

Материалы к занятию

Давайте создадим простого бота(пока ничего не делающего). Вынесем все необходимые переменные в отдельный файл `config.py`, в котором мы их будем подгружать из переменных окружения.

config.py

```
import os

BOT_TOKEN = os.getenv('BOT_TOKEN')
```

bot.py

```
import logging
from aiogram import Bot, Dispatcher, executor, types
from config import BOT_TOKEN

# Объект бота
bot = Bot(token=BOT_TOKEN)
# Диспетчер для бота
dp = Dispatcher(bot)
# Включаем логирование, чтобы не пропустить важные сообщения
logging.basicConfig(level=logging.INFO)

if __name__ == "__main__":
    # Запуск бота
    executor.start_polling(dp, skip_updates=True)
```

В данном случае мы создали заготовку бота, в котором вынесли токен в отдельный файл. Между прочим разбивать программу, на отдельные файлы, является хорошей практикой, так как код становится более читабельным и редактируемым.

Для начала давайте будем перехватывать сообщения с текстом Привет, а в ответ отправлять Привет username. Для этого нам достаточно указать в обработчике, что текст равен строке 'Привет'

```
@dp.message_handler(text='Привет')
async def cmd_test1(message: types.Message):
    await message.reply(f"Привет {message.from_user.username}")
```

В примере мы перехватываем все сообщения, текст которых равен Привет. В переменной `message`, которая приходит нам вместе с сообщением и имеет тип `Message`, хранятся все данные о сообщении и отправителе/чате/группе.

Выводить сообщения можно также используя специальное оформление(html, markdown)

В распоряжении у нас имеется три способа разметки текста: HTML, Markdown и MarkdownV2. Наиболее продвинутыми из них считаются HTML и MarkdownV2, «классический» маркдаун оставлен для обеспечения обратной совместимости и поддерживает меньше возможностей.

За выбор форматирования при отправке сообщений отвечает аргумент `parse_mode`, например:

```
@dp.message_handler(text='Привет')
async def cmd_test1(message: types.Message):
    await message.reply(f"Привет <i>{message.from_user.username}</i>",
                        parse_mode=types.ParseMode.HTML)
```

```
@dp.message_handler(text='Привет')
async def cmd_test1(message: types.Message):
    await message.reply(f"Привет *{message.from_user.username}*",
                        parse_mode=types.ParseMode.MARKDOWN_V2)
```

Если в боте повсеместно используется определённое форматирование, то каждый раз указывать аргумент `parse_mode` довольно накладно. К счастью, в `aiogram` можно передать необходимый тип прямо в объект `Bot`, а если в каком-то конкретном случае нужно обойтись без разметок, то просто укажите `parse_mode=""` (пустая строка):

```
bot = Bot(token=BOT_TOKEN, parse_mode=types.ParseMode.HTML)

#с использованием разметки
@dp.message_handler(text='Привет')
async def cmd_test1(message: types.Message):
    await message.reply(f"Привет <b>{message.from_user.username}</b>")
```

```

bot = Bot(token=BOT_TOKEN, parse_mode=types.ParseMode.HTML)

#без использования разметки
@dp.message_handler(text='Привет')
async def cmd_test1(message: types.Message):
    await message.reply(f"Привет <b>{message.from_user.username}</b>",
parse_mode='')

```

Существует и более «программный» или даже «динамический» способ формирования сообщения. Для этого нужно импортировать модуль `markdown` из `aiogram.utils`, который, несмотря на название, поддерживает и HTML тоже. Далее вызовите функцию `text()`, в которую передайте произвольное число таких же вызовов функции `text()`. Тип форматирования определяется названием функции, а буква "h" в начале означает HTML, т.е. функция `hbold()` обрамляет переданный ей текст как жирный в HTML-разметке (`текст`). Аргумент `sep` определяет разделитель между кусками текста.

```

import aiogram.utils.markdown as fmt

@dp.message_handler(text='Привет')
async def cmd_test1(message: types.Message):
    await message.answer(
        fmt.text(
            fmt.text(fmt.hunderline("Яблоки"), ", вес 1 кг."),
            fmt.text("Старая цена:", fmt.hstrikethrough(50), "рублей"),
            fmt.text("Новая цена:", fmt.hbold(25), "рублей"),
            sep="\n"
        ), parse_mode="HTML"
    )

```

Помимо отправки с форматированием, Aiogram позволяет извлекать входящий текст как простое содержимое (plain text), как HTML и как Markdown. Сравнить можно на скриншоте ниже. Это удобно использовать, например, если вы хотите вернуть отправителю его сообщение с небольшими изменениями:

```

@dp.message_handler(text='Привет')
async def any_text_message(message: types.Message):
    await message.answer(message.text)
    await message.answer(message.md_text)
    await message.answer(message.html_text)
    # Дополняем исходный текст:
    await message.answer(
        f"<u>Ваш текст</u>:\n\n{message.html_text}", parse_mode="HTML"
    )

```

С использованием форматирования есть проблема: не в меру хитрые пользователи могут использовать спец. символы в именах или сообщениях, ломая бота. Впрочем, в aiogram существуют методы экранирования таких символов: `escape_md()` и `quote_html()`. Либо можно использовать упомянутые выше методы `(h)bold`, `(h)italic` и прочие:

```
@dp.message_handler(text='Привет')
async def any_text_message2(message: types.Message):
    await message.answer(f"Привет, <b>{fmt.quote_html(message.text)}</b>",
        parse_mode=types.ParseMode.HTML)
    # А можно и так:
    await message.answer(fmt.text("Привет, ", fmt.hbold(message.text)),
        parse_mode=types.ParseMode.HTML)
```

Помимо обычных текстовых сообщений Telegram позволяет обмениваться медиафайлами различных типов: фото, видео, гифки, геолокации, стикеры и т.д. У большинства медиафайлов есть свойства `file_id` и `file_unique_id`. Первый можно использовать для повторной отправки одного и того же медиафайла много раз, причём отправка будет мгновенной, т.к. сам файл уже лежит на серверах Telegram. Это самый предпочтительный способ.

К примеру, следующий код заставит бота моментально ответить пользователю той же гифкой, что была прислана:

```
@dp.message_handler(content_types=[types.ContentType.ANIMATION])
async def echo_document(message: types.Message):
    await message.reply_animation(message.animation.file_id)
```

`file_id` уникален для каждого бота, т.е. переиспользовать чужой идентификатор нельзя. Однако в Bot API есть ещё `file_unique_id`. Его нельзя использовать для повторной отправки или скачивания медиафайла, но зато он одинаковый у всех ботов. Нужен `file_unique_id` обычно тогда, когда нескольким ботам требуется знать, что их собственные `file_id` относятся к одному и тому же файлу.

Кстати, про скачивание: aiogram предлагает удобный вспомогательный метод `download()` для загрузки небольших файлов на сервер, где запущен бот:

```

@dp.message_handler(content_types=[types.ContentType.DOCUMENT])
async def download_doc(message: types.Message):
    # Скачивание в каталог с ботом с созданием подкаталогов по типу файла
    await message.document.download()

# Типы содержимого тоже можно указывать по-разному.
@dp.message_handler(content_types=["photo"])
async def download_photo(message: types.Message):
    # Убедитесь, что каталог /tmp/somedir существует!
    await message.photo[-1].download(destination="/tmp/somedir/")

```

С выводом текста в сообщениях мы разобрались. Но как работают в целом хендлеры? Ведь обработка сообщений - одно из самых важных, если не самое важное.

Вернемся к нашему шаблону

```

import logging
from aiogram import Bot, Dispatcher, executor, types
import aiogram.utils.markdown as fmt
from config import BOT_TOKEN

# Объект бота
bot = Bot(token=BOT_TOKEN, parse_mode=types.ParseMode.HTML)
# Диспетчер для бота
dp = Dispatcher(bot)
# Включаем логирование, чтобы не пропустить важные сообщения
logging.basicConfig(level=logging.INFO)

@dp.message_handler(CommandStart())
async def bot_start(message: types.Message):
    await message.answer(f"Привет, {message.from_user.full_name}")

if __name__ == "__main__":
    # Запуск бота
    executor.start_polling(dp, skip_updates=True)

```

Что же тут делает декоратор message_handler?

Открываем источник нажав на декораторе с зажатой клавишей CTRL и видим такое:

```

486 | self.message_handlers.register(self._wrap_async_task(callback, run_task), filters_set)
487 |
488 | def message_handler(self, *custom_filters, commands=None, regexp=None, content_types=None, state=None,
489 |                     run_task=None, **kwargs):
490 |     """
491 |     Decorator for message handler
492 |
493 |     Examples:
494 |
495 |     Simple commands handler:
496 |
497 |     .. code-block:: python3
498 |
499 |         @dp.message_handler(commands=['start', 'welcome', 'about'])
500 |         async def cmd_handler(message: types.Message):
501 |
502 |     Filter messages by regular expression:
503 |
504 |     .. code-block:: python3
505 |
506 |         @dp.message_handler(regexp='^[a-z]*[0-9]*')
507 |         async def msg_handler(message: types.Message):
508 |

```

```

555 | :param run_task: run callback in task (no wait results)
556 | :return: decorated function
557 | """
558 |
559 | def decorator(callback):
560 |     self.register_message_handler(callback, *custom_filters,
561 |                                  commands=commands, regexp=regexp, content_types=content_types,
562 |                                  state=state, run_task=run_task, **kwargs)
563 |     return callback
564 |
565 | return decorator
566 |
567 | def register_edited_message_handler(self, callback, *custom_filters, commands=None, regexp=None, content_types=None,
568 |                                   state=None, run_task=None, **kwargs):
569 |     """
570 |     Register handler for edited message
571 |
572 |     :param callback:
573 |     :param commands: list of commands
574 |     :param regexp: REGEXP
575 |     :param content_types: list of content types.
576 |     :param state:
577 |     :param custom_filters: list of custom filters
578 |     :param run_task: run callback in task (no wait results)

```

Dispatcher > message_handler()

Декоратор вызывает функцию `register_message_handler` и передает в нее все условия (фильтры), что мы указали и `callback` (функцию, в которой мы хотим обрабатывать наше сообщение).

А что же происходит в функции `register_message_handler`?

```

574 | :param custom_filters: list of custom filters
575 | :param kwargs:
576 | :param state:
577 | :return: decorated function
578 | """
579 |
580 | filters_set = self.filters_factory.resolve(self.message_handlers,
581 |                                           *custom_filters,
582 |                                           commands=commands,
583 |                                           regexp=regexp,
584 |                                           content_types=content_types,
585 |                                           state=state,
586 |                                           **kwargs)
587 |
588 | self.message_handlers.register(self._wrap_async_task(callback, run_task), filters_set)
589 |
590 | def message_handler(self, *custom_filters, commands=None, regexp=None, content_types=None, state=None,
591 |                     run_task=None, **kwargs):
592 |     """
593 |     Decorator for message handler
594 |
595 |     Examples:
596 |
597 |     Simple commands handler:
598 |
599 |     .. code-block:: python3

```

Dispatcher > register_message_handler()

Тут регистрируются наши условия (фильтры) и выполняется метод `register` у объекта `message_handlers`. То есть, если мы возьмем вместо декоратора просто функцию `dp.register_message_handler` - будет тот же результат, что и декоратором.

Давайте разберемся, что же это за функция register?

```
40 class Handler:
41     def __init__(self, dispatcher, once=True, middleware_key=None):
42         self.dispatcher = dispatcher
43         self.once = once
44
45         self.handlers: List[Handler.HandlerObj] = []
46         self.middleware_key = middleware_key
47
48     def register(self, handler, filters=None, index=None):
49         """
50         Register callback
51
52         Filters can be awaitable or not.
53
54         :param handler: coroutine
55         :param filters: list of filters
56         :param index: you can reorder handlers
57         """
58         from .filters import get_filters_spec
59
60         spec = _get_spec(handler)
61
62         if filters and not isinstance(filters, (list, tuple, set)):
63             filters = [filters]
```

```
64         :param filters: list of filters
65         :param index: you can reorder handlers
66         """
67         from .filters import get_filters_spec
68
69         spec = _get_spec(handler)
70
71         if filters and not isinstance(filters, (list, tuple, set)):
72             filters = [filters]
73         filters = get_filters_spec(self.dispatcher, filters)
74
75         record = Handler.HandlerObj(handler=handler, spec=spec, filters=filters)
76         if index is None:
77             self.handlers.append(record)
78         else:
79             self.handlers.insert(index, record)
80
81     def unregister(self, handler):
82         """
83         Remove handler
84
85         :param handler: callback
86         :return:
87         """
```

Есть некий класс Handler, у которого есть атрибут-список хендлеров, называется он `self.handlers`

И выполняя функцию `register`, мы добавляем наш `HandlerObj`, в котором будет нужная нам функция с нужными фильтрами, в список хендлеров этого типа (например `message` или `callback_query`).

Если посмотреть в объект `Dispatcher`, то увидим следующее:

```

37 class Dispatcher(DataMixin, ContextInstanceMixin):
38     """
39     Simple Updates dispatcher
40
41     It will process incoming updates: messages, edited messages, channel posts, edited channel posts,
42     inline queries, chosen inline results, callback queries, shipping queries, pre-checkout queries.
43     """
44
45     def __init__(self, bot, loop=None, storage: typing.Optional[BaseStorage] = None,
46                 run_tasks_by_default: bool = False,
47                 throttling_rate_limit=DEFAULT_RATE_LIMIT, no_throttle_error=False,
48                 filters_factory=None):
49
50         if not isinstance(bot, Bot):
51             raise TypeError(f"Argument 'bot' must be an instance of Bot, not '{type(bot).__name__}'")
52
53         if storage is None:
54             storage = DisabledStorage()
55         if filters_factory is None:
56             filters_factory = FiltersFactory(self)
57
58         self.bot: Bot = bot
59         self._main_loop = loop
60         self.storage = storage
61         self.run_tasks_by_default = run_tasks_by_default
62
63         self.throttling_rate_limit = throttling_rate_limit
64         self.no_throttle_error = no_throttle_error
65
66         self.filters_factory: FiltersFactory = filters_factory
67         self.updates_handler = Handler(self, middleware_key='update')
68         self.message_handlers = Handler(self, middleware_key='message')
69         self.edited_message_handlers = Handler(self, middleware_key='edited_message')
70         self.channel_post_handlers = Handler(self, middleware_key='channel_post')
71         self.edited_channel_post_handlers = Handler(self, middleware_key='edited_channel_post')
72         self.inline_query_handlers = Handler(self, middleware_key='inline_query')
73         self.chosen_inline_result_handlers = Handler(self, middleware_key='chosen_inline_result')
74         self.callback_query_handlers = Handler(self, middleware_key='callback_query')
75         self.shipping_query_handlers = Handler(self, middleware_key='shipping_query')

```

Dispatcher > __init__()

У диспатчера есть набор атрибутов-хендлеров. Это и `message_handlers`, и `edited_message_handlers` и `channel_post_handlers` и другие. Это те самые объекты типа `Handler`, у которого есть список `self.handlers`, в который мы добавляем наши обработчики. Теперь, когда нам понятно как регистрируются наши хендлеры, давайте глянем, что происходит, когда приходит апдейт.


```
243     """
244     Process single update object
245
246     :param update:
247     :return:
248     """
249     types.Update.set_current(update)
250
251     try:
252         if update.message:
253             types.Message.set_current(update.message)
254             types.User.set_current(update.message.from_user)
255             types.Chat.set_current(update.message.chat)
256             return await self.message_handlers.notify(update.message)
257         if update.edited_message:
258             types.Message.set_current(update.edited_message)
259             types.User.set_current(update.edited_message.from_user)
260             types.Chat.set_current(update.edited_message.chat)
261             return await self.edited_message_handlers.notify(update.edited_message)
262         if update.channel_post:
263             types.Message.set_current(update.channel_post)
264             types.Chat.set_current(update.channel_post.chat)
265             return await self.channel_post_handlers.notify(update.channel_post)
266         if update.edited_channel_post:
267             types.Message.set_current(update.edited_channel_post)
268             types.Chat.set_current(update.edited_channel_post.chat)
269             return await self.edited_channel_post_handlers.notify(update.edited_channel_post)
270         if update.inline_query:
271             types.InlineQuery.set_current(update.inline_query)
272             types.User.set_current(update.inline_query.from_user)
273             return await self.inline_query_handlers.notify(update.inline_query)
274         if update.chosen_inline_result:
275             types.ChosenInlineResult.set_current(update.chosen_inline_result)
276             types.User.set_current(update.chosen_inline_result.from_user)
277             return await self.chosen_inline_result_handlers.notify(update.chosen_inline_result)
278         if update.callback_query:
279             types.CallbackQuery.set_current(update.callback_query)
280             if update.callback_query.message:
```

У Dispatcher есть метод `process_update`, который проверяет тип апдейта, и выполняет функцию `notify` у нужного типа хендлеров.

Заходим в метод `notify` и видим:

```
84     async def notify(self, *args):
85         """
86         Notify handlers
87
88         :param args:
89         :return:
90         """
91         from .filters import check_filters, FilterNotPassed
92
93         results = []
94
95         data = {}
96         ctx_data.set(data)
97
98         if self.middleware_key:
99             try:
100                 await self.dispatcher.middleware.trigger(f"pre_process_{self.middleware_key}", args + (data,))
101             except CancelHandler: # Allow to cancel current event
102                 return results
103
104         try:
105             for handler_obj in self.handlers:
106                 try:
107                     data.update(await check_filters(handler_obj.filters, args))
108                 except FilterNotPassed:
109                     continue
110                 else:
111                     ctx_token = current_handler.set(handler_obj.handler)
112                     try:
113                         if self.middleware_key:
114                             await self.dispatcher.middleware.trigger(f"process_{self.middleware_key}", args + (data,))
115                         partial_data = _check_spec(handler_obj.spec, data)
116                         response = await handler_obj.handler(*args, **partial_data)
117                         if response is not None:
118                             results.append(response)
119                         if self.once:
120                             break
121                     except SkipHandler:
122                         continue
```

1. Если на типа хендлеров установлен ключ миддлваря, то триггерится функция `pre_process_` у необходимого типа апдейтов.

2. Происходит итерация по ЗАРЕГИСТРИРОВАННЫМ хендлерам этого типа (например message). Тот самый список, в который делали append в объекте Handler.
3. Идет проверка на фильтры в каждом хендлере. Если все условия не выполняются - поднимается ошибка FilterNotPassed, и в этой строке она обрабатывается и происходит переход к следующему хендлеру.
4. Теперь, когда ошибка не случилась- фильтры подошли, триггерится уже миддлварь ключа process_
5. После чего выполняется та функция, которую мы зарегистрировали с нужными параметрами (переданными, как самим аиограмом, так и через миддлвари).

Давайте разберем каким образом мы можем работать с сообщениями, а конкретнее со встроенными фильтрами. Используя aiogram вы получаете множество возможностей, которых нет в других библиотеках, как например встроенные фильтры. Вместо того, чтобы писать func=lambda message: message.text=="Текст", как в pyTelegramBotApi, в aiogram вы пропишете просто text="Текст". Но это далеко не все. При этом встроенные фильтры постоянно добавляются.

Давайте посмотрим какие же у нас присутствуют фильтры.

Составим небольшой список всех фильтров

Command — проверка сообщения на команду

CommandHelp — проверка на команду /help

CommandPrivacy — проверка на команду /privacy

CommandSettings — проверка на команду /settings

CommandStart — проверка на команду /start

ContentTypeFilter — проверка типа контента

ExceptionsFilter — исключение для errors_handler

Hashtag — обработка сообщений с #hashtag и \$cashtags

Regex — регулярное выражение для сообщений callback query

RegexCommandsFilter — проверка команды регулярным выражением

StateFilter — проверка состояния пользователя

Text — фильтр текста. Работает на большинстве обработчиков

IDFilter — фильтр для проверки id чата или пользователя

AdminFilter — проверка на то, является ли пользователь администратором чата

IsReplyFilter — проверяет, что отправленное сообщение является ответом

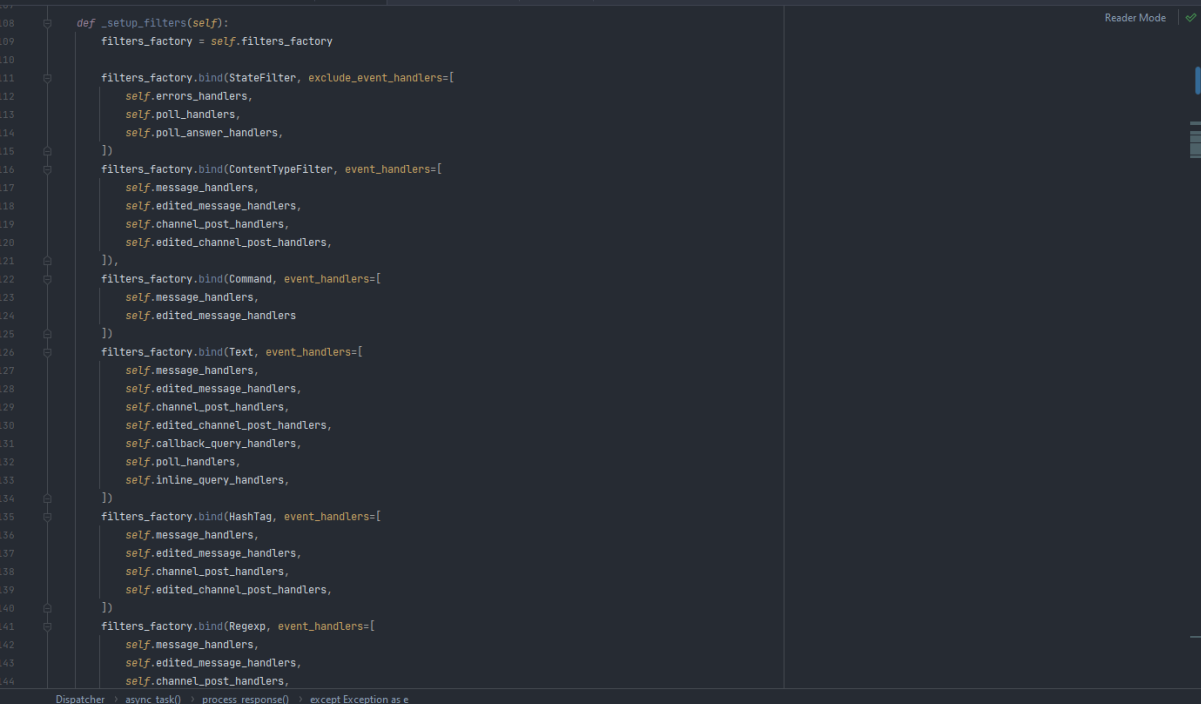
IsSenderContact — проверяет, что пользователь отправил именно свой контакт

ForwardedMessageFilter — проверка на то, что сообщение переслано

ChatTypeFilter — проверка типа чата

Все стандартные фильтры подключаются на Dispatcher при его инициализации, при помощи приватного метода `_setup_filters`. Здесь мы можем сразу увидеть к каким из обработчиков применяются фильтры. Например первый подключаемый фильтр — *StateFilter*, который вы используете для работы с машиной состояний, подключается всем обработчикам кроме `errors_handlers`, `poll_handlers` и `poll_answer_handlers`, а

ContentTypeFilter исключительно к сообщениям, которые содержат в update тип Message (т.к. только в нем есть content_type). Это следует из следующих строк:



```
def _setup_filters(self):
    filters_factory = self.filters_factory

    filters_factory.bind(StateFilter, exclude_event_handlers=[
        self.errors_handlers,
        self.poll_handlers,
        self.poll_answer_handlers,
    ])
    filters_factory.bind(ContentTypeFilter, event_handlers=[
        self.message_handlers,
        self.edited_message_handlers,
        self.channel_post_handlers,
        self.edited_channel_post_handlers,
    ])
    filters_factory.bind(Command, event_handlers=[
        self.message_handlers,
        self.edited_message_handlers,
    ])
    filters_factory.bind(Text, event_handlers=[
        self.message_handlers,
        self.edited_message_handlers,
        self.channel_post_handlers,
        self.edited_channel_post_handlers,
        self.callback_query_handlers,
        self.poll_handlers,
        self.inline_query_handlers,
    ])
    filters_factory.bind(HashTag, event_handlers=[
        self.message_handlers,
        self.edited_message_handlers,
        self.channel_post_handlers,
        self.edited_channel_post_handlers,
    ])
    filters_factory.bind(Regexp, event_handlers=[
        self.message_handlers,
        self.edited_message_handlers,
        self.channel_post_handlers,
    ])
```

Все command фильтры применяются исключительно на обработчики message_handler и edited_message_handler.

Может использоваться как аргумент обработчика commands. Т.е. вы можете использовать данный фильтр двумя способами:

```
@dp.message_handler(commands='myCommand', commands_ignore_caption=False)
```

```
from aiogram.dispatcher.filters import Command

@dp.message_handler(Command('myCommand', ignore_caption=False))
```

Наверное самый часто используемый фильтр — Command. Кроме аргумента commands он принимает prefixes, ignore_case, ignore_mention, ignore_caption. *prefixes* — префиксы команды, т.е. то, с чего команда начинается. Самыми частыми префиксами являются стандартный "/", а также "!". *ignore_case* — игнорировать регистр команды. Проверяется с помощью str.lower(). *ignore_mention* — игнорировать упоминание бота. По умолчанию False. Таким образом, когда бот получает команду с mention другого бота, он её не обрабатывает. Если

передать True, в независимости от mention в /command@mention команда попадёт в обработчик, даже если это команда с упоминанием другого бота. Помните, что бот с включенным privacy mode не получит команду с упоминанием другого бота.

ignore_caption — игнорировать команды, которые написаны под изображением. По умолчанию True.

CommandStart - Этот фильтр, для проверки команды /start. Это фильтр Command, в который передаётся команда 'start', а остальные аргументы остаются по умолчанию. Фильтр CommandStart принимает лишь два аргумента:

deep_link — строка или регулярное выражение, для обработки deep_link.

encoded — обрабатывать закодированную ссылку deep_link (по умолчанию False).

```
from aiogram import types
from aiogram.dispatcher.filters import CommandStart

@dp.message_handler(filters.CommandStart(deep_link='deep_link'))
async def deep_link(msg: types.Message):
    await msg.answer('Да, знаем мы такое')

@dp.message_handler(filters.CommandStart())
async def command_start_handler(msg: types.Message):
    await msg.answer('Привет!')
```

Кроме обычного ответа на команду /start, этот фильтр также обрабатывает deep_link, для обработки deep_link. Мы ловим с его помощью ссылки с говорящим аргументом 'deep_link' (т.е ссылок вида https://t.me/{bot.username}?start=deep_link).

Фильтры CommandHelp, CommandPrivacy и CommandSettings не представляют из себя ничего особо интересного. Это просто фильтр Command с переданной в него командой 'help', 'privacy' или 'settings' соответственно. Особенность этих команд заключается в том, что это глобальные команды, наличие которые добавляет дополнительные кнопки в профиле бота.

ContentTypeFilter - Этот фильтр проверяет тип контента, будь то фото, текст или что-нибудь другое. Должен использоваться исключительно как аргумент content_types. По умолчанию проверяет на текст. Принимает либо строку, либо aiogram.types.ContentTypes (что строкой и является).

```
from aiogram import types
from aiogram.dispatcher import filters
```

```
from app.loader import dp

@dp.message_handler(content_types='photo')
@dp.message_handler(content_types=types.ContentTypes.PHOTO)
async def content_type_example(msg: types.Message):
    await msg.answer('Красиво 😊')
```

ExceptionsFilter - Фильтр, используемый в error_handlers. Принимает исключение.
Работает исключительно в качестве аргумента по ключевому слову exception

```
from aiogram import types
from aiogram.utils import exceptions
from loguru import logger

from app.loader import dp

@dp.errors_handler(exception=exceptions.BotBlocked)
async def bot_blocked_error(update: types.Update, exception:
exceptions.BotBlocked):
    logger.exception(f'Bot blocked by user {update.message.from_user.id}')
    return True
```

Данный обработчик сработает, когда бот словит исключение BotBlocked. Здесь можно, например, удалять пользователя из базы данных, чтобы во время следующей рассылки не тратить время на данного пользователя.