Дополнительные возможности разработки ботов

Мы изучили основы создания ботов, теперь осталось много доп. возможностей!

Клавиатура в боте

Создание клавиатуры

Для начала давайте познакомимся с такой возможностью, как создание клавиатур.

Принцип простой: что написано на кнопке, то и будет отправлено в текущий чат. Соответственно, чтобы обработать нажатие такой кнопки, бот должен распознавать входящие текстовые сообщения.

Создание клавиатуры по своей сути состоит из 2-ух шагов:

1. Создание кнопок
2. Создание клавиатуры из кнопок

Каждая кнопка создается очень легко:

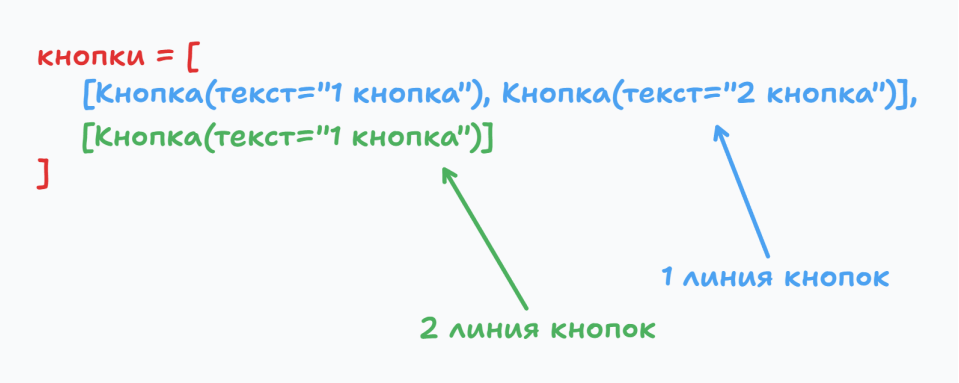
```
from aiogram import types

key1 = types.KeyboardButton(text="Кнопка №1")
key2 = types.KeyboardButton(text="Кнопка №2")
```

Эти строки создают две кнопки с текстом "Кнопка №1" и "Кнопка №2".

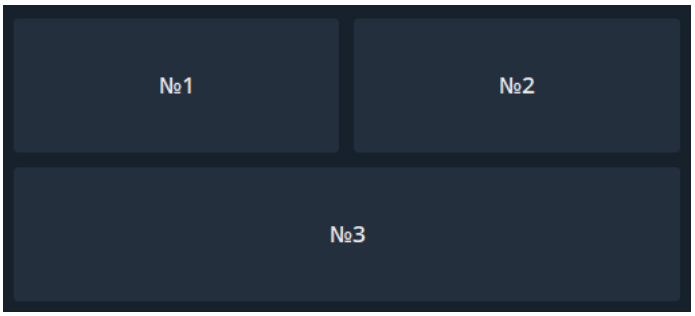
- `types.KeyboardButton` — это класс, который используется для создания кнопок клавиатуры.
- Параметр `text` задаёт текст, который будет отображаться на кнопке.

Итак, переменная `key1` теперь содержит кнопку с надписью "Кнопка №1", а переменная `key2` содержит кнопку с надписью "Кнопка №2".

Для создания клавиатуры нам понадобится создать структуру - список списков. В нашем главном списке каждый вложенный список - это новая линия кнопок. Схематично это выглядит примерно так:



А в телеграмме это выглядит вот так:



С точки зрения кода, нам сначала необходимо создать кнопки, потом из кнопок клавиатуру и отправить её пользователю:

```
@dp.message(Command("start"))
async def start(message: types.Message):
    keys = [
        [types.KeyboardButton(text="Да"), types.KeyboardButton(text="Нет")],
        [types.KeyboardButton(text="А где?")]
    ]
    keyboard = types.ReplyKeyboardMarkup(keyboard=keys)
    await message.answer("Видишь клавиатуру?", reply_markup=keyboard)
```

1. **Создание кнопок для клавиатуры:**

   ```
   keys = [
       [types.KeyboardButton(text="Да"), types.KeyboardButton(text="Нет")],
       [types.KeyboardButton(text="А где?")]
   ]
   ```

   В этой части мы создаём список кнопок.

   - `types.KeyboardButton(text="Да")` и `types.KeyboardButton(text="Нет")` — создаём две кнопки с текстом "Да" и "Нет", которые расположены в одном ряду.
   - `types.KeyboardButton(text="А где?")` — создаём кнопку с текстом "А где?", которая будет расположена в следующем ряду.

2. **Создание объекта клавиатуры:**

   ```
   keyboard = types.ReplyKeyboardMarkup(keyboard=keys)
   ```

   Здесь мы создаём объект клавиатуры `ReplyKeyboardMarkup` и передаём в него список кнопок `keys`, созданный на предыдущем шаге. Это определяет, как кнопки будут организованы на клавиатуре.

3. **Отправка сообщения с клавиатурой:**

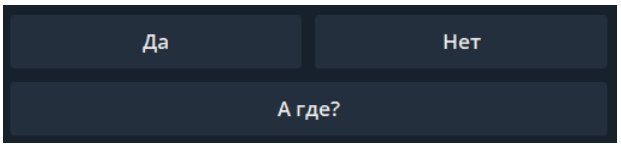
   ```
   await message.answer("Видишь клавиатуру?", reply_markup=keyboard)
   ```

   Эта строка отправляет ответное сообщение пользователю.

   - `message.answer("Видишь клавиатуру?", reply_markup=keyboard)` — отправляет сообщение с текстом "Видишь клавиатуру?" и прикрепляет к нему клавиатуру, созданную ранее.

- `await` используется, чтобы подождать выполнения отправки сообщения, так как функция асинхронная.

Итак, этот код реагирует на команду `/start`, создаёт клавиатуру с кнопками "Да", "Нет" и "А где?", и отправляет её пользователю вместе с сообщением "Видишь клавиатуру?".

> Дополнительно можно уменьшить кнопки. Для этого понадобится добавить лишь один параметр при создании клавиатуры
>
> ```
> keyboard = types.ReplyKeyboardMarkup(keyboard=keys, resize_keyboard=True)
> ```
>
> `resize_keyboard` со значением `True`.
>
> 
>

Для того чтобы убрать клавиатуру необходимо отправить сообщение со специальным типом клавиатуры:

```
await message.reply("Убираем клавиатуру",
reply_markup=types.ReplyKeyboardRemove())
```

Обработка нажатий на клавиатуру

Для обработки нам понадобятся отдельные функции, которые будут реагировать исключительно на текст, который был на кнопках.

И первое что стоит сделать - добавить импорт "магического фильтра":

```
from aiogram import F
```

С помощью него мы и будем фильтровать приходящие сообщения (Больше никаких макарон из `if`, `elif`, `else`)

А далее пишем функцию для каждого текста с каждой кнопки:

```
@dp.message(F.text.lower() == "да")
async def yes(message: types.Message):
    await message.answer("Ну и хорошо!")

@dp.message(F.text.lower() == "нет")
async def yes(message: types.Message):
    await message.answer("А как же ты нажал на эту кнопку?")

@dp.message(F.text.lower() == "а где?")
async def yes(message: types.Message):
    await message.answer("Всё ясно...")
```

Ключевой момент здесь это проверка входящего сообщения:

`F.text.lower() == "да"`: Важно чтобы текст в кавычках был маленькими буквами.

А так это будет выглядеть в телеграмме:

Машина состояний

Иногда бывает такое, что необходимо обрабатывать сообщения пользователя последовательно, и тут на помощь нам приходит машина состояний.

Для работы нам понадобится импортировать некоторые классы:

```
from aiogram.fsm.state import State, StatesGroup
from aiogram.fsm.context import FSMContext
```

Далее мы должны создать целый класс-наследник, в котором и пропишем все возможные состояния нашего телеграмм бота:

```
class MyStates(StatesGroup):
    wait_name = State()
    wait_nickname = State()
```

Этот код создаёт класс `MyStates`, который используется для управления состояниями пользователя в диалоге с ботом. Давайте разберём его построчно:

1. **Объявление класса:**

   ```
   class MyStates(StatesGroup):
   ```

   Здесь мы создаём новый класс `MyStates`, который наследует от `StatesGroup`. `StatesGroup` — это класс из библиотеки aiogram, который помогает организовать и управлять состояниями пользователя в чат-боте.

2. **Определение состояния `wait_name`:**

   ```
   wait_name = State()
   ```

   Это строка объявляет состояние `wait_name`. Состояние `State` используется для обозначения, что бот ожидает от пользователя ввода имени.

3. **Определение состояния `wait_nickname`:**

   ```
   wait_nickname = State()
   ```

Эта строка объявляет состояние `wait_nickname`. Оно указывает, что бот будет ждать ввода никнейма от пользователя.

Таким образом, `MyStates` — это группа состояний, которая включает два состояния:

- `wait_name` — бот ждёт, когда пользователь введёт своё имя.
- `wait_nickname` — бот ждёт, когда пользователь введёт свой никнейм.

Эти состояния можно использовать, чтобы контролировать, на каком этапе взаимодействия находится пользователь, и управлять логикой диалога в зависимости от текущего состояния.

Для работы с состояниями нам понадобится несколько команда:

```
await state.set_state(MyStates.wait_name)
await state.set_data({"name": message.text})
data = await state.get_data()
await state.clear()
```

1. **Установка состояния:**

   ```
   await state.set_state(MyStates.wait_name)
   ```

   Эта строка устанавливает состояние пользователя на `wait_name`. Это означает, что бот теперь ожидает от пользователя ввода имени.

2. **Сохранение данных:**

   ```
   await state.set_data({"name": message.text})
   ```

   Здесь бот сохраняет данные, введённые пользователем. `message.text` содержит текст, который пользователь отправил боту. В данном случае, это имя пользователя, которое бот сохраняет в виде словаря с ключом `"name"`.

3. **Получение данных:**

   ```
   data = await state.get_data()
   ```

   Эта строка получает все данные, которые были сохранены для текущего состояния пользователя. В переменной `data` теперь хранится словарь с данными, например, `{"name": "введённое имя"}`.

4. **Очистка состояния и данных:**

   ```
   await state.clear()
   ```

   Здесь бот очищает текущее состояние и все связанные с ним данные. Это означает, что бот больше не будет ожидать от пользователя ввода имени и все сохранённые данные будут удалены.

Для обработки каждого состояния, нам понадобится соответствующая функция, где будет указано какое состояние она обрабатывает:

```
@dp.message(Command("start"))
async def start(message: types.Message, state: FSMContext):
```

```
    await state.set_state(MyStates.wait_name)
    await message.answer("Введите ваше имя")

@dp.message(MyStates.wait_name)
async def name(message: types.Message, state: FSMContext):
    await state.set_state(MyStates.wait_nickname)
    await state.set_data({"name": message.text})
    await message.answer("Введите ваш никнейм")

@dp.message(MyStates.wait_nickname)
async def nickname(message: types.Message, state: FSMContext):
    data = await state.get_data()
    await message.answer(f"Вас зовут {data['name']}, а ваш никнейм
{message.text}")
    await state.clear()

@dp.message()
async def default(message: types.Message):
    await message.answer("Я не знаю что ответить")
```

1. **Функция `start`:**

   ```
   @dp.message(Command("start"))
   async def start(message: types.Message, state: FSMContext):
       await state.set_state(MyStates.wait_name)
       await message.answer("Введите ваше имя")
   ```

   Эта функция срабатывает, когда пользователь отправляет команду `/start`.

   - Устанавливается состояние `wait_name`, что означает, что бот теперь ждёт ввода имени.
   - Бот отправляет пользователю сообщение "Введите ваше имя".

2. **Функция `name`:**

   ```
   @dp.message(MyStates.wait_name)
   async def name(message: types.Message, state: FSMContext):
       await state.set_state(MyStates.wait_nickname)
       await state.set_data({"name": message.text})
       await message.answer("Введите ваш никнейм")
   ```

   Эта функция срабатывает, когда пользователь ввёл своё имя в состоянии `wait_name`.

   - Устанавливается новое состояние `wait_nickname`, что означает, что бот теперь ждёт ввода никнейма.
   - Сохранение имени пользователя, которое он ввёл.
   - Бот отправляет пользователю сообщение "Введите ваш никнейм".

3. **Функция `nickname`:**

```
@dp.message(MyStates.wait_nickname)
async def nickname(message: types.Message, state: FSMContext):
    data = await state.get_data()
    await message.answer(f"Вас зовут {data['name']}, а ваш никнейм
{message.text}")
    await state.clear()
```

Эта функция срабатывает, когда пользователь ввёл свой никнейм в состоянии `wait_nickname`.

- Получение сохранённых данных (введённого имени).
- Бот отправляет пользователю сообщение вида "Вас зовут {имя}, а ваш никнейм {никнейм}".
- Очистка состояния и данных, связанных с этим пользователем.

4. **Функция `default`:**

```
@dp.message()
async def default(message: types.Message):
    await message.answer("Я не знаю что ответить")
```

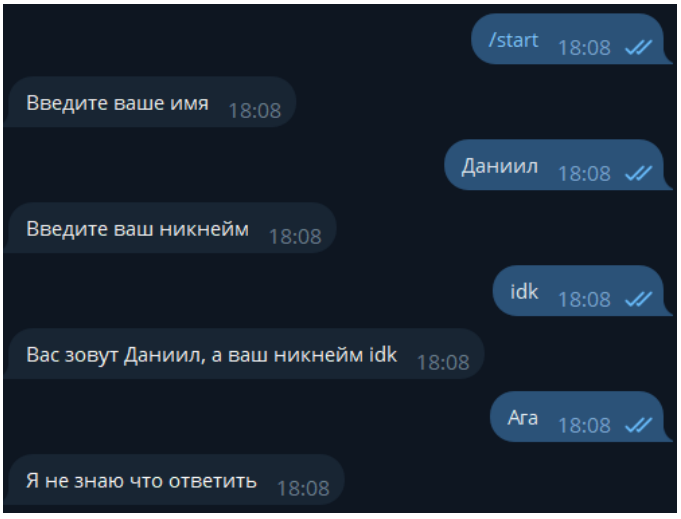
Эта функция срабатывает, если бот получает сообщение, которое не относится к команде `/start` или не соответствует текущему состоянию.

- Бот отвечает пользователю сообщением "Я не знаю что ответить".

Итак, последовательность работы бота выглядит так:

1. Пользователь отправляет команду `/start`.
2. Бот просит ввести имя.
3. Пользователь вводит имя, бот просит ввести никнейм.
4. Пользователь вводит никнейм, бот подтверждает введённые данные (имя и никнейм) и очищает состояние.
5. Если пользователь отправляет сообщение, не связанное с текущим состоянием, бот отвечает "Я не знаю что ответить".

Так это выглядит в телеграмме:

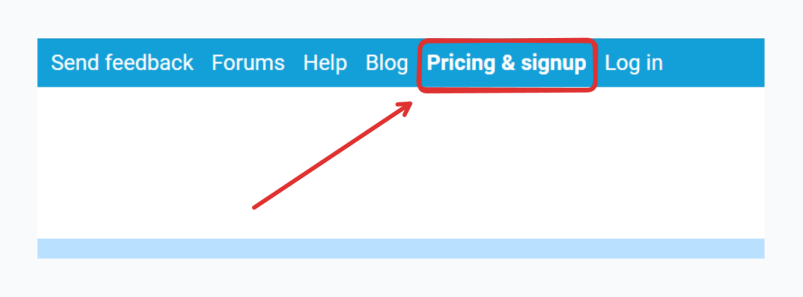


Загрузка и запуск бота на сервере

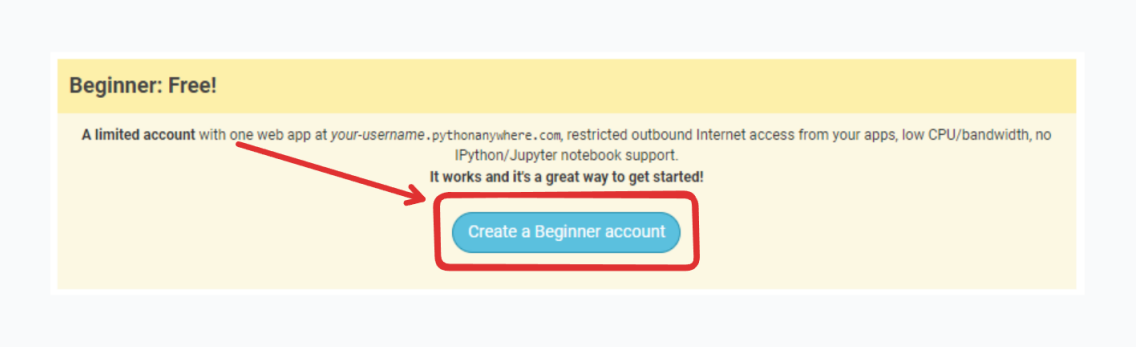
Конечно нам бы хотелось чтобы бот работал не только когда мы его запускаем на своем компьютере, поэтому нам будет нужен какой-то сервер. В большинстве случаем - сервер является платной услугой. Но есть и бесплатные сервер, однако у них есть свои минусы.

Регистрация

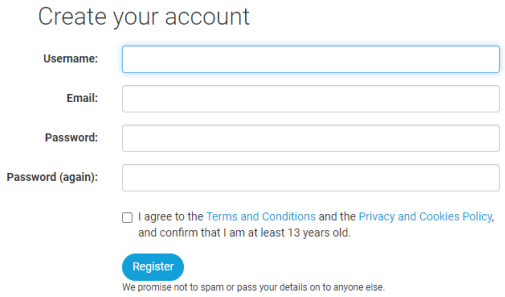
1. Переходим на сайт [Pythonanywhere]
2. В правом верхнем углу нажимаем `Pricing & signup`



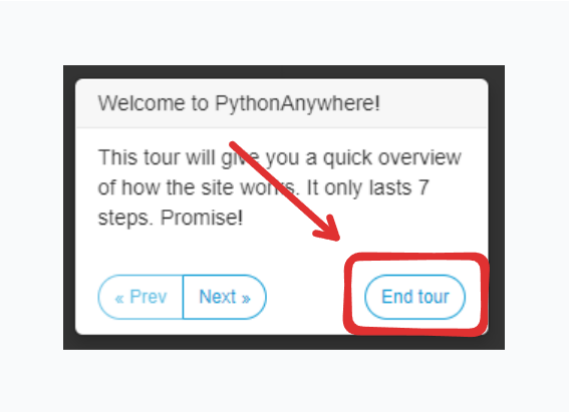
3. Выбираем создание бесплатного аккаунта



4. Вводим логин, почту, пароль и ставим галочку



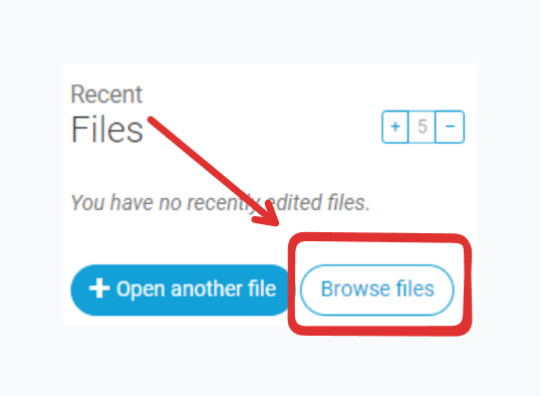
5. Нажимаем кнопку `End tour`



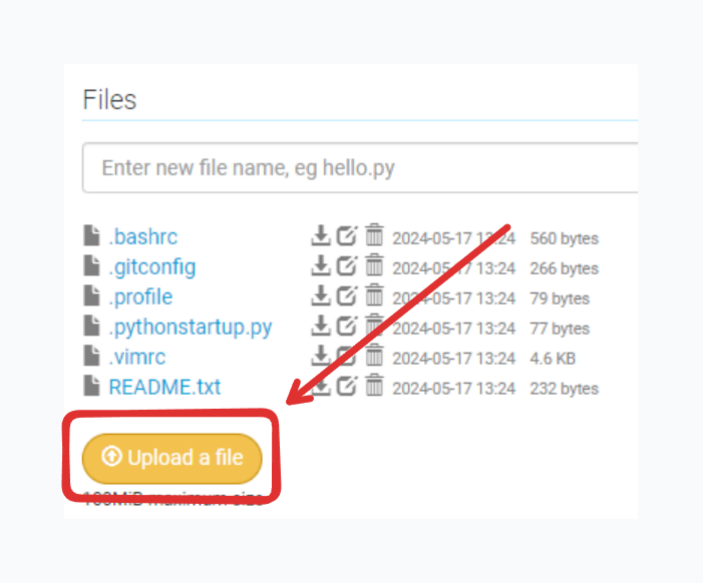
6. Успешно

Загрузка и запуск бота

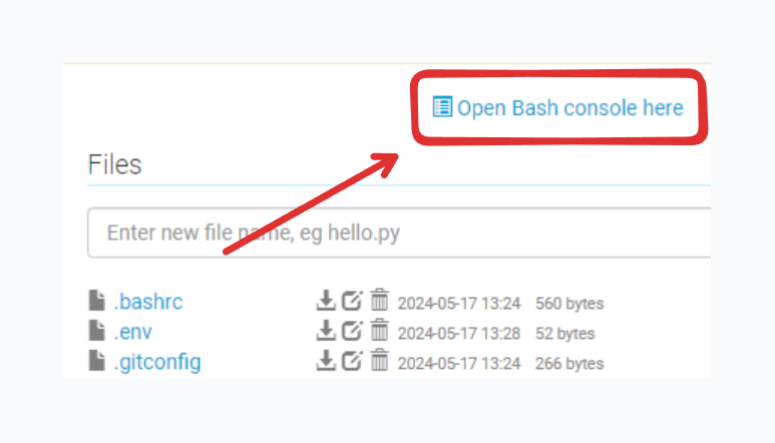
1. Нажимаем кнопку `Browse files`



2. Нажимаем кнопку `Upload a file` и поочередно загружаем все файлы нашего бота, в том числе `.env` с токеном



3. После успешной загрузки всех файлов, нажимаем кнопку `Open Bash console here`

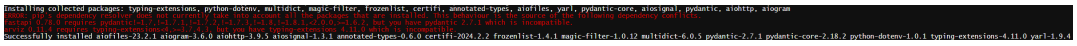


4. Устанавливаем необходимые модули для нашего бота. Вводим в консоли:

```
pip install python-dotenv aiogram
```

и дожидаемся успешной загрузки модулей.

> Если произойдет ошибка:



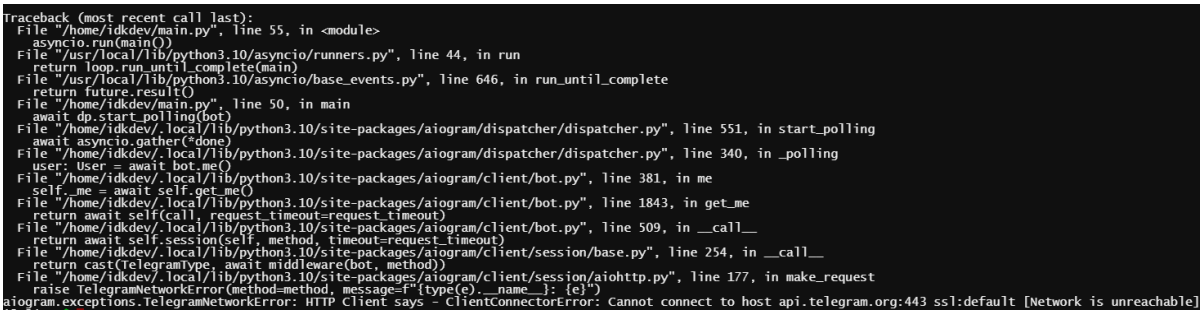
То ничего страшного 👍🙂

5. Осталось запустить бота командой:

```
python main.py
```

❗ Решение проблем с запуском:

Если вдруг при запуске бота возникла ошибка, типа такой:



`[Network is unreachable]`, то вот инструкция для исправления:

1. Для начала нужно установить модуль `aiohttp-socks` в **bash-консоли**

```
pip install aiohttp-socks
```

2. Затем нужно **импортировать** модуль `AiohttpSession`

```
from aiogram.client.session.aiohttp import AiohttpSession
```

3. Теперь нужно создать переменную `session`

```
session = AiohttpSession(proxy='http://proxy.server:3128') # в proxy указан прокси сервер pythonanywhere, он нужен для подключения
```

4. В объекте бота указываем `session=session`

```
bot = Bot(token=TOKEN, session=session)
```

Лайфхаки

Для того, чтобы при отключение бота (нажатии `CTRL + C` в консоли) не вылетали ошибки, запуск следует делать таким, обрабатывая сочетание клавиш:

```
if __name__ == "__main__":
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("Бот остановлен")
```

Кому интересно прочитать про кнопки, которые крепятся к сообщению - Ссылка

Чуть больше про обработку сообщение - Ссылка