

Урок 2. Определяем настройки бота и создаем первые обработчики

ОГЛАВЛЕНИЕ

[ОПРЕДЕЛЯЕМ НАСТРОЙКИ БОТА 3](#_Toc79587106)

[СОЗДАЕМ БАЗОВЫЙ КЛАСС-ОБРАБОТЧИК 6](#_Toc79587107)

[СОЗДАЕМ МОДУЛЬ, ОБЕСПЕЧИВАЮЩИЙ ФОРМИРОВАНИЕ РАЗМЕТКИ ИНТЕРФЕЙСА 7](#_Toc79587108)

[СОЗДАЕМ ГЛАВНЫЙ КЛАСС-ОБРАБОТЧИК 10](#_Toc79587109)

[СОЗДАЕМ ЗАПУСКАЕМЫЙ ФАЙЛ 11](#_Toc79587110)

[ВЫПОЛНЯЕМ РЕГИСТРАЦИЮ БОТА 12](#_Toc79587111)

[ЗАКЛЮЧЕНИЕ 16](#_Toc79587112)

[КЛЮЧЕВЫЕ ВЫВОДЫ 16](#_Toc79587113)



# ОПРЕДЕЛЯЕМ НАСТРОЙКИ БОТА

Считается хорошей практикой при разработке приложений какие-то общие постоянные настройки (константы) выносить в отдельный модуль. Это упрощает процесс их дальнейшего редактирования. Создадим в нашем проекте отдельную директорию **settings**, в которой будем хранить модули с различного рода настройками. Первый модулем станет **config.py**.

**Листинг 1. step\_2/settings/config.py**

|  |
| --- |
| **import** os *# импортируем модуль emoji для отображения эмоджи* **from** emoji **import** emojize  *# токен выдается при регистрации приложения* TOKEN = **'1903805843:AAGEK\_TCRKTDSwgZ7AQatn2ex74hN4lyVbU'** *# название БД* NAME\_DB = **'products.sqlite'** *# версия приложения* VERSION = **'0.0.1'** *# автор приложния* AUTHOR = **'User'** *# родительская директория* BASE\_DIR = os.path.dirname(os.path.abspath(\_\_file\_\_)) *# путь до базы данных* DATABASE = os.path.join(**'sqlite:///'**+BASE\_DIR, NAME\_DB)  COUNT = 0  *# кнопки управления* KEYBOARD = {  **'CHOOSE\_GOODS'**: emojize(**':open\_file\_folder: Выбрать товар'**),  **'INFO'**: emojize(**':speech\_balloon: О магазине'**),  **'SETTINGS'**: emojize(**'⚙️ Настройки'**),  **'SEMIPRODUCT'**: emojize(**':pizza: Полуфабрикаты'**),  **'GROCERY'**: emojize(**':bread: Бакалея'**),  **'ICE\_CREAM'**: emojize(**':shaved\_ice: Мороженое'**),  **'<<'**: emojize(**'⏪'**),  **'>>'**: emojize(**'⏩'**),  **'BACK\_STEP'**: emojize(**'◀️'**),  **'NEXT\_STEP'**: emojize(**'▶️'**),  **'ORDER'**: emojize(**'✅ ЗАКАЗ'**),  **'X'**: emojize(**'❌'**),  **'DOUWN'**: emojize(**'🔽'**),  **'AMOUNT\_PRODUCT'**: COUNT,  **'AMOUNT\_ORDERS'**: COUNT,  **'UP'**: emojize(**'🔼'**),  **'APPLAY'**: **'✅ Оформить заказ'**,  **'COPY'**: **'©️'** }  *# id категорий продуктов* CATEGORY = {  **'SEMIPRODUCT'**: 1,  **'GROCERY'**: 2,  **'ICE\_CREAM'**: 3, }  *# названия команд* COMMANDS = {  **'START'**: **"start"**,  **'HELP'**: **"help"**, } |

Обратите внимание, что все переменные этого файла определены в верхнем регистре. Это так называемые константы – постоянные параметры проекта. Обсудим некоторые из них.

TOKEN = **''***# название БД*NAME\_DB = **'products.sqlite'***# версия приложения*VERSION = **'0.0.1'***# автор приложния*AUTHOR = **'User'**

Значение переменной TOKEN пока оставим пустым. Заполним его позднее. Здесь указывается значение токена, которое генерируется при регистрации бота. Пока оставим эту переменную без значения, но позднее зададим его. Поскольку мы будем работать с файловой СУБД SQLite, нужно указать имя файла БД. Также можно задать версию и имя автора проекта.

|  |
| --- |
| *# родительская директория* BASE\_DIR = os.path.dirname(os.path.abspath(\_\_file\_\_)) *# путь до базы данных* DATABASE = os.path.join(**'sqlite:///'**+BASE\_DIR, NAME\_DB) |

Определим путь сохранения файла БД. Пусть этот будет та же директория **settings**, расположенная в корне проекта.

|  |
| --- |
| **from** emoji **import** emojize  KEYBOARD = {  **'CHOOSE\_GOODS'**: emojize(**':open\_file\_folder: Выбрать товар'**),  **'INFO'**: emojize(**':speech\_balloon: О магазине'**),  **'SETTINGS'**: emojize(**'⚙️ Настройки'**),  **'SEMIPRODUCT'**: emojize(**':pizza: Полуфабрикаты'**),  **'GROCERY'**: emojize(**':bread: Бакалея'**),  **'ICE\_CREAM'**: emojize(**':shaved\_ice: Мороженое'**),  **'<<'**: emojize(**'⏪'**),  **'>>'**: emojize(**'⏩'**),  **'BACK\_STEP'**: emojize(**'◀️'**),  **'NEXT\_STEP'**: emojize(**'▶️'**),  **'ORDER'**: emojize(**'✅ ЗАКАЗ'**),  **'X'**: emojize(**'❌'**),  **'DOUWN'**: emojize(**'🔽'**),  **'AMOUNT\_PRODUCT'**: COUNT,  **'AMOUNT\_ORDERS'**: COUNT,  **'UP'**: emojize(**'🔼'**),  **'APPLAY'**: **'✅ Оформить заказ'**,  **'COPY'**: **'©️'** } |

Функционал нашего проекта позволяет выбирать различные виды товаров. Для перехода в режим выбора товара применяется кнопка **'Выбрать товар'**. Кнопка **'О магазине'** позволяет получить информацию о нашем боте. Кнопка **'Настройки'** указывает на панель с информацией об опциях бота. Следующие три кнопки соответствуют трем номенклатурным позициям нашего ассортимента. Кнопки **'<<'** и **'>>'** позволяют перемещаться между пунктами меню. Кнопки **'BACK\_STEP'** и **'NEXT\_STEP'** обеспечивают перемещение между выбранными позициями заказа, например для редактирования количества каждой позиции. Далее следует кнопка, обеспечивающая переход к текущему заказу. Кнопка **'X'** применяется для удаления, кнопки **'DOUWN'** и **'UP'** нужны для изменения количества выбранного товара в заказе. Параметры **'AMOUNT\_PRODUCT'** и **'AMOUNT\_ORDERS'** соответствуют значениям количества отдельного товара в заказе и количества номенклатурных позиций в заказе. Последние две кнопки – подтверждение заказа и метка «Все права защищены».

Набор доступных кнопок определен через словарь. Для придания элементам управления меню стильного внешнего вида применяется модуль emojize («эмодзи»). Список элементов меню может быть со временем расширен.

Также в список констант внесем идентификаторы номенклатурных позиций и названия простейших базовых команд для управления ботом.

|  |
| --- |
| *# id категорий продуктов* CATEGORY = {  **'SEMIPRODUCT'**: 1,  **'GROCERY'**: 2,  **'ICE\_CREAM'**: 3, }  *# названия команд* COMMANDS = {  **'START'**: **"start"**,  **'HELP'**: **"help"**, } |

# СОЗДАЕМ БАЗОВЫЙ КЛАСС-ОБРАБОТЧИК

В текущем и последующем блоках курса нам предстоит реализовать логику нашего бота (его бэкенд). Это непростая задача и мы будем решать ее постепенно. Создадим в корне проекта директорию **handlers** и поместим в нее модуль **hanlder.py**, в котором разместим код нашего главного базового класса-обработчика.

**Листинг 2. step\_2/handlers/handler.py**

|  |
| --- |
| *# импортируем библиотеку abc для реализации абстрактных классов* **import** abc *# импортируем разметку клавиатуры и клавиш* **from** markup.markup **import** Keyboards *# импортируем класс-менеджер для работы с библиотекой* **from** data\_base.dbalchemy **import** DBManager   **class** Handler(metaclass=abc.ABCMeta):   **def** \_\_init\_\_(self, bot):  *# получаем объект бота* self.bot = bot  *# инициализируем разметку кнопок* self.keybords = Keyboards()  *# инициализируем менеджер для работы с БД* self.BD = DBManager()   @abc.abstractmethod  **def** handle(self):  **pass** |

В приведенном листинге мы определяем базовый абстрактный класс-обработчик. Текущая структура его интерфейса говорит о том, что для каждого обработчика мы определим такие параметры, как объект бота (будет передаваться при инициализации объекта обработчика), объект-инициализатор разметки кнопок, объект-инициализатор менеджера для работы с БД.

Кроме того, вспомним зачем применяется декоратор @abc.abstractmethod. Его задача показать, что задекорированный метод должен быть обязательно переопределен в классах-наследниках.

В приведенном листинге видим два важных импорта:

|  |
| --- |
| *# импортируем разметку клавиатуры и клавиш* **from** markup.markup **import** Keyboards *# импортируем класс-менеджер для работы с библиотекой* **from** data\_base.dbalchemy **import** DBManager |

Об этих импортах поговорим далее.

# СОЗДАЕМ МОДУЛЬ, ОБЕСПЕЧИВАЮЩИЙ ФОРМИРОВАНИЕ РАЗМЕТКИ ИНТЕРФЕЙСА

В корне проекта создадим директорию **markup**, в которую поместим модуль **markup.py** и добавим в него следующий код. Этот код будет обеспечивать формирование интерфейса нашего бота.

**Листинг 3. step\_2/markup/markup.py**

|  |
| --- |
| *# импортируем специальные типы телеграм бота для создания элементов интерфейса* **from** telebot.types **import** KeyboardButton *# импортируем настройки и утилиты* **from** settings **import** config *# импортируем класс-менеджер для работы с библиотекой* **from** data\_base.dbalchemy **import** DBManager   **class** Keyboards:  *"""  Класс Keyboards предназначен для создания и разметки интерфейса бота  """  # инициализация разметки* **def** \_\_init\_\_(self):  self.markup = **None** *# инициализируем менеджер для работы с БД* self.BD = DBManager()   **def** set\_btn(self, name, step=0, quantity=0):  *"""  Создает и возвращает кнопку по входным параметрам  """* **return** KeyboardButton(config.KEYBOARD[name]) |

Пока в этом модуле только один класс с минимальным кодом. Давайте представим, что это пока просто заглушка. Позднее мы начнем наполнять кодом этот класс и обсудим его подробно.

Давайте также создадим и заглушку для класса **DBManager**. Его код будет находиться в модуле **dbalchemy.py** папки **data\_base**.

**Листинг 4. step\_2/data\_base/dbalchemy.py**

|  |
| --- |
| **class** Singleton(type):  *"""  Патерн Singleton предоставляет механизм создания одного  и только одного объекта класса,  и предоставление к нему глобальную точку доступа.  """* **def** \_\_init\_\_(cls, name, bases, attrs, \*\*kwargs):  super().\_\_init\_\_(name, bases, attrs)  cls.\_\_instance = **None   def** \_\_call\_\_(cls, \*args, \*\*kwargs):  **if** cls.\_\_instance **is None**:  cls.\_\_instance = super().\_\_call\_\_(\*args, \*\*kwargs)  **return** cls.\_\_instance   **class** DBManager(metaclass=Singleton):  *"""   Класс менеджер для работы с БД   """* **def** \_\_init\_\_(self):  **pass** |

В этом листинге представлен класс **DBManager**, выполняющий роль менеджера для работы с базой данных. При описании класса применен интересный атрибут **metaclass** со значением **Singleton**. Выше мы видим код класса **Singleton**. Стоит отметить, что в Python существует такое понятие как метакласс. Это сверхкласс, контролирующий поведение своих подчиненных классов. Не стоит путать понятие метакласса с наследованием. Цель наследования – передать интерфейс (атрибуты и методы) некоторого базового класса классам-потомкам. Метакласс же отвечает за контроль поведения подчиненных классов. В нашем случае метакласс контролирует, чтобы для подчиненного класса **DBManager** всегда создавался один и тот же объект.

Важно отметить, что у каждого класса в Python (как встроенного, так и пользовательского) по умолчанию метаклассом является **type**. Если необходимо вмешаться в его работу и добавить свою логику (т.е. написать свой метакласс), то нужно создать этот самый свой метакласс и унаследовать его от type.

|  |
| --- |
| **def** \_\_init\_\_(cls, name, bases, attrs, \*\*kwargs):  super().\_\_init\_\_(name, bases, attrs)  cls.\_\_instance = **None** |

Метакласс в методе **\_\_init\_\_()** принимает параметры подчиненного класса, на которые может влиять, в частности ссылку на подчиненный класс, его имя, его базовые классы, позиционные и именованные аргументы. Изначально мы говорим, что значением объекта нашего подчиненного класса является **None**. При создании объекта класса и его возврате (срабатывает метод **\_\_call\_\_()**) создается объект класса, ссылку на который мы помещаем в атрибут **cls.\_\_instance**. Этот атрибут доступен на уровне всего метакласса и в нем будет всегда находиться ссылка на тот объект, который мы создали в первый раз, т.е. объект не меняется, он всегда один и тот же.

Синглтон это популярный паттерн (шаблон проектирования). Работа с базой данных – это классический пример использования данного паттерна. Представьте, что у вас в рамках одного модуля несколько скриптов работают с базой данных. Необходима глобальная переменная соединения и нужно постоянно проверять, есть ли подключение к базе. Благодаря синглтону создается соединение, если оно еще не было установлено, либо возвращается готовая ссылка.

# СОЗДАЕМ ГЛАВНЫЙ КЛАСС-ОБРАБОТЧИК

Создадим центральный узел управления обработчиками бота. Пока сделаем его заглушкой, но позднее дополним необходимым кодом.

**Листинг 5. step\_2/handlers/handler\_main.py**

|  |
| --- |
| **class** HandlerMain:  *"""  Класс компоновщик  """* **def** \_\_init\_\_(self, bot):  *# получаем нашего бота* self.bot = bot  *# здесь будет иницаилизация обработчиков* **def** handle(self):  *# здесь будет запуск обработчиков* **pass** |

В этом блоке будет происходить инициализация и запусков основных обработчиков событий нашего бота.

# СОЗДАЕМ ЗАПУСКАЕМЫЙ ФАЙЛ

Мы существенно усложнили структуру и логику нашего проекта по сравнению с первым уроком. Пришло время попытаться реализовать запускаемый скрипт и выполнить пробный запуск проекта.

**Листинг 6. step\_2/telbot.py**

|  |
| --- |
| *# импортируем функцию создания объекта бота* **from** telebot **import** TeleBot *# импортируем основные настройки проекта* **from** settings **import** config *# импортируем главный класс-обработчик бота* **from** handlers.handler\_main **import** HandlerMain   **class** TelBot:  *"""  Основной класс телеграмм бота (сервер), в основе которого  используется библиотека pyTelegramBotAPI  """* \_\_version\_\_ = config.VERSION  \_\_author\_\_ = config.AUTHOR   **def** \_\_init\_\_(self):  *"""  Инициализация бота  """  # получаем токен* self.token = config.TOKEN  *# инициализируем бот на основе зарегистрированного токена* self.bot = TeleBot(self.token)  *# инициализируем оброботчик событий* self.handler = HandlerMain(self.bot)   **def** start(self):  *"""  Метод предназначен для старта обработчика событий  """* self.handler.handle()   **def** run\_bot(self):  *"""  Метод запускает основные события сервера  """  # обработчик событий* self.start()  *# служит для запуска бота (работа в режиме нон-стоп)* self.bot.polling(none\_stop=**True**)   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  bot = TelBot()  bot.run\_bot() |

Одним из простейших способов работы с Telegram в Python является использование библиотеки **pyTelegramBotApi**. Для ее установки необходимо выполнить команду:

|  |
| --- |
| pip install pytelegrambotapi |

Разберем некоторые фрагменты представленного листинга.

|  |
| --- |
| self.token = config.TOKENself.bot = TeleBot(self.token) self.handler = HandlerMain(self.bot) |

Из основного файла с настройками получаем значение токена и передаем его в класс-конструктор **TeleBot()** для создания объекта бота. Созданный объект бота мы передаем в наш главный класс обработчик для управления обработчиками его событий (см. листинг 5).

|  |
| --- |
| **def** start(self):  self.handler.handle() |

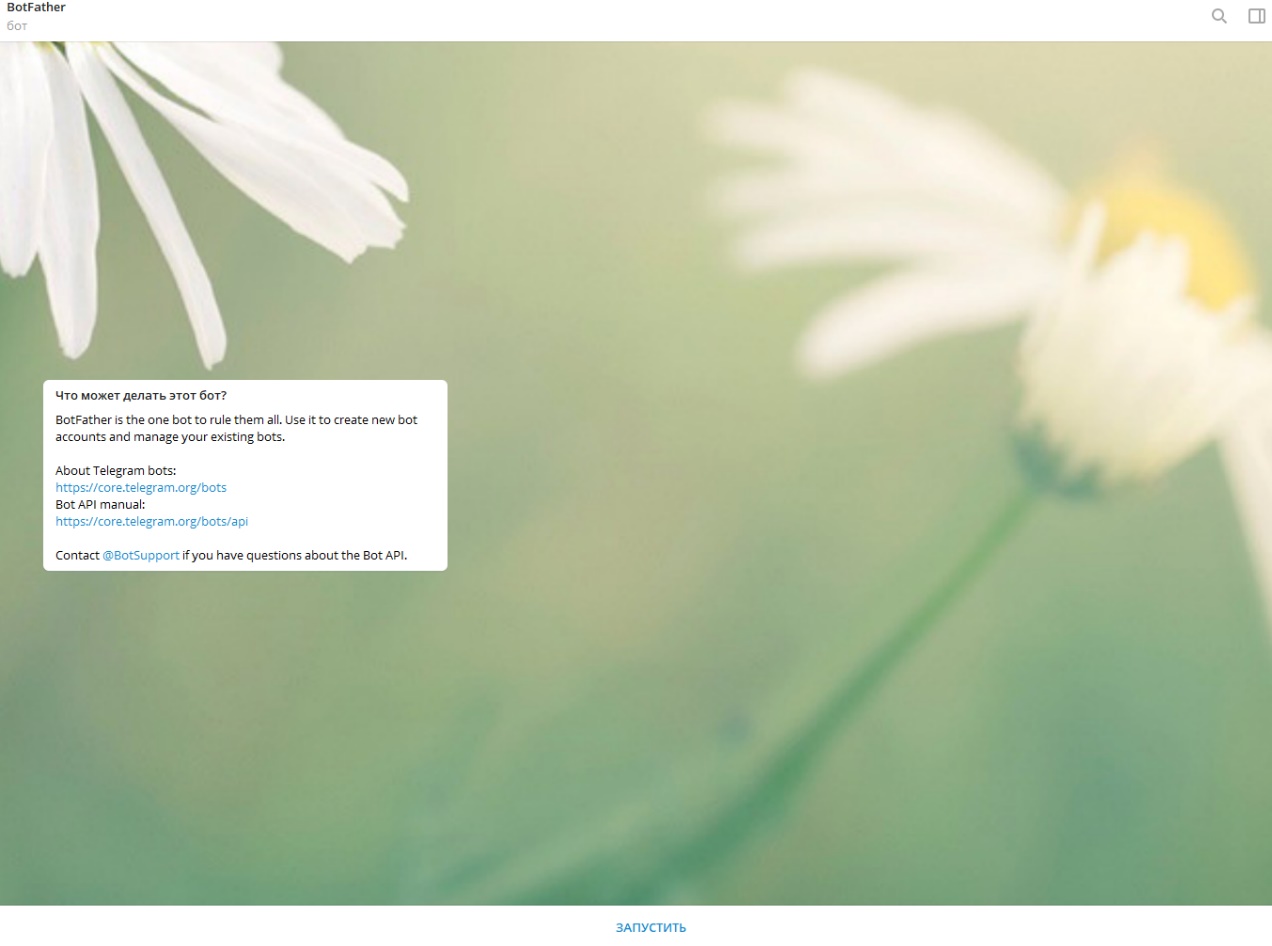
Обеспечиваем возможность запуска обработчиков событий бота.

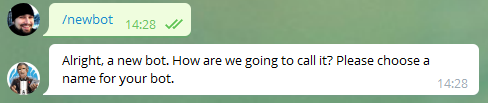
|  |
| --- |
| **def** run\_bot(self):  self.start() self.bot.polling(none\_stop=**True**) |

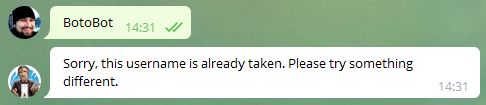
Последняя команда обеспечивает непосредственный запуск бота и его работу в постоянном режиме.

# ВЫПОЛНЯЕМ РЕГИСТРАЦИЮ БОТА

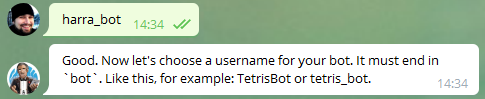
Мы уже несколько раз говорили о токене, но как его получить? Необходимо зарегистрировать наш бот. Для этого перейти в Telegram и найти канал @BotFather, через который и происходит регистрация ботов.



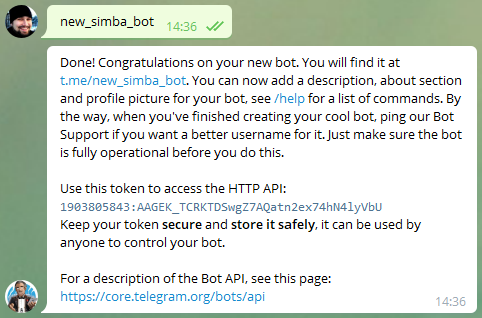
Нажимаем кнопку «Запустить». Теперь нужно ввести команду **/newbot**.

Вам будет предложено ввести имя бота.

Если оно занято, нужно ввести другое имя, но обязательно оканчивающееся на «Bot» или «bot».



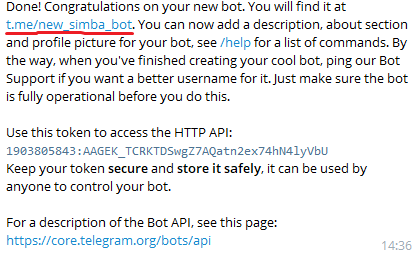
Далее нужно указать никнейм бота. Он также должен быть уникальным.

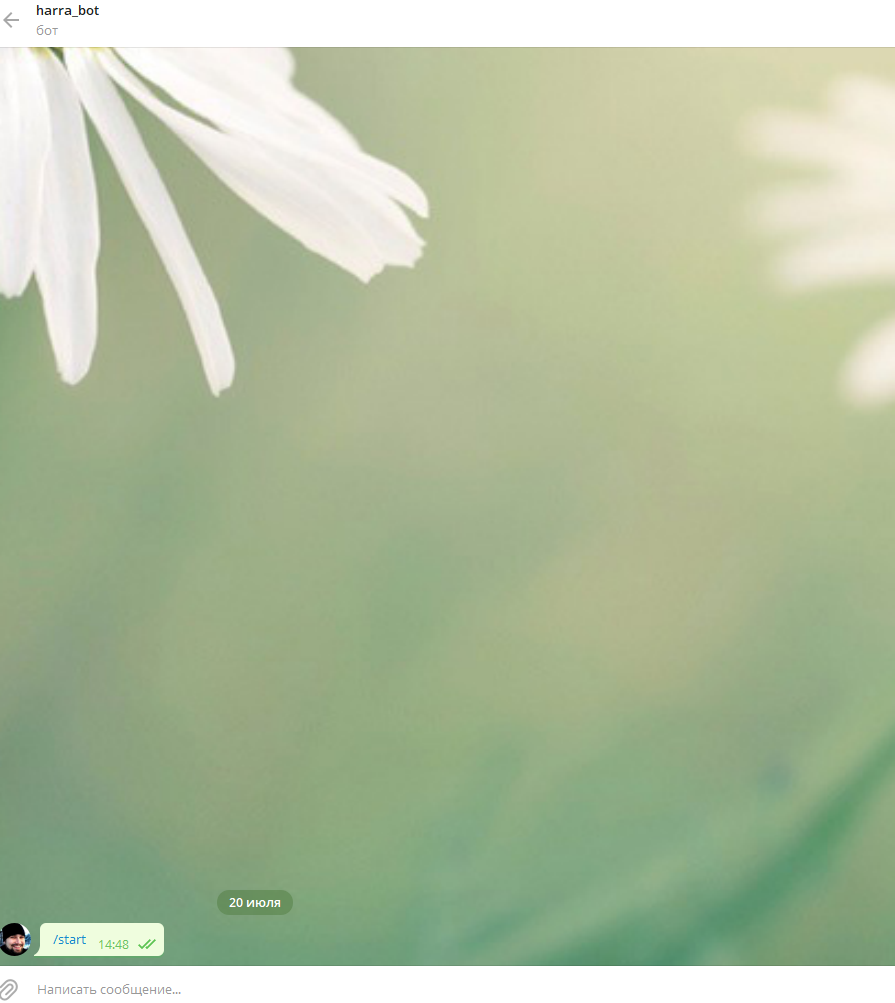


А вот и наш токен. Нужно его скопировать и добавить в наш файл **config.py**.

|  |
| --- |
| TOKEN = **'1903805843:AAGEK\_TCRKTDSwgZ7AQatn2ex74hN4lyVbU'** |

Если ваш токен стал кому-то известен, вам нужно ввести команду **/revoke** для генерации нового токена.

Теперь, чтобы проверить работу нашего бота, необходимо запустить скрипт **telbot.py** и в Telegram перейти по ссылке на ваш бот и нажать кнопку Запустить.

Наш бот запущен и работает. Пока, по сути, в нем нет никакой логики. Мы реализуем ее в следующих уроках.

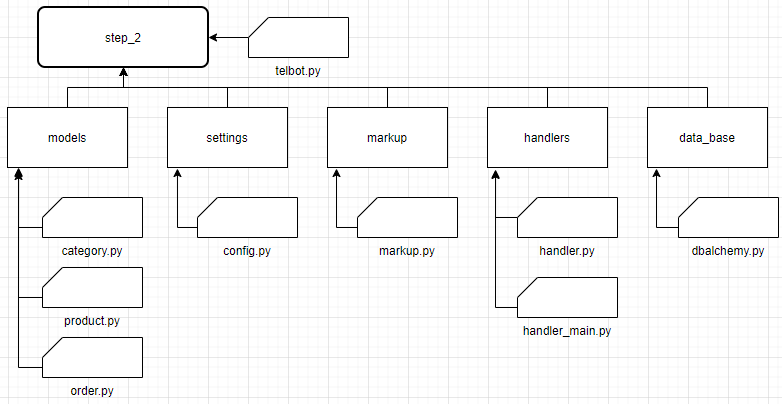
# Вид сверху на рукопожатие над деловыми документамиЗАКЛЮЧЕНИЕ

В рамках второго урока мы провели серьезную работу и подготовили «скелет» проекта. Мы определили базовые настройки проекта в отдельном файле, описали внешний вид элементов интерфейса. Кроме того, мы подготовили базовый класс-обработчик, класс-менеджер для управления базой данных и класс управления обработчиками. Мы создали запускаемый файл и выполнили регистрацию нашего бота.

## КЛЮЧЕВЫЕ ВЫВОДЫ

* Python-приложения «умеют» взаимодействовать с платформой Telegram. При этом применяется библиотека **pyTelegramBotApi**.
* Мы можем создавать для ботов яркие интерфейсы с привлекательным оформлением элементов управления.
* При выполнении операций с ботом мы говорим о событиях, для которых необходимо создавать обработчики.
* Для работы с базой данных мы создали класс-менеджер.
* Важная составляющая написания проекта – создание запускаемого файла и регистрация бота.

Текущая структура разрабатываемого Telegram-бота приведаена на на рис. 1.

Рис. 1. Структура проекта