot(BOT_TOKEN)
```

Во-первых, не забудьте сделать нужного бота администратором канала, иначе фокус не удастся. Во-вторых, обратите внимание, что в импортах появилась библиотека `eventlet`, она поможет нам избежать проблем при получении записей из ВК. В-третьих, в указанный `txt`-файл будем записывать номер верхнего поста на момент проверки, я решил не заморачиваться с созданием `key-value` хранилищ, ради одного числа-то. В-четвёртых, в качестве параметра `BASE_POST_URL` указываем ссылку на любой пост из нашей группы без последнего числа.

Иногда ВК начинает дурить и не возвращает список постов за приемлемое время. В этом случае, нам нужно отвалиться по таймауту и пропустить проверку. Можно, конечно, попробовать ещё раз, но мы люди не настойчивые :)

```
def get_data():
    timeout = eventlet.Timeout(10)
    try:
        feed = requests.get(URL_VK)
        return feed.json()
    except eventlet.timeout.Timeout:
        logging.warning('Got Timeout while retrieving VK JSON data. Cancelling...')
        return None
    finally:
        timeout.cancel()
```

Суть простая: получилось - возвращаем объект с постами. Не получилось - возвращаем None. Теперь перейдем непосредственно к парсингу. Алгоритм будет такой:

1. Открываем файл, хранящий последний известный номер верхнего поста.
2. Если метод `get_data()` вернул объект с записями, начинаем проходить по нему со второго элемента, т.к. первый - какое-то неизвестное мне рандомное число.
3. Если номер поста меньше или равен последнему известному - завершаем обход.
4. Проверяем наличие закрепленного поста. Если таковой есть, то передаем функции отправки сообщений все записи, кроме закрепленной. Иначе - просто передаем все.
5. У каждой проверяемой записи забираем ID, подставляем рядом с `BASE_POST_URL` и получаем полный ID записи.
6. Отправляем его в канал.
7. Как только обход завершился, берем номер первой (второй, если первая - закрепленная) записи и записываем в файл поверх старого значения.
8. Засыпаем или завершаем.

По поводу п.8: дополнительно предусмотрим в нашей программе два режима: в первом режиме скрипт постоянно работает, засыпая после каждой итерации на 4 минуты; во втором режиме скрипт просто завершает работу, что позволяет ставить его в [планировщик cron](#). В определении режима нам поможет константная переменная `SINGLE_RUN`, которую надо не забыть указать где-нибудь вверху.

```
def send_new_posts(items, last_id):
    for item in items:
        if item['id'] <= last_id:
            break
        link = '{!s}{!s}'.format(BASE_POST_URL, item['id'])
        bot.send_message(CHANNEL_NAME, link)
        # Спим секунду, чтобы избежать разного рода ошибок и ограничений (на всякий сл
        учай!)
        time.sleep(1)
    return
```



```

def check_new_posts_vk():
    # Пишем текущее время начала
    logging.info('[VK] Started scanning for new posts')
    with open(FILENAME_VK, 'rt') as file:
        last_id = int(file.read())
        if last_id is None:
            logging.error('Could not read from storage. Skipped iteration.')
            return
        logging.info('Last ID (VK) = {}'.format(last_id))
    try:
        feed = get_data()
        # Если ранее случился таймаут, пропускаем итерацию. Если всё нормально - парсим
        # посты.
        if feed is not None:
            entries = feed['response'][1:]
            try:
                # Если пост был закреплен, пропускаем его
                tmp = entries[0]['is_pinned']
                # И запускаем отправку сообщений
                send_new_posts(entries[1:], last_id)
            except KeyError:
                send_new_posts(entries, last_id)
            # Записываем новый last_id в файл.
            with open(FILENAME_VK, 'wt') as file:
                try:
                    tmp = entries[0]['is_pinned']
                    # Если первый пост - закрепленный, то сохраняем ID второго
                    file.write(str(entries[1]['id']))
                    logging.info('New last_id (VK) is {}'.format((entries[1]['id'])))
                )
            except KeyError:
                file.write(str(entries[0]['id']))
                logging.info('New last_id (VK) is {}'.format((entries[0]['id'])))
            )
        except Exception as ex:
            logging.error('Exception of type {} in check_new_post(): {}'.format(type(ex).__name__, str(ex)))
            pass
        logging.info('[VK] Finished scanning')
        return

```

Осталось дело за малым - написать логику запуска всего процесса и инициализировать логгер, который будет писать в текстовый файл обо всех событиях в жизни бота:

```
if __name__ == '__main__':
    # Избавляемся от спама в логах от библиотеки requests
    logging.getLogger('requests').setLevel(logging.CRITICAL)
    # Настраиваем наш логгер
    logging.basicConfig(format='[%asctime)s] %(filename)s:%(lineno)d %(levelname)s -
%(message)s', level=logging.INFO,
                        filename='bot_log.log', datefmt='%d.%m.%Y %H:%M:%S')
    if not SINGLE_RUN:
        while True:
            check_new_posts_vk()
            # Пауза в 4 минуты перед повторной проверкой
            logging.info('[App] Script went to sleep.')
            time.sleep(60 * 4)
    else:
        check_new_posts_vk()
    logging.info('[App] Script exited.\n')
```

Перед запуском бота, создадим вручную файл `last_known_id.txt` и впишем в него один из последних числовых ID, в моём случае это было чудесное число 1893. После включения бота, в зависимости от значения `SINGLE_RUN`, он будет либо постоянно работать, проверяя каждые 4 минуты на наличие новых постов, либо завершится после окончания первой проверки. Для себя я выбрал второй вариант, добавив скрипт в `cron`.

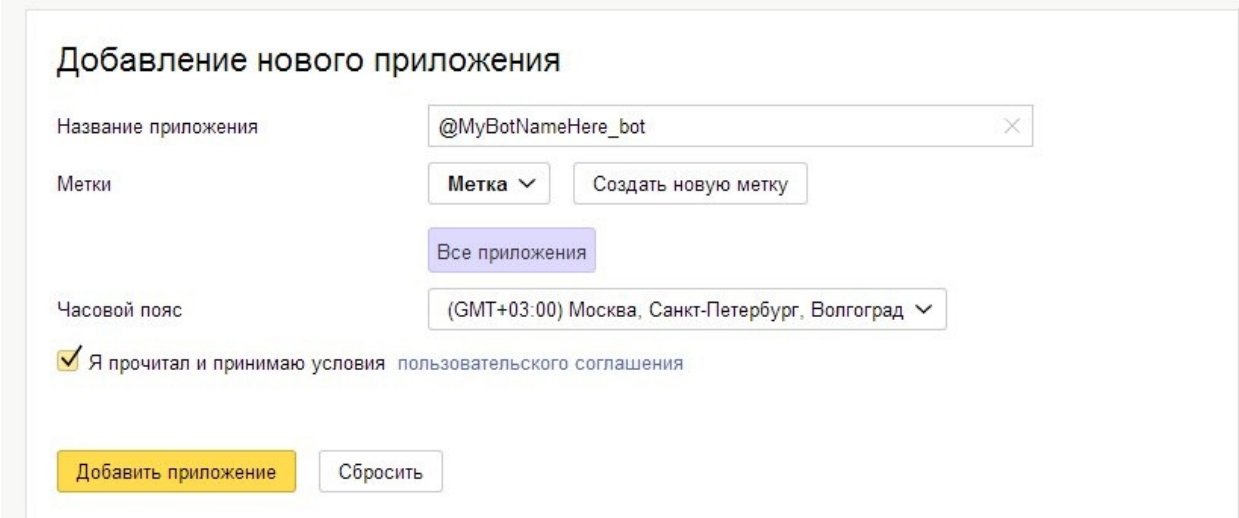
На сегодня всё, теперь вы можете легко и просто настроить автопостинг записей практически из любой группы ВК в свой канал.

Урок 6. Собираем аналитику при помощи Botan.

Вы наверняка уже видели [новость](#) о такой чудесной системе аналитики для ботов Telegram, как [Botan](#). Сам я ей, по правде говоря, пользуюсь уже несколько месяцев, поэтому могу с уверенностью сказать, что штука эта очень полезная и качественная. Сегодня мы научимся пользоваться Ботаном в наших Telegram-ботах. В данном уроке будет рассмотрено только подключение бота к Botan через веб-интерфейс, всё, что касается их бота, вы можете рассмотреть самостоятельно.

Регистрируемся в Yandex AppMetrica

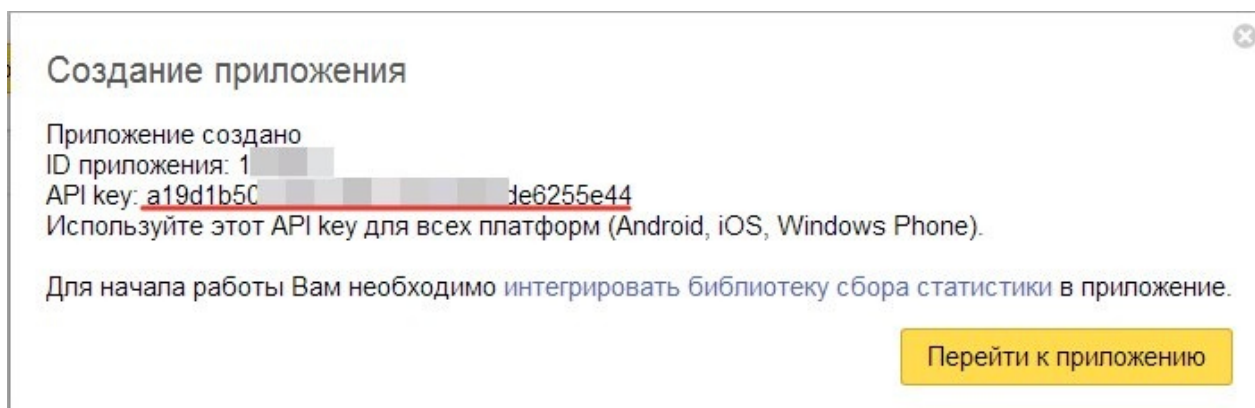
Первое, что необходимо сделать, это перейти по адресу <https://appmetrica.yandex.com> и войти при помощи своей Яндекс-почты. После того, как вы попадете в панель управления, нажимайте на "+ Добавить приложение". В появившемся окне нужно ввести ник бота (вместе с "собакой"), прочитать условия пользовательского соглашения и согласиться с ними, поставив галку напротив соответствующего пункта.



The screenshot shows the 'Добавление нового приложения' (Add new application) form in Yandex AppMetrica. The form includes the following fields and controls:

- Название приложения** (Application name): A text input field containing '@MyBotNameHere_bot'.
- Метки** (Tags): A section with a 'Метка' (Tag) dropdown menu, a 'Создать новую метку' (Create new tag) button, and a 'Все приложения' (All applications) button.
- Часовой пояс** (Time zone): A dropdown menu showing '(GMT+03:00) Москва, Санкт-Петербург, Волгоград'.
- Agreement:** A checked checkbox followed by the text 'Я прочитал и принимаю условия пользовательского соглашения' (I have read and accept the terms of the user agreement).
- Buttons:** A yellow 'Добавить приложение' (Add application) button and a grey 'Сбросить' (Reset) button.

После нажатия на "Добавить приложение", появится окошко, в котором нас интересует поле API Key. Скопируйте его куда-нибудь в надёжное место, скоро этот ключ нам понадобится.



Интегрируем аналитику в нашего бота

Теперь осталось начать сбор статистики по нашему боту! В качестве примера создадим ещё одного, в котором будет 2 команды: `/random` и `/yesorno`. Первая будет выводить случайное число от 1 до 10, а вторая - "да" или "нет".

Итак, для начала займемся файлом `config.py`. Помимо токена самого бота, создадим переменную `botan_key`, в которую запишем полученный ранее API Key.

Хорошо, с конфигом закончим. Создадим файл `botan.py`. Кто-то скажет, мол, можно же взять готовый из [соответствующего репозитория](#). Так-то да, но при работе с `pyTelegramBotAPI`, мне пришлось внести в него некоторые дополнения. Сам файл довольно большой, чтобы приводить его полностью здесь, поэтому отсылаю вас к [своему репозиторию](#), где его можно просмотреть.

Функция `make_json` получает на вход объект `Message` и выбирает из него необходимые поля. В принципе, можно собирать статистику по чему угодно, т.е. можно смотреть, пользователь с каким ID или ником сколько раз вызвал ту или иную команду, в каком чате чаще используют бота и т.д.

В данном примере я оставлю сбор различных параметров, но в коде будут указания, что делать, если нужно просто количество использований той или иной команды.

Создадим две функции:

```

@bot.message_handler(commands=['random'])
def cmd_random(message):
    bot.send_message(message.chat.id, random.randint(1, 10))
    # Если не нужно собирать ничего, кроме количества использований, замените третий а
    ргумент message на None
    botan.track(config.botan_key, message.chat.id, message, 'Случайное число')
    return

@bot.message_handler(commands=['yesorno'])
def cmd_yesorno(message):
    bot.send_message(message.chat.id, random.choice(strings))
    # Если не нужно собирать ничего, кроме количества использований, замените третий а
    ргумент message на None
    botan.track(config.botan_key, message.chat.id, message, 'Да или нет')
    return

```

Не забудьте импортировать стандартный модуль `random` и написать после него

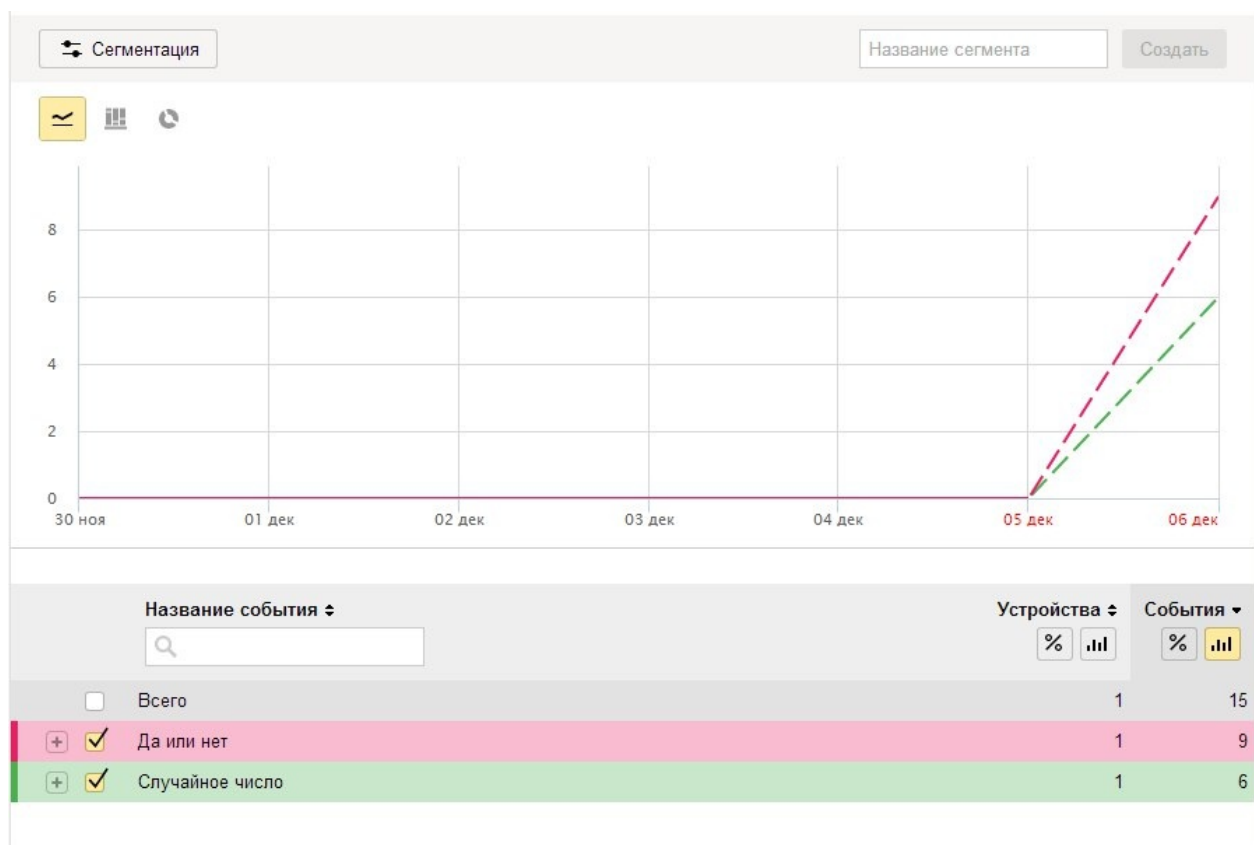
`random.seed()` для инициализации генератора!

Также обратите внимание на строчку `botan.track(...)`. Именно эта строка и отвечает за отправку статистики. Первый аргумент - ключ, полученный при регистрации бота в Ботане, второй аргумент - `chat.id`, получаемый в сообщении. Третий аргумент - объект сообщения, из которого вытягивается необходимая информация, а последний - что-то типа метки, описывающая команду.

В первой строке функции `cmd_yesorno` есть вызов `random.choise(strings)`, это выбор случайного элемента из массива, в нашем случае содержащего всего 2 элемента: строку "да" и строку "нет".

Собственно, всё. Дописываем нужные строки кода, как и раньше и запускаем бота!

Несколько раз используем команды `/random` и `/yesorno`, подождем пару минут и заглянем в статистику, выбрав бота в Метрике, пункт "Приложение" - "События":



Как видите, всё прекрасно работает!

Урок 7. Встраиваемые боты (Inline)

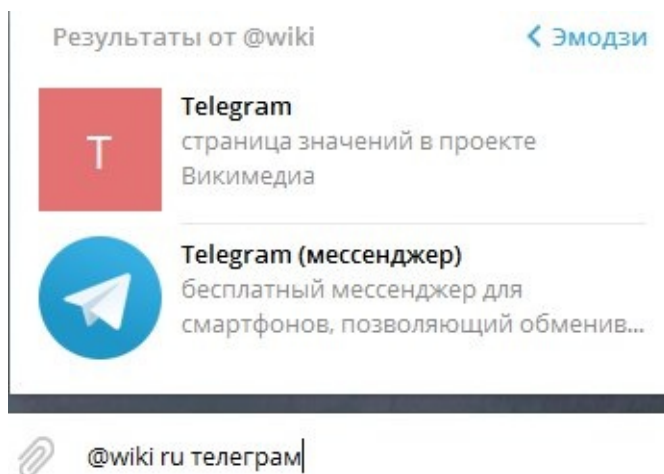
Важно: код, приведённый здесь, актуален для первой версии Bot API.

Пожалуйста, откройте [урок 8](#), чтобы узнать о важных изменениях.

4 января 2016 года разработчики Telegram выпустили большое дополнение к существующему BotAPI, заключающееся в появлении **встраиваемых ботов**.

К сожалению, не все до конца понимают, в чем их особенность. А вот в чем: если раньше для того, чтобы получить какую-либо информацию от бота и перекинуть её собеседнику нужно было открывать диалог с ботом, писать всякие команды, а потом пересылать ответ в нужный чат, то теперь всё стало быстрее и проще (для некоторых ситуаций): просто открываете нужный чат, вызываете бота, введя его ник в поле ввода сообщения, ставите пробел и пишете свой запрос. Бот отвечает на эти запросы в виде всплывающих подсказок, число и содержание которых зависит от того, что вы написали боту и от заложенного в него алгоритма. Если одна из подсказок удовлетворяет вашему запросу, нажимаете на неё и некоторое сообщение отправляется в тот чат, в котором вы находитесь.

Например, если вы хотите отправить своему собеседнику ссылку на статью о Telegram из русскоязычной Википедии, то достаточно ввести в поле ввода `@wiki ru Telegram` и подождать пару секунд. Результат представлен на рисунке:



Соответственно, если вы нажмете на одну из этих подсказок, то ссылка на соответствующую статью будет отправлена в текущий чат. Собственно, всё!

Помимо ссылок, при помощи Inline API можно отправлять фотографии, GIF-анимации, видеоанимации в формате Mpeg-4 и ссылки на видеозаписи, которые можно будет смотреть прямо в приложении.

Прежде, чем мы начнём писать своего встраиваемого бота, несколько слов о том, для чего НЕ надо их использовать и какие есть ограничения. Во-первых, такие боты нужны именно для подсказок, когда вы хотите чем-то поделиться, но вам нужна помощь бота. Вы же не помните все ссылки на те самые видео с котиками? А бот быстренько поможет его найти, чтобы вы поделились им в чатике. Во-вторых, существуют недокументированные ограничения по объемам отправляемой информации. Скажем, для текстов и ссылок нельзя отправлять элементы длиннее 300-400 символов (на самом деле, получается без проблем около двухсот, больше - с оговорками). На мой взгляд, это самый неприятный момент в API. Я давеча пытался прикрутить поиск цитат с сайта [Bash.im](https://bash.im) из своей базы для одного из моих [ботов](#). Увы, всё, чего удалось добиться, это отправка коротких цитат (примерно с твит размером) непосредственно текстом, длинных - через отставку ссылок на сам Баш.

Ну а теперь приступим к созданию нашего первого встраиваемого бота. Вообще говоря, никто не мешает добавить существующему Inline-возможности, но дабы не усложнять код, напомним отдельного.

Что же он будет уметь? Пусть на вход боту будут подаваться 2 числа, а он в подсказках будет предлагать основные математические действия: сложение, вычитание, умножение и деление. Да, пример надуманный, но как нельзя лучше показывает особенности Inline API.

В начале, зарегистрируем бота у [@BotFather](#) (или воспользуемся имеющимся) для него вызовем команду `/setinline`.

Внимание: после ввода этой команды и выбора бота нужно ввести подсказку, появляющуюся в поле ввода при вызове нашего бота в дальнейшем, иначе фокус не удастся.

Создадим файл исходного кода и добавим туда необходимые импорты и объект самого бота:

```
# -*- coding: utf-8 -*-

import telebot
import re
from telebot import types

token = 'ваш_токен'
bot = telebot.TeleBot(token)
```

Мы условились, что на вход будут подаваться 2 числа, поэтому нам нужно построить регулярное выражение, которое будет реагировать на корректный ввод. Оно очень простое: `digits_pattern = re.compile(r'^[0-9]+ [0-9]+$', re.MULTILINE)`.

А теперь самое главное - логика. Пока юзер пишет, Телеграм никак себя не проявляет, но как только он делает паузу, приложение считает, что это законченный ввод и отправляет его боту. Юзер дописал ещё пару символов и остановился? Окей, снова кидаем это боту. Соответственно, может сложиться ситуация, когда пользователь ввёл одно число и задумался. Наш парсер, попробовав сопоставить текст на наличие регулярного выражения, ругнётся и сотворит всякие нехорошие вещи, например, остановит весь скрипт (мы же помним, что Python - интерпретируемый язык, да?). Значит, надо ставить ловлю исключений:

```
try:
    matches = re.match(digits_pattern, query.query)
except AttributeError as ex:
    return
```

Если условие выше отработало без ошибок, значит, мы получили два искоемых числа. Теперь начинаются особенности Inline API. Каждая подсказка, присылаемая ботом, это объект с набором некоторых характеристик. Мы будем использовать объект типа "Статья".

Согласно [документации](#), обязательных параметров у такого объекта четыре: тип, уникальный идентификатор, заголовок и текст, который будет отправлен. По поводу типа можно не беспокоиться: используемая мной уже больше полугода библиотека [pyTelegramBotAPI](#) подставит нужное значение сама. Уникальный идентификатор придется ставить самим и тут есть подвох: при использовании подгрузки (об этом в конце урока) идентификаторы должны быть уникальными у всех значений в одном большом запросе (т.е. у исходных результатов и всех подгружаемых в пределах одного запроса). Учтите это. Заголовок - очевидно. Отправляемый текст - то, что будет отправлено в текущий чат при нажатии на данную подсказку.

Есть ещё необязательный параметр "описание" (description). Так вот, **описание != отправляемый текст**. В подсказке может быть написано "Наполеон - это торт", а в отправляемом тексте могут содержаться коды запуска ядерных ракет. Шутка, но смысл понятен.

Вооружившись этими знаниями и имея на руках два присланных пользователем числа, давайте создадим 4 объекта, каждый из которых отвечает за свою математическую операцию (на самом деле 5, т.к. учтем деление на ноль):

```

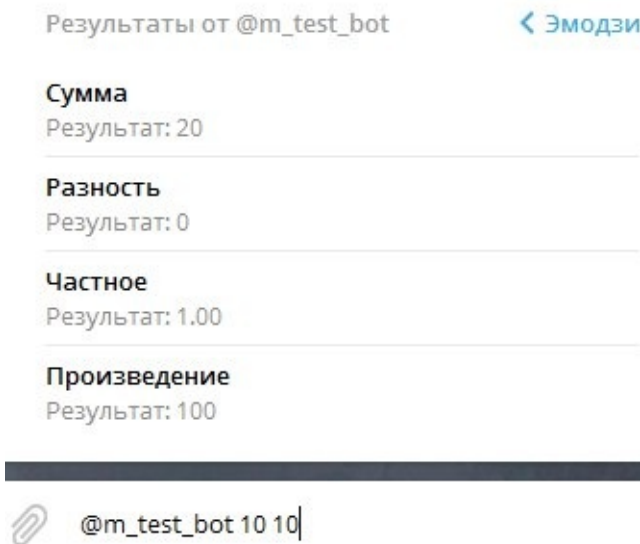
num1, num2 = matches.group().split()
try:
    m_sum = int(num1) + int(num2)
    r_sum = types.InlineQueryResultArticle(
        id='1', title="Сумма",
        # Описание отображается в подсказке,
        # message_text - то, что будет отправлено в виде сообщения
        description="Результат: {!s}".format(m_sum),
        input_message_content=types.InputTextMessageContent(
            message_text="{!s} + {!s} = {!s}".format(num1, num2, m_sum))
    )
    m_sub = int(num1) - int(num2)
    r_sub = types.InlineQueryResultArticle(
        id='2', title="Разность",
        description="Результат: {!s}".format(m_sub),
        input_message_content=types.InputTextMessageContent(
            message_text="{!s} - {!s} = {!s}".format(num1, num2, m_sub))
    )
    # Учтем деление на ноль и подготовим 2 варианта развития событий
    if num2 is not "0":
        m_div = int(num1) / int(num2)
        r_div = types.InlineQueryResultArticle(
            id='3', title="Частное",
            description="Результат: {0:.2f}".format(m_div),
            input_message_content=types.InputTextMessageContent(
                message_text="{0!s} / {1!s} = {2:.2f}".format(num1, num2, m_div))
        )
    else:
        r_div = types.InlineQueryResultArticle(
            id='3', title="Частное", description="На ноль делить нельзя!",
            input_message_content=types.InputTextMessageContent(
                message_text="Я нехороший человек и делю на ноль!")
        )
    m_mul = int(num1) * int(num2)
    r_mul = types.InlineQueryResultArticle(
        id='4', title="Произведение",
        description="Результат: {!s}".format(m_mul),
        input_message_content=types.InputTextMessageContent(
            message_text="{!s} * {!s} = {!s}".format(num1, num2, m_mul))
    )
    bot.answer_inline_query(query.id, [r_sum, r_sub, r_div, r_mul])
except Exception as e:
    print("{!s}\n{!s}".format(type(e), str(e)))

```

Полагаю, тут ничего сверхсложного нет. Вызывает интерес только последняя строка, а именно метод `answer_inline_query`. У него обязательных параметров два: уникальный идентификатор запроса (его получает хэндлер, так что про него забываем) и массив ответов. Загоняем теперь предыдущие два куска кода в одну функцию, обернутую хэндлером:

```
@bot.inline_handler(func=lambda query: len(query.query) > 0)
def query_text(query):
    (тут два куска кода, приведённых выше)
```

И можно запускать! Введём в любом чате, кроме секретного, ник нашего бота и два числа. Смотрим на подсказку:



В принципе, правильно, но как-то не наглядно. Да и не очень понятно для нового пользователя, как, что и зачем.

А давайте учтем ситуацию, когда пользователь ввёл только ник бота, и покажем ему подсказку:

```
@bot.inline_handler(func=lambda query: len(query.query) is 0)
def empty_query(query):
    hint = "Введите ровно 2 числа и получите результат!"
    try:
        r = types.InlineQueryResultArticle(
            id='1',
            parse_mode='Markdown',
            title="Бот \"Математика\"",
            description=hint,
            input_message_content=types.InputTextMessageContent(
                message_text="Эх, зря я не ввёл 2 числа :(")
        )
        bot.answer_inline_query(query.id, [r])
    except Exception as e:
        print(e)
```

Уже лучше, но всё равно не очень наглядно. А давайте добавим превью для каждой подсказки, чтобы слева была иконка соответствующей операции, и, заодно, сделаем превью при делении на ноль кликабельным, отправляя пользователя на [эту страницу](#)

Википедии.

Во-первых, закачаем на какой-нибудь бесплатный хостинг иконки с нашими математическими операциями (сгенерировал в [Metro Studio](#), размер каждой 48x48 px):

```
plus_icon = "https://pp.vk.me/c627626/v627626512/2a627/7dlh4RRhd24.jpg"
minus_icon = "https://pp.vk.me/c627626/v627626512/2a635/ILYe7N2n8Zo.jpg"
divide_icon = "https://pp.vk.me/c627626/v627626512/2a620/oAvUk7Awp0.jpg"
multiply_icon = "https://pp.vk.me/c627626/v627626512/2a62e/xqnPMigaP5c.jpg"
error_icon = "https://pp.vk.me/c627626/v627626512/2a67a/ZvTeGq6Mf88.jpg"
```

Во-вторых, отредактируем функцию `query_text`, добавив к каждому объекту типа "Статья" иконку, а для деления на ноль - свою иконку и ссылку:

```
@bot.inline_handler(func=lambda query: len(query.query) > 0)
def query_text(query):
    try:
        matches = re.match(digits_pattern, query.query)
        # Вылавливаем ошибку, если вдруг юзер ввёл чушь
        # или задумался после ввода первого числа
    except AttributeError as ex:
        return

    # В этом месте мы уже уверены, что всё хорошо,
    # поэтому достаём числа
    num1, num2 = matches.group().split()
    try:
        m_sum = int(num1) + int(num2)
        r_sum = types.InlineQueryResultArticle(
            id='1', title="Сумма",
            # Описание отображается в подсказке,
            # message_text - то, что будет отправлено в виде сообщения
            description="Результат: {!s}".format(m_sum),
            input_message_content=types.InputTextMessageContent(
                message_text=" {!s} + {!s} = {!s}".format(num1, num2, m_sum)),
            # Указываем ссылку на превью и его размеры
            thumb_url=plus_icon, thumb_width=48, thumb_height=48
        )
        m_sub = int(num1) - int(num2)
        r_sub = types.InlineQueryResultArticle(
            id='2', title="Разность",
            description="Результат: {!s}".format(m_sub),
            input_message_content=types.InputTextMessageContent(
                message_text=" {!s} - {!s} = {!s}".format(num1, num2, m_sub)),
            thumb_url=minus_icon, thumb_width=48, thumb_height=48
        )
        # Учтем деление на ноль и подготовим 2 варианта развития событий
    if num2 is not "0":
        m_div = int(num1) / int(num2)
        r_div = types.InlineQueryResultArticle(
            id='3', title="Частное",
            description="Результат: {:.2f}".format(m_div),
```

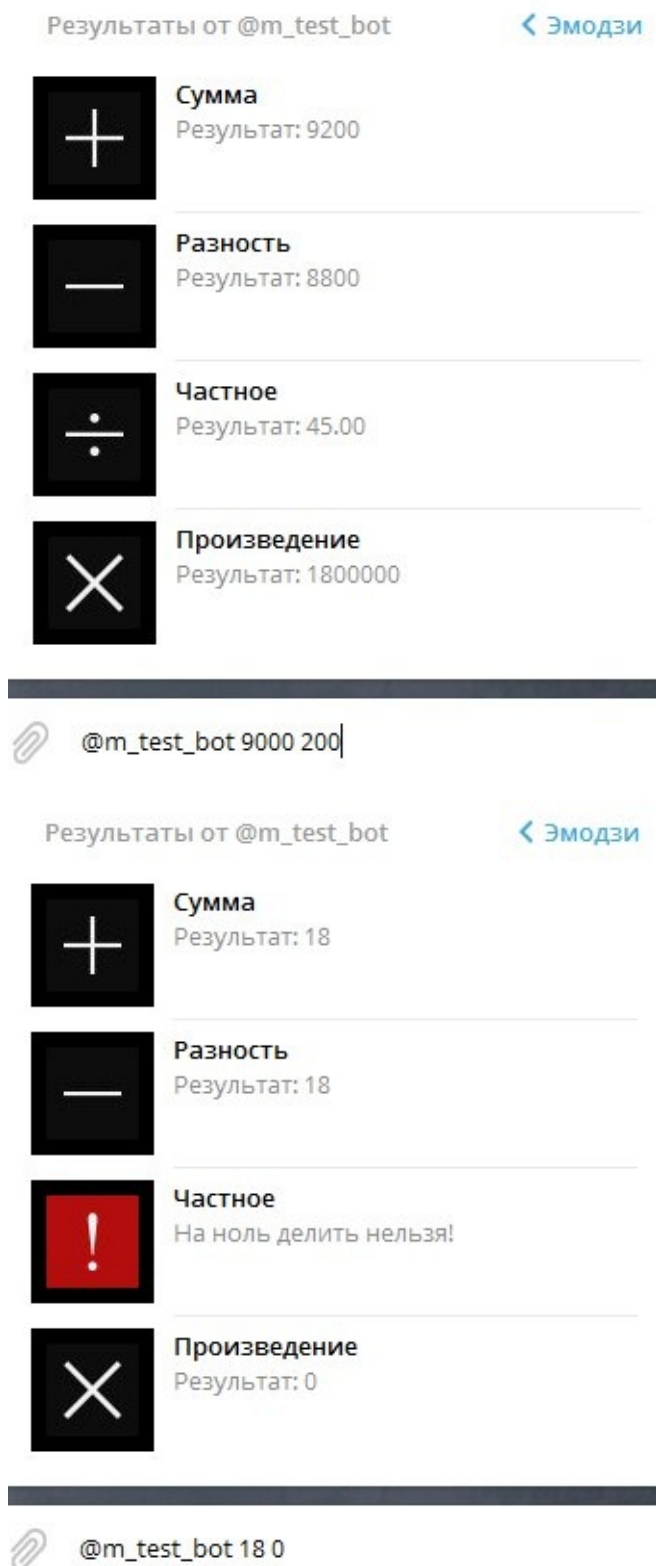
```

        input_message_content=types.InputTextMessageContent(
            message_text="{0!s} / {1!s} = {2:.2f}".format(num1, num2, m_div)),
            thumb_url=divide_icon, thumb_width=48, thumb_height=48
        )
    else:
        r_div = types.InlineQueryResultArticle(
            id='3', title="Частное", description="На ноль делить нельзя!",
            input_message_content=types.InputTextMessageContent(
                message_text="Я нехороший человек и делю на ноль!"),
            thumb_url=error_icon, thumb_width=48, thumb_height=48,
            # Сделаем превью кликабельным, по нажатию юзера направят по ссылке
            url="https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%BD%
D0%B8%D0%B5_%D0%BD%D0%B0_%D0%BD%D0%BE%D0%BB%D1%8C",
            disable_web_page_preview=True,
            # Не будем показывать URL в подсказке
            hide_url=True
        )
        m_mul = int(num1) * int(num2)
        r_mul = types.InlineQueryResultArticle(
            id='4', title="Произведение",
            description="Результат: {!s}".format(m_mul),
            input_message_content=types.InputTextMessageContent(
                message_text="{!s} * {!s} = {!s}".format(num1, num2, m_mul)),
            thumb_url=multiply_icon, thumb_width=48, thumb_height=48
        )
    # В нашем случае, результаты вычислений не изменятся даже через долгие годы, Н
    0!
    # если где-то допущена ошибка и cache_time уже выставлен большим, то это уже н
    икак не исправить (наверное)
    # Для справки: 2147483646 секунд - это 68 с копейками лет :)
    bot.answer_inline_query(query.id, [r_sum, r_sub, r_div, r_mul], cache_time=214
7483646)
except Exception as e:
    print("{!s}\n{!s}".format(type(e), str(e)))

```

Обратите снова внимание на функцию `answer_inline_query`, в ней я добавил аргумент `cache_time`. Значит он именно то, что первым приходит в голову — устанавливает время, на которое результат будет кэширован, чтобы лишний раз не дергать бота. Сейчас на моей стороне тот факт, что математика - фундаментальная наука, и за те 68 с копейками лет, что указаны у меня в аргументе `cache_time`, результат операций не изменится. Но! Выше на рис. 2 у меня слева от подсказок нет ничего, никаких картинок. И да, в течение 68 с лишним лет конкретно для данного запроса так и будет. Поэтому трижды подумайте, прежде чем установите большое время кэширования. Для каких-то динамических результатов, например, для подсказки при нулевом запросе, вполне достаточно времени `86400` (1 сутки). По умолчанию, кстати, вообще всего 300 секунд, т.е. 5 минут.

Теперь наши результаты выглядят куда симпатичнее:



Прекрасно! Мы написали своего первого встраиваемого бота! Но...

Но есть ещё такая штука, как `offset`. Вообще говоря, это нужно для постепенной подгрузки новых результатов. Например, пользователь запрашивает видео с котиками, бот возвращает пять штук. Как только юзер долистывает до конца, бот запрашивает у своего источника ещё пять, а Telegram дописывает их к уже имеющимся. В итоге, уже 10 вариантов и так далее. В одном из своих [ботов](#) я сделал так: запрашиваю из БД 5

записей с `offset` равным нулю. При отправке их пользователю, я устанавливаю в функции `answer_inline_query` параметр `next_offset` в значение `5` (строковый параметр, если что). Если юзер долистал до конца, то боту придет запрос с этим же текстом, но с значением `offset` как раз `5`, поэтому в следующий раз я из БД запрошу уже следующие пять записей, прибавлю ещё пятерку к текущему значению, получая `10` и снова верну пользователю значения. Как только у меня закончились соответствующие запросу данные в базе, отправляю пустую строку в качестве аргумента `next_offset`. В общем, всё выглядит примерно так (бОльшая часть кода вырезана для упрощения понимания):

```

def query_text(query):
    offset = int(query.offset) if query.offset else 0
    with sqlite3.connect('quotes.db') as connection:
        cursor = connection.cursor()
        # Тут используется SQLite Full-Text Search, не пугайтесь
        quotes = cursor.execute("SELECT num, text FROM quotes WHERE quotes.id IN (SELECT rowid FROM fts WHERE text MATCH ?) LIMIT 5 OFFSET ?", (query.query, offset,)).fetchall()
    if len(quotes) is 0:
        try:
            result = types.InlineQueryResultArticle(id='1',
                                                    title=(заголовок),
                                                    description=(описание),
                                                    input_message_content=types.InputTextMessageContent(
                                                        message_text="(текст)"))
            bot.answer_inline_query(query.id, [result])
        except Exception as e:
            print(e)
        return
    results_array = []
    try:
        m_next_offset = str(offset + 5) if len(quotes) == 5 else None
        for index, quote in enumerate(quotes):
            try:
                # При использовании подгрузки, ID должны быть уникальными в пределах всей большой пачки!
                results_array.append(types.InlineQueryResultArticle(id=str(offset + index),
                                                                    title=(заголовок),
                                                                    description=(описание),
                                                                    input_message_content=types.InputTextMessageContent(
                                                                        message_text=(текст)),
                                                                    )
                )
            except Exception as e:
                print(e)
        # устанавливаем новый offset или сбрасываем, если в БД закончились релевантные записи
        bot.answer_inline_query(query.id, results_array, next_offset=m_next_offset if m_next_offset else "")
    except Exception as e:
        print(e)

```

На этом урок действительно закончен.

Урок 8. Bot API v2: Кнопки и редактирование сообщений

В начале апреля 2016 года [вышло](#) первое по-настоящему крупное обновление API для ботов. Изменений довольно много, поэтому материал я разобью на несколько частей. Сегодня поговорим об inline-кнопках и редактировании сообщений, а затем обсудим новые инлайн-режимы вместе со специальными кнопками для отправки геолокации и номера телефона.

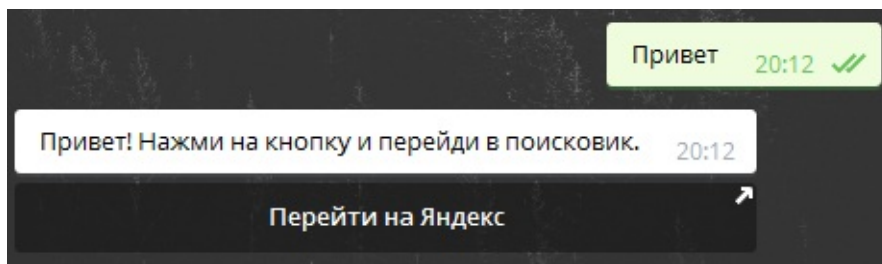
Начнём с двух важных изменений:

- 1) Каждая кнопка, будь то обычная или инлайн, это теперь самостоятельный объект `KeyboardButton` или `InlineKeyboardButton`, не забудьте обновить своих ботов!
- 2) В Inline-режиме все текстовые поля теперь представлены отдельными объектами `InputMessageContent`, которые, в свою очередь могут быть аж 4-х типов (подробности [тут](#)).

Итак, **инлайн-кнопки**. Что это такое? Это специальные объекты, которые "цепляются" к конкретным сообщениям и распространяют своё действие, в общем случае, только на них. Делятся такие кнопки на три типа: URL-кнопки, Callback-кнопки и Switch-кнопки. Самыми простыми являются кнопки-ссылки (URL). Как видно из названия, их цель - просто перекидывать пользователей по определенным веб-адресам. Давайте сразу напишем обработчик, который будет на любое сообщение отвечать каким-либо текстом и предложением перейти, например, на Яндекс.

```
@bot.message_handler(content_types=["text"])
def default_test(message):
    keyboard = types.InlineKeyboardMarkup()
    url_button = types.InlineKeyboardButton(text="Перейти на Яндекс", url="https://ya.ru")
    keyboard.add(url_button)
    bot.send_message(message.chat.id, "Привет! Нажми на кнопку и перейди в поисковик.",
    , reply_markup=keyboard)
```

Инлайн-клавиатура представляет собой объект `InlineKeyboardMarkup`, а каждая инлайн-кнопка — это объект `InlineKeyboardButton`. Чтобы получилась URL-кнопка, нужно указать значения параметров `text` (текст на кнопке) и `url` (валидный веб-адрес). В результате бот пришлет нам такое сообщение (см. рис.). В целях обеспечения безопасности, перед переходом по URL-кнопкам появляется всплывающее окно, в котором видна ссылка целиком.



Прежде, чем мы перейдем к другим кнопкам, давайте познакомимся с функциями редактирования сообщений, коих тоже три: `editMessageText` (редактирование текста), `editMessageCaption` (редактирование подписи к медиа) и `editMessageReplyMarkup` (редактирование инлайн-клавиатуры). В рамках этого урока рассмотрим только первую функцию, остальные работают аналогично и предлагаются для самостоятельного изучения.

Чтобы отредактировать сообщение, нам надо знать, про какое именно идёт речь. В случае, если оно было отправлено самим ботом, идентификаторами служит связка `chat_id + message_id`. Но если сообщение было отправлено в инлайн-режиме, то ориентироваться надо по параметру `inline_message_id`.

И вот теперь вернемся к нашим баранам кнопкам. На очереди – **Callback**. Это, на мой взгляд, самая крутая фишка нового обновления. Колбэк-кнопки позволяют выполнять произвольные действия по их нажатию. Всё зависит от того, какие параметры каждая кнопка в себе несёт. Соответственно, все нажатия будут приводить к отправке боту объекта `CallbackQuery`, содержащему поле `data`, в котором написана некоторая строка, заложенная в кнопку, а также либо объект `Message`, если сообщение отправлено ботом в обычном режиме, либо поле `inline_message_id`, если сообщение отправлено в инлайн-режиме.

Приведу пример, после которого все вопросы должны отпасть: пусть, например, если сообщение отправлено ботом в обычном режиме, то нажатие на кнопку заменит текст сообщения на "Пыщь", если в инлайн – то "Бдыщь". При этом в обоих случаях значение `callback_data` будет равно `test`. Что для этого нужно сделать: во-первых, написать простейший хэндлер для всех входящих сообщений, во-вторых, написать простейший хэндлер для инлайн-сообщений, в-третьих, написать простейший хэндлер для колбэка, который определит, из какого режима пришло сообщение.

```

# Обычный режим
@bot.message_handler(content_types=["text"])
def any_msg(message):
    keyboard = types.InlineKeyboardMarkup()
    callback_button = types.InlineKeyboardButton(text="Нажми меня", callback_data="test")
    keyboard.add(callback_button)
    bot.send_message(message.chat.id, "Я - сообщение из обычного режима", reply_markup=keyboard)

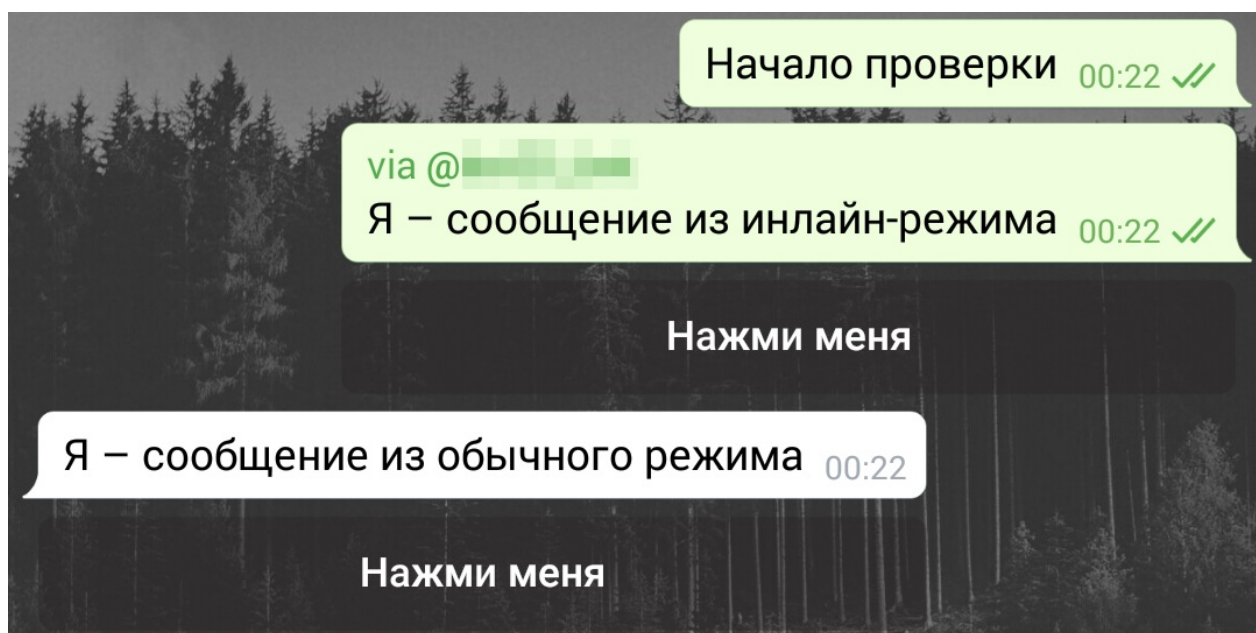
# Инлайн-режим с непустым запросом
@bot.inline_handler(lambda query: len(query.query) > 0)
def query_text(query):
    kb = types.InlineKeyboardMarkup()
    # Добавляем колбэк-кнопку с содержимым "test"
    kb.add(types.InlineKeyboardButton(text="Нажми меня", callback_data="test"))
    results = []
    single_msg = types.InlineQueryResultArticle(
        id="1", title="Press me",
        input_message_content=types.InputTextMessageContent(message_text="Я - сообщение из инлайн-режима"),
        reply_markup=kb
    )
    results.append(single_msg)
    bot.answer_inline_query(query.id, results)

# В большинстве случаев целесообразно разбить этот хэндлер на несколько маленьких
@bot.callback_query_handler(func=lambda call: True)
def callback_inline(call):
    # Если сообщение из чата с ботом
    if call.message:
        if call.data == "test":
            bot.edit_message_text(chat_id=call.message.chat.id, message_id=call.message.message_id, text="Пыщь")
    # Если сообщение из инлайн-режима
    elif call.inline_message_id:
        if call.data == "test":
            bot.edit_message_text(inline_message_id=call.inline_message_id, text="Бдыщь")

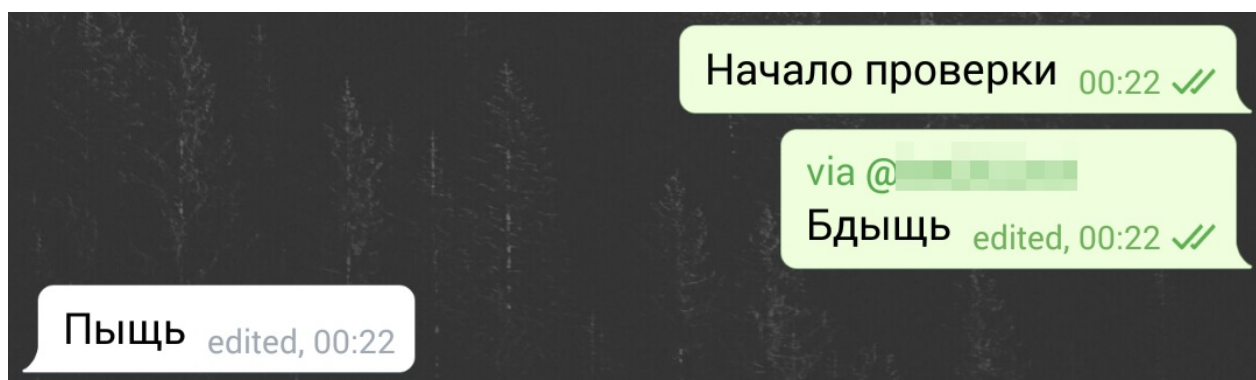
if __name__ == '__main__':
    bot.polling(none_stop=True)

```

Запускаем бота, отправляем инлайн-сообщение, которое, в свою очередь, вызовет обычное:



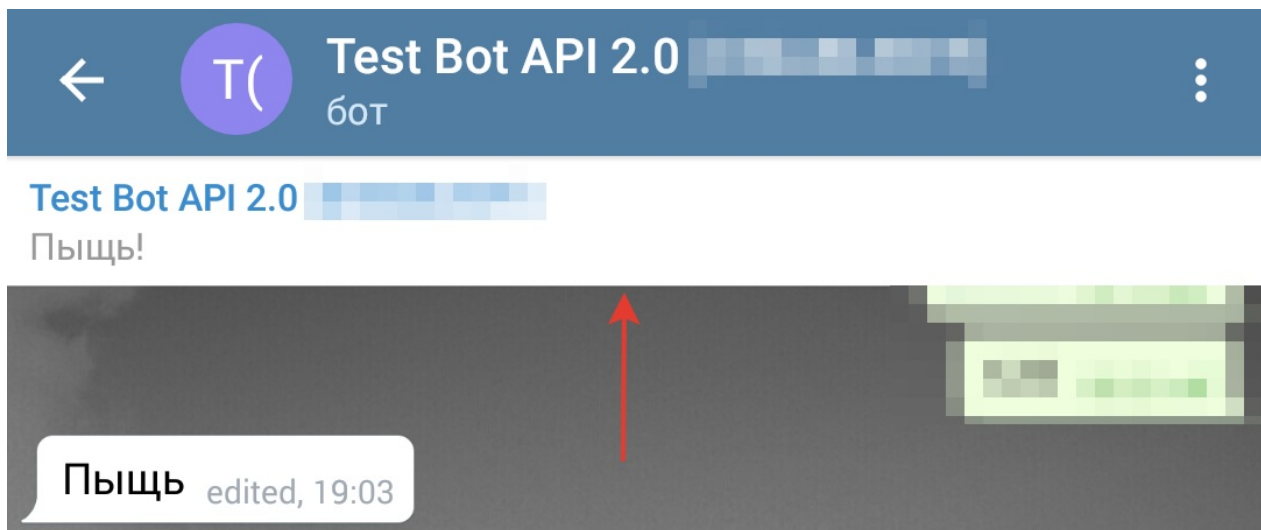
Нажмем на обе кнопки, результат правильный:



Таким образом, callback-кнопки – это очень мощный инструмент для взаимодействия пользователей с ботом, а редактирование сообщений дополнительно помогает в этом. Более того, нажатие на колбэк-кнопку может дополнительно триггернуть либо уведомление в верхней части экрана, либо всплывающее окно. Покажу первый вариант. Пускай помимо изменения сообщения на "Пыщь", аналогичное слово показывается уведомлением. Для этого перепишем первое `if`-условие в хендлере колбэков:

```
if call.message:
    if call.data == "test":
        bot.edit_message_text(chat_id=call.message.chat.id, message_id=call.message.message_id, text="Пыщь")
        bot.answer_callback_query(callback_query_id=call.id, show_alert=False, text="Пыщь!")
```

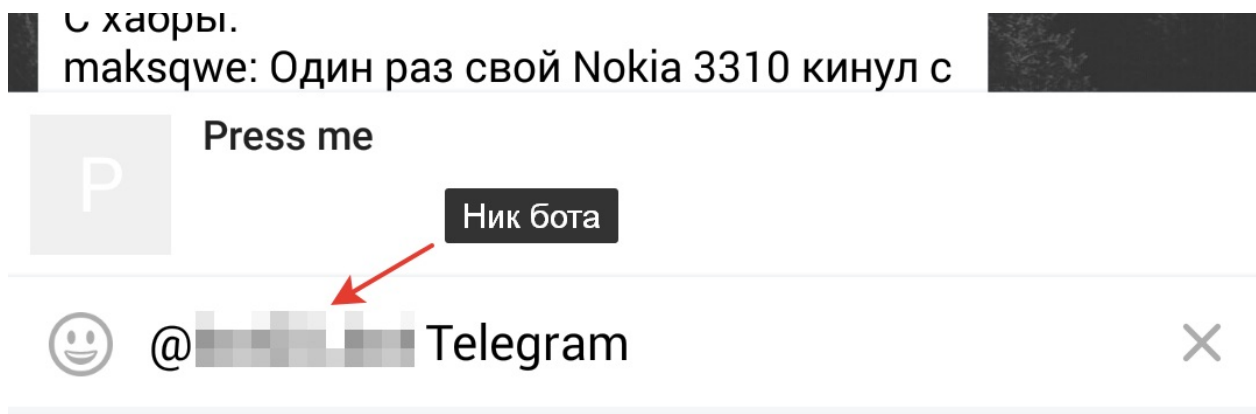
Результат – на скриншоте. Попробуйте, кстати, изменить аргумент `show_alert` на `True` и посмотрите, что получится.



Наконец, остался последний тип кнопок - **Switch** (переключатель). Они нужны, чаще всего, для обучения пользователей работе с ботом в инлайн-режиме. Чтобы активировать сделать кнопку такого типа, нужно указать аргумент `switch_inline_query` либо пустой, либо с каким-либо текстом. В последнем случае этот текст будет сразу подставлен в поле ввода, например, для показа демонстрации инлайна. Как вообще работает такая кнопка? При нажатии на неё Telegram предложит выбрать чат, после чего подставит в поле ввода ник вашего бота и (если есть), текст, указанный вами в аргументе `switch_inline_query`. Давайте попробуем так сделать. Добавим кнопку, которая будет перенаправлять пользователя в какой-либо чат и предлагать в инлайн-режиме запрос "Telegram". Код всего хендлера выглядит вот так:

```
@bot.message_handler(content_types=["text"])
def any_msg(message):
    keyboard = types.InlineKeyboardMarkup()
    switch_button = types.InlineKeyboardButton(text="Нажми меня", switch_inline_query=
"Telegram")
    keyboard.add(switch_button)
    bot.send_message(message.chat.id, "Я - сообщение из обычного режима", reply_markup
=keyboard)
```

Теперь, если мы нажмем на кнопку и выберем чат, вот что получится:



Итак, в этом уроке мы познакомились с новыми кнопками в Telegram Bot API, научились ~~переписывать историю~~ редактировать сообщения и отправлять небольшие уведомления по нажатию. В следующий раз продолжим изучать новые возможности для ботов.

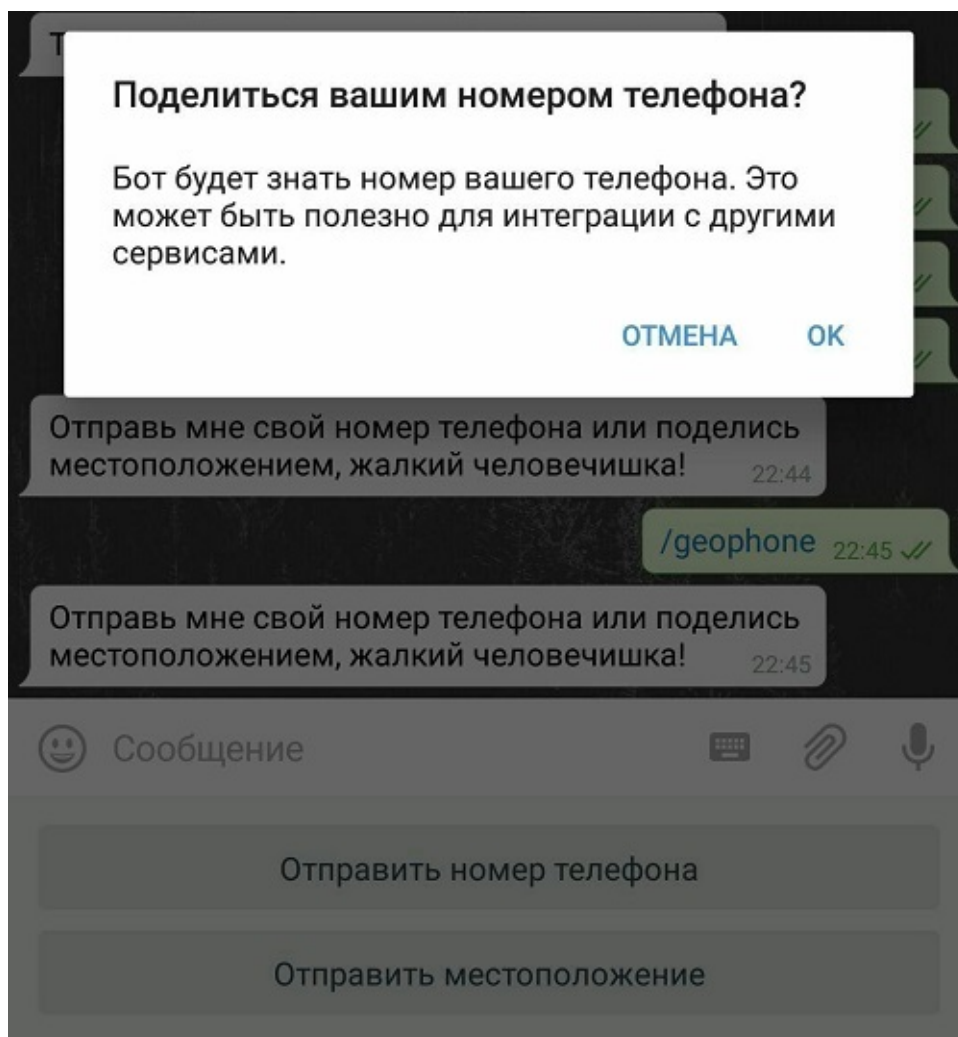
А исходники к этому уроку можно найти в [этом репозитории](#)

Урок 9. Bot API v2: Специальные кнопки, опять редактирование сообщений, кэшированный инлайн.

Продолжаю рассказывать о нововведениях в Bot API версии 2. Я не буду рассказывать о методах `getChat`, `getChatMember` и т.д., которые появились в обновлении 2.1: они интуитивно понятны и особых проблем не вызывают. Вопросы могут возникнуть при изучении специальных обычных кнопок, вроде тех, что запрашивают у вас номер телефона и геолокацию, при попытке получить отредактированное сообщение, а также при работе с уже загруженными в облако объектами с инлайн-режимом. Обо всём по порядку.

Специальные кнопки

Некоторым ботам жизненно необходим ваш номер телефона или местоположение, например, для привязки к учётным записям на других сайтах или же поиска близлежащих объектов на карте. Разработчики Telegram прислушались к мнению ботоводов и добавили особые свойства обычным (не инлайновым) кнопкам. Итак, чтобы запросить номер телефона, нужно помимо аргумента `text` передать аргумент `request_contact=True`, а для геолокации, соответственно, `request_location=True`. Обратите внимание, что одновременно у кнопки может быть не больше одного особого свойства (можно не указывать никакой), а также что специальные кнопки могут быть отправлены только в диалоги (бот-человек). Напишем код, который на команду `/geophone` отправит нам клавиатуру с этими кнопками.



```
# не забудьте про from telebot import types
@bot.message_handler(commands=["geophone"])
def geophone(message):
    # Эти параметры для клавиатуры необязательны, просто для удобства
    keyboard = types.ReplyKeyboardMarkup(row_width=1, resize_keyboard=True)
    button_phone = types.KeyboardButton(text="Отправить номер телефона", request_contact=True)
    button_geo = types.KeyboardButton(text="Отправить местоположение", request_location=True)
    keyboard.add(button_phone, button_geo)
    bot.send_message(message.chat.id, "Отправь мне свой номер телефона или поделись местоположением, жалкий человечиска!", reply_markup=keyboard)
```

При нажатии на кнопку отправки номера телефона сервер вернёт объект Message с непустым типом [Contact](#), а при нажатии на кнопку отправки геолокации – с непустым типом [Location](#).

Важно: если вы используете [конечные автоматы](#) или любой другой механизм состояний, который будет ждать от пользователя его телефон в объекте Contact, помните, что ушлый юзер может попробовать обмануть бота и скинуть любой другой

контакт из записной книжки. Чтобы убедиться, что номер телефона принадлежит именно этому конкретному пользователю, сравните `user_id` в объекте `from` с `user_id` в объекте `Contact`, они должны совпадать.

Редактирование сообщений пользователями

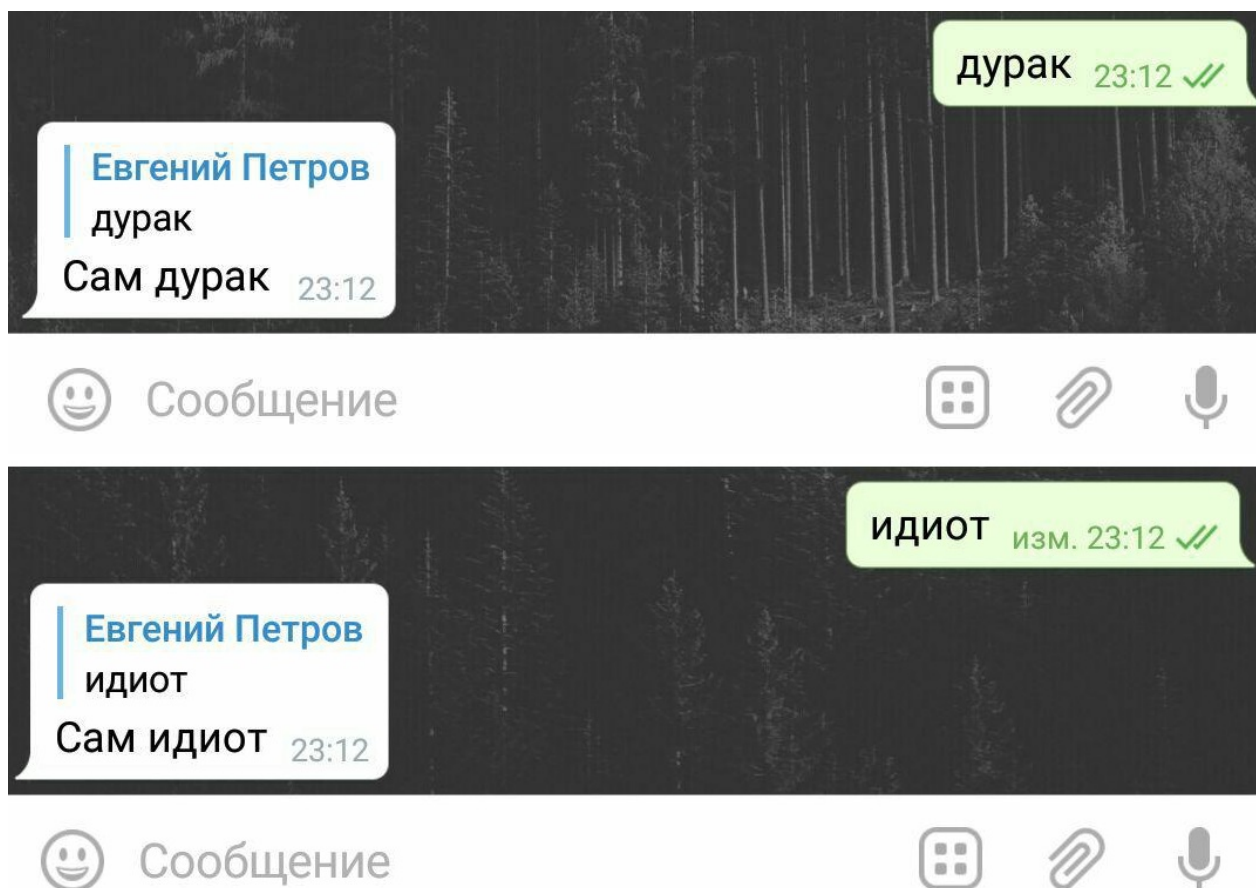
Начиная с [мая 2016 года](#), пользователи могут редактировать свои сообщения, а боты могут видеть исправления. Как им в этом помочь, давайте разберёмся вместе. В качестве примера заставим нашего бота отвечать на ругательства. К примеру, если пользователь пишет "дурак", бот ответит "сам дурак". Хитрые люди могут попробовать отредактировать своё сообщение и выставить бота в дурном свете, но мы будем изменять ответ бота под пользовательский текст.

Для отслеживания изменений, у нас в копилке появился новый тип хэндлеров – `edited_message_handler`, который настраивается точно так же, как и `message_handler`, просто "ловит" он только те сообщения, которые отредактированы. Что ж, ничего сложного, пишем!

```
@bot.message_handler(func=lambda message: True)
def any_message(message):
    bot.reply_to(message, "Сам {!s}".format(message.text))

@bot.edited_message_handler(func=lambda message: True)
def edit_message(message):
    bot.edit_message_text(chat_id=message.chat.id,
                          text= "Сам {!s}".format(message.text),
                          message_id=message.message_id + 1)
```

Заметили? Да, при вызове `edit_message_text` надо указать `message_id` на единицу бОльший, чем тот, который прислан сервером, потому что сервер сообщает о сообщении **от** пользователя, а нам нужно редактировать сообщение бота, которое шло за ним следом. И вот как это будет выглядеть (это одни и те же сообщения, что видно по метке "изм." около моего)

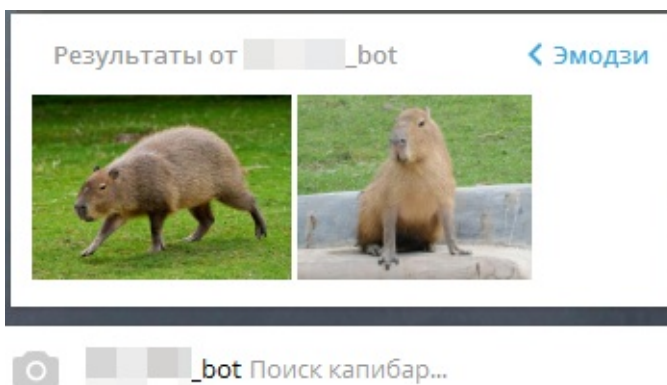


Кэшированный инлайн

Когда [инлайн-боты](#) только появились, то в качестве источника данных для ответов надо было указывать внешние ссылки, причем с ограничениями по размеру указываемого файла. Очевидно, такой подход мог быть не очень быстрым, а чем дольше пользователь ждёт, тем он менее доволен результатами работы бота :) В итоге, в Bot API v2 инлайн-режиму разрешили в качестве источника для медиа использовать `file_id` уже имеющихся на сервере файлов (*напомню, что `file_id` для одного и того же файла будут разниться от бота к боту*) Итак, у меня есть `file_id` двух фотографий с [капибарами](#) (как получить `file_id` загружаемых боту фотографий, считайте это заданием для самоподготовки), надо на любой инлайн-запрос (даже пустой) предложить эти 2 изображения. По сути, всё сводится к замене типа `InlineQueryResultPhoto` на тип `InlineQueryResultCachedPhoto`

```
@bot.inline_handler(func=lambda query: True)
def inline_mode(query):
    capibara1 = types.InlineQueryResultCachedPhoto(
        id="1",
        photo_file_id="AgADAgAD6rMxGyBnGwABgBmcoHgy01IENAAQSYK_1gyoAAU-5aQACAg",
        caption="Это капибара №1"
    )
    capibara2 = types.InlineQueryResultCachedPhoto(
        id="2",
        photo_file_id="AgADAgAD67MxGyBnGwABCvqPIYxMoNHENAAS51Hj088y_Z0ffaQABAg",
        caption="Это капибара №2"
    )
    bot.answer_inline_query(query.id, [capibara1, capibara2])
```

Запускаем бота. Ура, теперь мы умеем очень быстро предлагать разных капибар нашим пользователям :)



Помимо фотографий, из "кэша" можно показывать любые типы, поддерживаемые мессенджером: видео, аудио, стикеры, файлы (пока что только pdf и zip).

На этом всё. **Хороших ботов!**

Дополнительное чтение

В этом разделе расположены заметки, касающиеся напрямую Telegram Bot API, но выходящие за рамки моих "уроков".

- [Работа с библиотекой pyTelegramBotAPI](#)
- [Отправляем сообщения через Telegram Bot API из Powershell](#)
- [Запускаем несколько ботов на одной машине: CherryPy only](#)
- [Запускаем несколько ботов на одной машине: nginx + CherryPy](#)
- [Упаковываем ботов правильно: файлы *.ini и *.pyz](#)

Работа с библиотекой pyTelegramBotAPI.

Хотелось бы обратить ваше внимание на особенности работы с [pyTelegramBotAPI](#), оберткой, которую мы будем использовать.

Те, кто используют другую обертку или вообще пишут на другом ЯП, могут спокойно прекратить чтение, ~~съесть ещё этих мягких французских булочек и выпить чаю.~~

Декораторы

В [уроке №1](#) мы создавали так называемого "слушателя", в котором обрабатывали входящее сообщение. Это, несомненно, круто, но как только наш бот начинает расти, это становится проблемой. В огромном количестве конструкций `if-elif-else` можно легко запутаться.

Для решения этой проблемы автор библиотеки написал хэндлеры (или декораторы, кому как проще).

Давайте сразу рассмотрим их на примере.

```
# Обработчик команд '/start' и '/help'.
@bot.message_handler(commands=['start', 'help'])
def handle_start_help(message):
    pass

# Обработчик для документов и аудиофайлов
@bot.message_handler(content_types=['document', 'audio'])
def handle_docs_audio(message):
    pass

# Обработчик сообщений, подходящих под указанное регулярное выражение
@bot.message_handler(regex="SOME_REGEX")
def handle_message(message):
    pass

# Обработчик сообщений, содержащих документ с mime_type 'text/plain' (обычный текст)
@bot.message_handler(func=lambda message: message.document.mime_type == 'text/plain',
content_types=['document'])
def handle_text_doc(message):
    pass
```

Таким образом, мы можем разнести код по различным функциям и не кучковать его в одном месте, что повышает читабельность. **Обратите внимание:** декораторы будут проверяться в порядке их следования в коде, по принципу "какой первый подошел, тот

и используем". Под "подошел" будем понимать равенство нашей лямбда-функции значению `True`.

В итоге, код из первого урока можно переписать так:

```
# -*- coding: utf-8 -*-
import time
import telebot
from lesson_01 import config

bot = telebot.TeleBot(config.token)

@bot.message_handler(func=lambda message: True, content_types=['text'])
def echo_msg(message):
    bot.send_message(message.chat.id, message.text)

if __name__ == '__main__':
    bot.polling(none_stop=True)
```

Как видите, не так уж и сложно!

Кастомные клавиатуры

Одной из ключевых "фишек" Bot API стало появление кастомных клавиатур. На самом деле, это обычные шаблоны сообщений, не более, поэтому относиться к ним надо осторожно. Давайте просто покажу на примере, как делается такая клавиатура:

```
from telebot import types
markup = types.ReplyKeyboardMarkup()
markup.row('a', 'v')
markup.row('c', 'd', 'e')
bot.send_message(message.chat.id, "Choose one letter:", reply_markup=markup)
```



Создаем объект типа `ReplyKeyboardMarkup()`, добавляем построчно элементы (или можно сделать автоматическое разделение на строки при помощи аргумента `row_width`, но это выходит за рамки урока), передаем полученную разметку в метод `send_message()`. Если же мы хотим убрать кастомную клавиатуру вообще, заменив её на кнопку [/], то вместо объекта `ReplyKeyboardMarkup()` надо создать объект `ReplyKeyboardHide()` и просто передать его в `send_message()`. Всё просто!

Отправляем сообщения через Telegram Bot API из Powershell.

Powershell - это очень крутая штука в руках толкового Windows-сисадмина, которая умеет кучу всего. Сегодня я расскажу, как можно отправлять текстовые сообщения себе в Telegram через бота.

Сразу о минусах: во-первых, у меня не получилось передавать какие-либо файлы, т.к. подставить в качестве аргумента "photo" в JSON-массиве содержимое файла в Powershell не представляется возможным, 2 часа гугления рабочего решения не дали. Во-вторых, в силу особенностей Powershell в русифицированных виндах кириллический текст отправляется в виде вопросительных знаков, а попытка в явном виде преобразовать текст в UTF-8 приводит к фейлу. В комментариях к этому уроку подсказали, как добавить поддержку юникода (в который входит и кириллица).

Для чего вообще тогда нужен этот скрипт? Ну, например, если на сервере планировщиком запускаются какие-либо бэкапы, то было бы удобно получать информацию о результатах в Telegram. Или, например, можно сообщать о неполадках на удаленной машине в канал, который читают несколько системных администраторов. В общем, кому надо будет, тот придумает назначение сам. Итак, поехали.

Прежде всего, давайте определимся с аргументами командной строки. Я сделал обязательными `chat_id` и `text` (согласно [документации](#)) и дополнительно сделал возможной поддержку Markdown и отсутствие генерации превью для ссылок:

```
param(
    [string]$chat_id = $(Throw "'-chat_id' argument is mandatory"),
    [string]$text = $(Throw "'-text' argument is mandatory"),
    [switch]$markdown,
    [switch]$nopreview
)
```

В случае, если не указан какой-либо из обязательных аргументов, скрипт выпадет с ошибкой, описанной в скобках.

Переменная типа `switch` работает так: если соответствующий аргумент присутствует в вызове, то значение переменной устанавливается в `$True`, если нет - `$False`. Нас такой расклад не совсем устраивает, поэтому добавим парочку проверок:


```
if($nopreview) { $preview_mode = "True" }
if($markdown) { $markdown_mode = "Markdown" } else { $markdown_mode = ""}
```

Теперь сформируем тело сообщения:

```
$payload = @{
    "chat_id" = $chat_id;
    "text" = $text
    "parse_mode" = $markdown_mode;
    "disable_web_page_preview" = $preview_mode;
}
```

И, наконец, осуществим http-запрос к API:

```
Invoke-WebRequest `
    -Uri ("https://api.telegram.org/bot{0}/sendMessage" -f $token) `
    -Method Post `
    -ContentType "application/json;charset=utf-8" `
    -Body (ConvertTo-Json -Compress -InputObject $payload)
```

Вот и всё! Весь код приведён в следующем листинге:

```
param(
    [string]$chat_id = $(Throw "'-chat_id' argument is mandatory"),
    [string]$text = $(Throw "'-text' argument is mandatory"),
    [switch]$markdown,
    [switch]$nopreview
)

$token = "BAШ TOKEN"
if($nopreview) { $preview_mode = "True" }
if($markdown) { $markdown_mode = "Markdown" } else { $markdown_mode = ""}

$payload = @{
    "chat_id" = $chat_id;
    "text" = $text
    "parse_mode" = $markdown_mode;
    "disable_web_page_preview" = $preview_mode;
}

Invoke-WebRequest `
    -Uri ("https://api.telegram.org/bot{0}/sendMessage" -f $token) `
    -Method Post `
    -ContentType "application/json;charset=utf-8" `
    -Body (ConvertTo-Json -Compress -InputObject $payload)
```

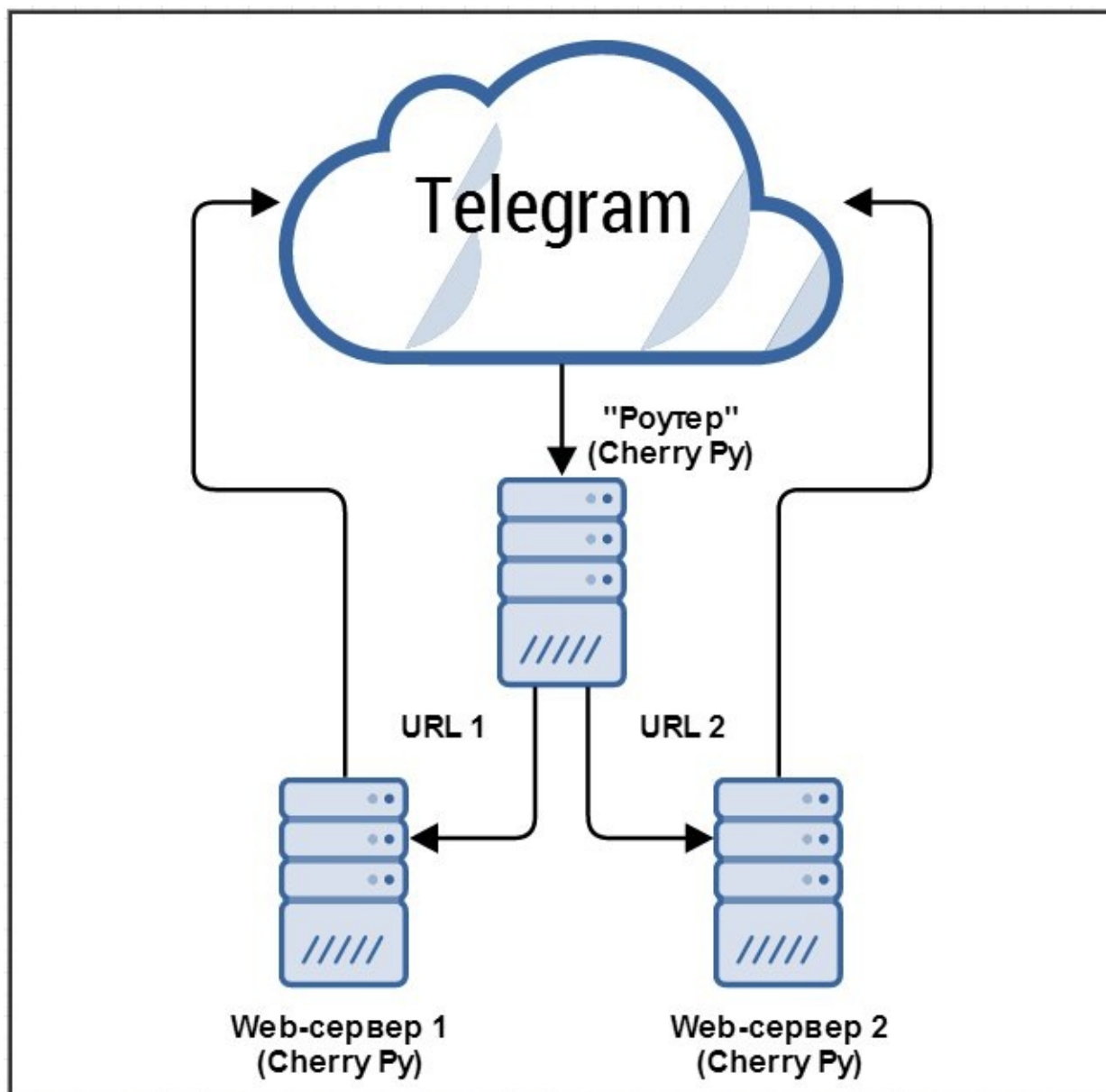

Запускаем несколько ботов на одной машине: CherryPy only

По моему мнению (а оно может не совпадать с вашим), боты на вебхуках надёжнее ботов, использующих поллинг.

Связано это в первую очередь с моими воспоминаниями из августа-сентября 2015 года, когда каждую ночь падали боты из-за Gateway Timeout'ов. Затем в Bot API добавили поддержку самоподписанных сертификатов и я начал использовать их, благо это бесплатно.

В [уроке номер четыре](#) я писал, что т.к. портов для использования вебхуков всего 4, то, казалось бы, можно на одной машине запустить всего четырёх ботов, и что есть решение этой проблемы. Вот об этом сейчас и пойдет речь. Я планирую разбить материал на две части: в первой мы научимся запускать сколько угодно ботов при помощи одних лишь серверов [CherryPy](#) и самоподписанных сертификатов, во второй же вместо основного сервера поставим [nginx](#), а вместо самоподписанных сертификатов - бесплатные от [Let's Encrypt](#).

Общая схема взаимодействия



Подготавливаем "роутер"

Давайте для начала создадим наш "роутер", то есть, сервер CherryPy, который будет принимать все сообщения и раскидывать их по нужным ботам. Условимся также, что наш сервер будет иметь IP 122.122.122.122 и вебхуки от первого бота будут приходить на адрес <https://122.122.122.122/AAAA>, а от второго на <https://122.122.122.122/ZZZZ>.

Предполагается, что вы уже прочитали [4-й урок](#) и структура вебхук-ботов вас не пугает.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import cherrypy
import requests
import telebot
```

```

WEBHOOK_HOST = 'Здесь.Ваш.ИР.Адрес'
WEBHOOK_PORT = 443 # 443, 80, 88 или 8443
WEBHOOK_LISTEN = '0.0.0.0' # Слушаем отовсюду
WEBHOOK_SSL_CERT = 'webhook_cert.pem' # Путь к сертификату
WEBHOOK_SSL_PRIV = 'webhook_pkey.pem' # Путь к закрытому ключу
WEBHOOK_URL_BASE = "https://{!s}:{!s}".format(WEBHOOK_HOST, WEBHOOK_PORT)

BOT_1_TOKEN = "Токен первого бота"
BOT_2_TOKEN = "Токен второго бота"

# Вводим здесь IP-адреса и порты, куда перенаправлять входящие запросы.
# Т.к. всё на одной машине, то используем локалхост + какие-нибудь свободные порты.
# https в данном случае не нужен, шифровать незачем.
BOT_1_ADDRESS = "http://127.0.0.1:7771"
BOT_2_ADDRESS = "http://127.0.0.1:7772"

bot_1 = telebot.TeleBot(BOT_1_TOKEN)
bot_2 = telebot.TeleBot(BOT_2_TOKEN)

# Описываем наш сервер
class WebhookServer(object):

    # Первый бот (название функции = последняя часть URL вебхука)
    @cherry.py.expose
    def AAAA(self):
        if 'content-length' in cherry.py.request.headers and \
            'content-type' in cherry.py.request.headers and \
            cherry.py.request.headers['content-type'] == 'application/json':
            length = int(cherry.py.request.headers['content-length'])
            json_string = cherry.py.request.body.read(length).decode("utf-8")
            # Вот эта строчка и пересылает все входящие сообщения на нужного бота
            requests.post(BOT_1_ADDRESS, data=json_string)
            return ''
        else:
            raise cherry.py.HTTPError(403)

    # Второй бот (действуем аналогично)
    @cherry.py.expose
    def ZZZZ(self):
        if 'content-length' in cherry.py.request.headers and \
            'content-type' in cherry.py.request.headers and \
            cherry.py.request.headers['content-type'] == 'application/json':
            length = int(cherry.py.request.headers['content-length'])
            json_string = cherry.py.request.body.read(length).decode("utf-8")
            requests.post(BOT_2_ADDRESS, data=json_string)
            return ''
        else:
            raise cherry.py.HTTPError(403)

if __name__ == '__main__':

    bot_1.remove_webhook()
    bot_1.set_webhook(url='https://122.122.122.122/AAAA',

```

```
        certificate=open(WEBHOOK_SSL_CERT, 'r'))

bot_2.remove_webhook()
bot_2.set_webhook(url='https://122.122.122.122/ZZZZ',
                  certificate=open(WEBHOOK_SSL_CERT, 'r'))

cherry.py.config.update({
    'server.socket_host': WEBHOOK_LISTEN,
    'server.socket_port': WEBHOOK_PORT,
    'server.ssl_module': 'builtin',
    'server.ssl_certificate': WEBHOOK_SSL_CERT,
    'server.ssl_private_key': WEBHOOK_SSL_PRIV,
    'engine.autoreload.on': False
})
cherry.py.quickstart(WebhookServer(), '/', {'/': {}})
```

Подготавливаем ботов

Создадим их две штуки, каждый из которых будет на команду `/start` представляться по своему номеру.

Номер раз:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import cherrypy
import telebot

BOT_TOKEN = "токен нашего бота"

bot = telebot.TeleBot(BOT_TOKEN)

@bot.message_handler(commands=["start"])
def command_start(message):
    bot.send_message(message.chat.id, "Привет! Я бот номер 1")

class WebhookServer(object):
    # index равнозначно /, т.к. отсутствию части после ip-адреса (грубо говоря)
    @cherrypy.expose
    def index(self):
        length = int(cherrypy.request.headers['content-length'])
        json_string = cherrypy.request.body.read(length).decode("utf-8")
        update = telebot.types.Update.de_json(json_string)
        bot.process_new_updates([update])
        return ''

if __name__ == '__main__':
    cherrypy.config.update({
        'server.socket_host': '127.0.0.1',
        'server.socket_port': 7771,
        'engine.autoreload.on': False
    })
    cherrypy.quickstart(WebhookServer(), '/', {'/': {}})
```

Второй делается аналогично, только ставим порт 7772 и меняем сообщение по команде `/start`.

Запускаем "роутер", запускаем ботов. Если мы всё сделали правильно, то при создании чата с первым ботом, сначала вебхук получит "роутер", перешлет его первому серверу, который отправит сообщение непосредственно в Telegram, в точности так же, как на схеме выше.

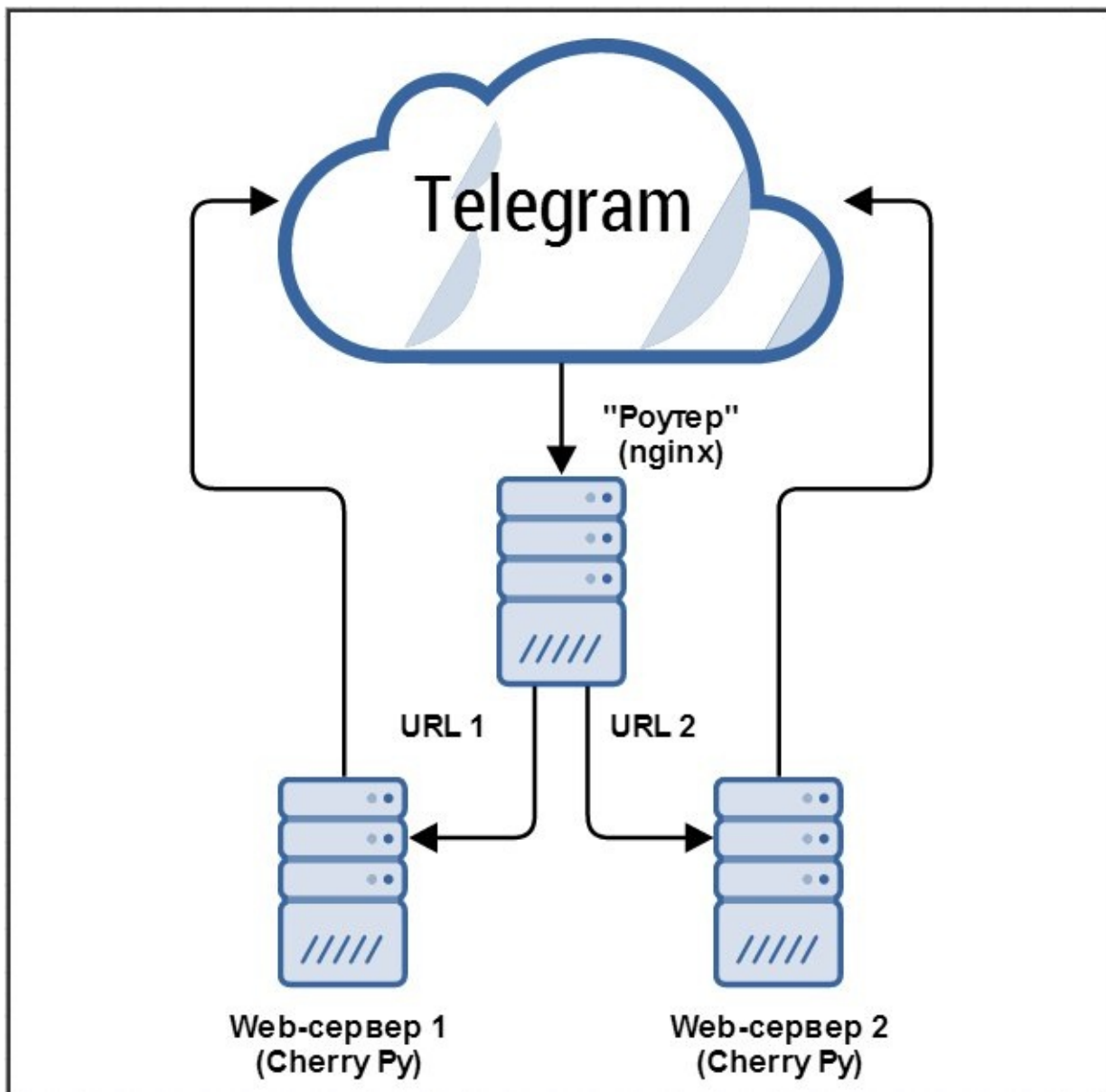
Запускаем несколько ботов на одной машине: nginx + CherryPy

В [прошлый раз](#) мы говорили о запуске нескольких ботов при помощи только лишь серверов CherryPy, а сегодня будем поступать правильно - в качестве роутера будем использовать [nginx](#), а самоподписанные сертификаты заменим на официальные от [Let's Encrypt](#).

Получаем сертификат от Let's Encrypt

Используйте официальную утилиту [certbot](#) для получения сертификата. Укажите свою почту и доменное имя вашего сервера (просто IP-адрес не подойдет). Обратите внимание на дату окончания действия сертификата! Каждые 90 дней его надо обновлять (данный процесс легко автоматизируется).

Общая схема взаимодействия



Настраиваем nginx

Скачиваем и устанавливаем веб-сервер nginx любым удобным для вас способом и открываем `nginx.conf` в каталоге `/etc/nginx`. Внутри блока `http {}` создаем блок `server{}` и заполняем его следующим образом (все основные параметры я взял из [прошлой части](#)):

```
server {
    listen 443 ssl;
    server_name (адрес вашего сервера);

    ssl_protocols      TLSv1 TLSv1.1 TLSv1.2;
    ssl_certificate /etc/letsencrypt/live/(адрес вашего сервера)/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/(адрес вашего сервера)/privkey.pem;

    # Первый бот
    location /AAAA/ {
        proxy_pass      http://127.0.0.1:7771/;
        proxy_redirect   off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Host $server_name;
    }

    # Второй бот
    location /ZZZZ/ {
        proxy_pass      http://127.0.0.1:7772/;
        proxy_redirect   off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Host $server_name;
    }
}
```

Сохраняем файл конфигурации и перезапускаем службу: `sudo service nginx reload` .

Единственное, что надо изменить в самих ботах - это добавить `bot.remove_webhook()` и `bot.set_webhook("https://(адрес вашего сервера)/xxxx)` , где xxxx в нашем случае либо AAAA, либо ZZZZ, при этом аргумент `certificate` в методе `set_webhook` уже не нужен, т.к. сертификат не самоподписанный.

Упаковываем ботов правильно: файлы *.ini и *.pyz

В Python 3.5 появился модуль [zipapp](#), позволяющий "упаковать" каталог с кодом в один исполняемый интерпретатором файл с расширением `.pyz`. К слову, запускать (но без создания) могут версии Python, начиная с 2.6.

Казалось бы, всё отлично, давайте упаковывать каждого бота в один исполняемый файл и не заморачиваться с пересылкой архивов с кучей файлов внутри. Но внимательный читатель заметит, что прежде для конфигурирования ботов мы создавали отдельный файл `config.py`, который при использовании упаковщика, очевидно, попадет внутрь `pyz`-архива. В этом нет никакой проблемы, если бот создается исключительно для себя, но при выкладывании исходников в общий доступ, надо что-то придумать, чтобы конечный пользователь мог вписать нужный токен и вообще указать необходимые параметры. И тут на помощь придёт модуль [configparser](#). Я не буду тратить время на описание формата ini-файла, который использует этот модуль, желающие могут пройти по ссылке и лицезреть несколько простых и доступных примеров.

Итак, давайте создадим простого бота на поллинге, упакуем его, создадим файл конфигурации и проверим. Для пущего разнообразия, все действия будут производиться в ОС Windows.

Создание файла конфигурации

Давайте создадим файл `config.ini` следующего содержания:

```
[DEFAULT]
Token = 1234567:ABCDEFGHIJKlmnoPQrsTuvWXYZ

[Bot Specific]
Answer message = 123
```

Пусть для примера у нас будет две секции: `[DEFAULT]` (по умолчанию) и `[Bot Specific]`. Для тех, кто всё-таки не стал читать документацию по `configparser`, укажу, что названия секций в коде case-sensitive, а названия ключей – нет.

Пишем код

Чтобы показать все прелести упаковки кода в `pyz`-файл, создадим проект с такой структурой (уточню: файл `another.py` лежит в каталоге `other`):

```
|other
|__another.py
|bot.py
|requirements.txt
```

Файл **another.py** будет очень простым – всего лишь вывод строки:

```
# -*- coding: utf-8 -*-

def just_print():
    return "Hello world!"
```

Файл **bot.py** будет ненамного сложнее: создадим функцию, которая будет реагировать только на сообщение, текст которого указан в конфигурационном файле. Обратите также внимание на последние строки, если вам вдруг захочется вынести содержимое функции `main()` в последний `if` – не делайте этого, позднее увидите, почему.

```
# -*- coding: utf-8 -*-

import telebot
import configparser
from other.another import just_print

config = configparser.ConfigParser()
config.read("config.ini")

bot = telebot.TeleBot(config["DEFAULT"]["Token"])

@bot.message_handler(func=lambda message: message.text == config["Bot Specific"]["answer message"])
def reply_to(message):
    bot.send_message(message.chat.id, "Hello there!")

def main():
    print(just_print())
    bot.polling(none_stop=True)

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

По-хорошему, конечно, надо проверять наличие файла `config.ini` , но для простоты кода мы этого делать не будем.

Давайте сделаем ещё вот что: создадим файл `requirements.txt` , в котором укажем, какие библиотеки нужны нашему боту, чтобы получатель мог одной командой установить все зависимости и радоваться жизни. В нашем случае в файл достаточно записать одну строку `pytelegrambotapi>=1.4.2` , т.к. эта библиотека потянет за собой необходимые боту `requests` , `six` и т.д.

Упаковка в pyz-файл

Настало время объединить все py-файлы в один исполняемый архив. Напоминаю, в этом уроке мы работаем в Windows. Предположим, что наш проект лежит в каталоге `C:\project` . Уточним сразу пару моментов: во-первых, нам необходимо указать имя главной вызываемой функции, в нашем случае это функция `main()` в модуле `bot.py` (именно поэтому я вынес код запуска бота в отдельный метод); во-вторых, при создании можно не указывать имя выходного файла, тогда в родительском каталоге будет создан файл с названием папки и расширением `pyz` , если вы хотите указать своё название, то расширение нужно будет ввести вручную и файл будет по умолчанию создан внутри упаковываемого каталога (но не в самом архиве, очевидно).

Итак, начинаем упаковку: `C:\Python35\python.exe -m zipapp "C:\project" -m bot:main -o mybot.pyz` .

В результате получится файл `C:\project\mybot.pyz` . Как же теперь поступить человеку, который захочет запустить вашего бота у себя?

1. Скачать себе в компьютер файлы `mybot.pyz` , `config.ini` и `requirements.txt` , например, в каталог `D:\Bot`
2. Указать нужные параметры в файле `config.ini` .
3. Установить зависимости командой `C:\Python34\python.exe -m pip install -r requirements.txt` . Заметьте, я намеренно назвал каталог `Python34` вместо `Python35` с намёком на более старую версию используемого интерпретатора. Python 3.5 нужен только для упаковки в `pyz` .
4. Запустить бота командой `C:\Python34\python.exe -m D:\Bot\mybot.pyz` .
5. (по желанию) Радоваться жизни.

Собственно, это всё. Конечно, в нашем примере число файлов уменьшилось совсем незначительно, но для крупных ботов, когда число модулей превосходит десяток, такой метод упаковки может показаться крайне удобным.