Вспомнить и научиться

> *Данный бот был написан с целью обучить команду и подготовить к написанию проекта, чат бота на Aiogram, для внутреннего пользования сотрудниками компании X.*

Aiogram

Начало

```
import asyncio
import os

import dotenv

from aiogram import Bot, Dispatcher, types
from aiogram.filters import CommandStart


dotenv.load_dotenv()

TOKEN = os.getenv('BOT_TOKEN')

# Класс самого бота - инициализация.
bot = Bot(token=TOKEN)

# Обрабатывает все апдейты из сервера - всё что касается бота.
# Отвечает за фильтрацию сообщений полученных с сервера.
# PS:
# В предыдущей версии нужно было передать объект бота.
dispatcher = Dispatcher()


# С помощью готовой системы фильтрации хендлер
# будет реагировать на /start нужным образом.
@dispatcher.message(CommandStart())
async def start_cmd(message: types.Message):
    await message.answer('Отвечаю на сообщение /start')

# ВАЖНА ПОСЛЕДОВАТЕЛЬНОСТЬ ФИЛЬТРАЦИИ СОБЫТИЙ!
# Если echo разместить перед start_cmd, то start_cmd никогда не выполнится!!!

# Хэндлер для обработки любого текстового сообщения (так как в скобках декоратора ничего не заданно),
# что бы бот отреагировал на него.
# Декоратор с типом события message - сообщение от пользователя.
@dispatcher.message()
async def echo(message: types.Message):
    await message.answer(message.text)
```

```
    # Что-бы ответить с упоминанием автора.
    await message.reply(message.text)


async def main():
    # Здесь бот начнет слушать сервер ТГ, и спрашивать у него о наличии обновлений.
    await dispatcher.start_polling(bot)


if __name__ == '__main__':
    asyncio.run(main())
```

При запуске бота пропустить (skip) update, когда бот не был в онлайне

Это нужно для того, что-бы бот не начал отвечать на 100 пропущенных сообщений

```
async def main():
    # Перед запуском сбрасываем старые обновления и начнем пуллинг с новых.
    await bot.delete_webhook(drop_pending_updates=True)

    # Здесь бот начнет слушать сервер ТГ, и спрашивать у него о наличии обновлений.
    await dispatcher.start_polling(bot)


if __name__ == '__main__':
    asyncio.run(main())
```

Allowed updates: какие обновления принимать (message, картинки ..)

Было бы здорово прописать константу со списком разрешенных обновлений:

```
...
ALLOWED_UPDATES = ['message', 'edited_message']

...

async def main():

    await bot.delete_webhook(drop_pending_updates=True)

    # Новый параметр allowed_updates
    await dispatcher.start_polling(bot, allowed_updates=ALLOWED_UPDATES)


if __name__ == '__main__':
    asyncio.run(main())
```

Фильтрация событий. Разница между `диспетчерами` и `роутерами`

Диспетчер - более расширенная версия роутера. Можно воспринимать диспетчер как главного, а роутеры (хендлеры) подключать к главному диспетчеру.

> *[!attention] Итак, может быть куча роутеров, разбросанных по модулям, и ВСЕ роутеры можно подключить к одному диспетчеру.*

Например

Файл *user_handlers.py*

```
from aiogram import types, Router
from aiogram.filters import CommandStart

# Иницыализируем роутер
user_private_router = Router()

@user_private_router.message(CommandStart())
async def start_cmd(message: types.Message):
    await message.answer('Отвечаю на сообщение /start')

@user_private_router.message()
async def echo(message: types.Message):
    await message.answer(message.text)
```

Файл *main.py*

```
...

from user_handlers import user_private_router


bot = Bot(token=TOKEN)
dispatcher = Dispatcher()

# Подключаем роутер к диспетчеру.
dispatcher.include_router(user_private_router)
...
```

Фильтрация разных команд: класс (фильтр) `Command`

Слэш добавится автоматически!

```
from aiogram.filters import Command
...

@user_private_router.message(Command('menu'))
async def menu_cmd(message: types.Message):
    await message.answer('Наше меню')
...
```

Реализация кнопки "меню" программно

Как бот будет стартовать он будет отправлять автоматически те команды, которые позволят отобразить меню.

```
PRIVATE = [
    BotCommand(command='menu', description='Посмотреть меню'),
    BotCommand(command='about', description='О нас'),
    BotCommand(command='payment', description='Варианты оплаты'),
    BotCommand(command='shipping', description='Варианты доставки')
]
```

```
async def main():
    await bot.delete_webhook(drop_pending_updates=True)

    # Здесь прописана настройка отображения меню в приватных чатах.
    await bot.set_my_commands(
        commands=PRIVATE,
        scope=BotCommandScopeAllPrivateChats()
    )

    await dispatcher.start_polling(
        bot,
        allowed_updates=ALLOWED_UPDATES
    )
```

Магические фильтры, кастомные фильтры, фильтрация сообщений

> *Не всегда удобно работать с пользователями при помощи команд, бывает нужно как то отреагировать на события по более тонким признакам.*
>
> *Например можно отреагировать на какое то определенное слово в тексте (в предложении) и так далее*

Дока

Дока к магическим фильтрам

F-магический фильтр

> *[!attention] Можно воспринимать `F` как объект `message`*

Реагируем на разные типы событий

Можно реагировать и на текст, аудио, видео, и так далее (`F.text` ; `F.audio` ...)

```
# Фильтр по типу месседжа-фото
@user_private_router.message(F.photo)
async def payment_cmd(message: types.Message):
    print(message.photo)
    await message.answer('Это магический фильтр!')
```

Проверяем текст сообщения на равенство чему либо

```
@user_private_router.message(F.text.lower() == 'был')
async def payment_cmd(message: types.Message):
    print(message.photo)
    await message.answer('Это магический фильтр!')
```

Contains

```
@user_private_router.message(F.text.lower().contains('ом'))
async def magic_f(message: types.Message):
    print(message.text)
    await message.answer('Это магический фильтр 2')
```

Not (~) - выражения

Хэндлер сработает в любом случае когда в сообщении не будет присутствовать "ом"

```
@user_private_router.message(~(F.text.lower().contains('ом')))
async def magic_f(message: types.Message):
    print(message.text)
    await message.answer('Это магический фильтр 2')
```

Комбинирование выражений (and, or..)

Можно комбинировать через запятую

```
# Если это текст и если это не "ом"
@user_private_router.message(F.text, ~(F.text.lower().contains('ом')))
async def magic_f(message: types.Message):
    print(message.text)
    await message.answer('Это магический фильтр 2')
```

Есть и специальные операторы:

- `|` - или
- `&` - и

```
@user_private_router.message(
    (F.text.lower().contains('достав')) |
    ~(F.text.lower().contains('варианты доставки'))
)
async def magic_f(message: types.Message):
    print(message.text)
    await message.answer('Это магический фильтр 2')
```

`or_f`

Внутри этой функции можно передать через запятую фильтры и всё это будет работать как ИЛИ

```
@user_private_router.message(
    or_f(
        Command(cmd.MENU.command),
```

```
        F.text.lower() == 'меню'
    )
)
async def menu_cmd(message: types.Message):
    await message.answer(
        text='Наше меню',
        reply_markup=reply.remove_start_kb
    )
```

Несколько обработчиков для одного хендлера

```
@user_private_router.message(Command('shipping'))
@user_private_router.message(
    (F.text.lower().contains('достав')) |
    ~(F.text.lower().contains('варианты доставки'))
)
async def menu_cmd(message: types.Message):
    await message.answer('Варианты доставки')
```

Работа в группах

1. Создать группу
2. Добавить туда бота
3. Сделать бота администратором

Что-бы бота никто не мог добавлять в другие группы

1. Бот-фазер
2. Bot settings
3. Allow groups?
4. Turn groups off

> *[!attention] Если бот до этих настроек был добавлен в группу, то он так там и останется*

Хендлеры для работы бота в группе

Нужно создать отдельный файл хендлеров для работы бота в группе. Хендлер ниже будет проверять сообщения на недопустимые слова.

user_group.py

```
from string import punctuation

from aiogram import types, Router

user_group_router = Router()


RESTRICTED_WORDS = {
    'хрен',
    'грязь',
    'блин'
}
```

```
def clean_text(text: str):
    return text.translate(str.maketrans('', '', punctuation))


# edited_message для того, чтобы отлавливать отредактированные сообщения.
@user_group_router.edited_message()
@user_group_router.message()
async def check_and_clean_bad_words(message: types.Message):
    if RESTRICTED_WORDS.intersection(clean_text(message.text.lower()).split()):
        # Написать пользователю сообщение перед удалением
        await message.reply(
            f'{message.from_user.first_name}, соблюдайте порядок в чате!'
        )
        # Удалить сообщение.
        await message.delete()

        # Если нужно забанить пользователя
        # await message.chat.ban(message.from_user.id)
```

Кастомные фильтры. Разделить хендлеры: одни для работы в группах, другие для лички

> *[!attention] Будет круто навесить фильтр прям на роутер чтобы проверка срабатывала сразу до прохождения по всем хендлерам.*

Сам фильтр

```
from aiogram.filters import Filter
from aiogram import types


# Наследуемся от базового класса Filter.
class ChatTypeFilter(Filter):

    # Буем передавать список типов чатов в которых
    # будет работать тот или иной роутер.
    def __init__(self, chat_types: list[str]) -> None:
        self.chat_types = chat_types

    # Чтобы фильтр сработал нужно переопределить этот метод - асинхронный.
    # Aiogram сам перекинет объект message -
    # сообщение которое пользователь отправил.
    # PS: Так как сюда пробрасывается message - можно устроить любую проверку!
    async def __call__(self, message: types.Message) -> bool:
        # Тут проверяем что тип message имеется в списке наших типов
        return message.chat.type in self.chat_types
```

Применение фильра

Отделить хендлеры только для групп

```python
from aiogram import types, Router

from app.filters.chat_types import ChatTypeFilter


# Список разрешенных типов чатов.
CHAT_TYPES = [
    'group',
    'supergroup',
]

user_group_router = Router()
user_group_router.message.filter(
    ChatTypeFilter(CHAT_TYPES)
)
...
```

Отделить хендлеры только для личных сообщений

```python
from aiogram import types, Router

from app.filters.chat_types import ChatTypeFilter


# Список разрешенных типов чатов.
CHAT_TYPES = [
    'private',
]

user_private_router = Router()
user_private_router.message.filter(
    ChatTypeFilter(CHAT_TYPES)
)
...
```

Клавиатуры и кнопки, форматирование текста

> *[!info]* *Reply keyboard* / *Inline keyboard*

`ReplyKeyboardMarkup` и `KeyboardButton`

Создание и применение

reply.py

```python
from aiogram.types import ReplyKeyboardMarkup, KeyboardButton

from app.common import constants as button


INPUT_TEXT = 'Что Вас интересует?'
```

```
# Каждый список из KeyboardButton - это строка из кнопок.
# Если судить по такой конструкции, то получится две строки по две кнопки.
START_BUTTONS = [
    [
        KeyboardButton(text=button.MENU.description),
        KeyboardButton(text=button.ABOUT.description),
    ],
    [
        KeyboardButton(text=button.SHIPPING.description),
        KeyboardButton(text=button.PAYMENT.description),
    ]
]

start_kb = ReplyKeyboardMarkup(
    keyboard=[
        # Распаковать список с кнопками в список.
        *START_BUTTONS
    ],
    # Что-бы кнопки не были огромными
    resize_keyboard=True,
    # Поменять строку приглашения к вводу сообщения на кастомную.
    input_field_placeholder=INPUT_TEXT
)
```

Применить клавиатуру

```
...
from app.keyboards import reply

...

@user_private_router.message(CommandStart())
async def start_cmd(message: types.Message):
    await message.answer(
        text='Отвечаю на сообщение /start',
        # Вторым параметром можно передать клавиатуру.
        reply_markup=reply.start_kb
    )
...
```

Удалить клавиатуру при нажатии на другую кнопку

reply.py

```
...

start_kb = ReplyKeyboardMarkup(
    ...
)

# Если нужно удалить клавиатуру,
```

```
# то применить данный объект в нужном хендлере.
remove_start_kb = ReplyKeyboardRemove()
```

Как применить удаление

Например при нажатии на кнопку "menu" не нужна клавиатура

```
...

@user_private_router.message(
    or_f(
        Command(cmd.MENU.command),
        F.text.lower() == 'меню'
    )
)
async def menu_cmd(message: types.Message):
    await message.answer(
        text='Наше меню',
        # При переходе к меню удалить клавиатуру.
        reply_markup=reply.remove_start_kb
    )
...
```

ReplyKeyboardBuilder

> *Позволяет работать с клавиатурами более гибко*

Создание и применение

reply.py

```
...

# Создаем экземпляр
start_kb_2 = ReplyKeyboardBuilder()

# Теперь передает кнопки с помощью методов.
start_kb_2.add(*(START_BUTTONS[0] + START_BUTTONS[1]))

# Теперь можно цифрами расписать сколько
# строк по сколько столбцов отображать кнопки.
# Каждая цифра - это ряд (1-я: 1 ряд 2 кнопки; 2-я: 2 ряд 2 кнопки).
start_kb_2.adjust(2, 2)

# Добавить еще клавиатуру на основе предыдущей, но с добавлением новой кнопки.
start_kb_3 = ReplyKeyboardBuilder()

# Наследуемся как бы от другой клавиатуры
start_kb_3.attach(start_kb_2)
# Добавляем новую кнопку
# start_kb_3.add(KeyboardButton(text=button.REVIEW.description))

# Метод row добавит кнопку новым рядом.
start_kb_3.row(KeyboardButton(text=button.REVIEW.description))
```

Применить клавиатуру

```
...
from app.keyboards import reply

...

@user_private_router.message(CommandStart())
async def start_cmd(message: types.Message):
    await message.answer(
        text='Отвечаю на сообщение /start',
        # Вторым параметром можно передать клавиатуру
        reply_markup=reply.start_kb_3.as_markup(
            resize_keyboard=True,
            input_field_placeholder=cmd.INPUT_TEXT
        )
    )
...
```

Дополнительные параметры самих кнопок класса `KeyboardButton`

ПАРАМЕТРЫ

- `request_contact` - Запросить контакт
- `request_location` - запросить локацию

Оба принимают булево значение `True` / `False`

> *[!attention] К одной кнопке нельзя применить несколько параметров - только один!*

Пример

```
...

contact_location_kb = ReplyKeyboardMarkup(
    keyboard=[
        [
            KeyboardButton(
                text='Создать опрос',
                request_poll=KeyboardButtonPollType()
            ),
        ],
        [
            KeyboardButton(text='Отправить номер', request_contact=True),
            KeyboardButton(text='Отправить локацию', request_location=True),
        ]
    ],
    resize_keyboard=True
)
```

Хендлеры

```
...
```

```python
@user_private_router.message(F.contact)
async def get_contact(message: types.Message):
    await message.answer('Номер получен')
    await message.answer(str(message.contact.phone_number))


@user_private_router.message(F.location)
async def get_location(message: types.Message):
    await message.answer('Локация получена')
    await message.answer(str(message.location))
```

Функция для создания клавиатуры

```python
# Удобно будет видеть в хендлерах какими кнопками его клавиатура располагает.
def get_keyboard(
        *buttons: str,
        placeholder: str = None,
        # Передается индекс кнопки к которой нужно
        # прицепить передачу контакта или локации.
        request_contact: int = None,
        request_location: int = None,
        sizes: tuple[int] = (2,),
):
    """
    Parameters request_contact and request_location
    must be as indexes of buttons args for buttons you need.
    Example:
    get_keyboard(
        "Меню",
        "О магазине",
        "Варианты оплаты",
        "Варианты доставки",
        "Отправить номер телефона"
        placeholder="Что вас интересует?",
        request_contact=4,
        sizes=(2, 2, 1)
    )
    """
    keyboard = ReplyKeyboardBuilder()

    for index, text in enumerate(buttons, start=0):
        if request_contact and request_contact == index:
            keyboard.add(KeyboardButton(text=text, request_contact=True))
        elif request_location and request_location == index:
            keyboard.add(KeyboardButton(text=text, request_location=True))
        else:
            keyboard.add(KeyboardButton(text=text))

    return keyboard.adjust(*sizes).as_markup(
        resize_keyboard=True, input_field_placeholder=placeholder
    )
```

Форматирование текста

> *Само форматирование происходит на серверах ТГ, через aiogram передается сам текст с нужными элементами форматирования и каким способом форматировать текст.*

Варианты форматирования

- Marcdown
- HTML

Параметр `parse_mode`, класс `ParseMode`

У каждого метода (например message) есть дополнительные параметры, например `parse_mode`

```
...

@user_private_router.message(
    (F.text.lower().contains('достав')) |
    (F.text.lower().contains('варианты доставки'))
)
@user_private_router.message(Command(cmd.SHIPPING.command))
async def shipping_cmd(message: types.Message):
    await message.answer(
        # Можно писать HTML-теги
        text='<b>Варианты доставки</b>',
        # Нужно указать режим форматирования
        parse_mode=ParseMode.HTML
    )
...
```

`parse_mode` указать для всего бота целиком (в экземпляре бота)

```
...
# Класс самого бота - инициализация.
bot = Bot(token=TOKEN, default=DefaultBotProperties(parse_mode='HTML'))
...
```

> *[!attention] Далее parse_mode можно не указывать в хендлерах*

Дополнительные плюшки для форматирования: `as_list`, `as_marked_section`, `Bold`

```
from aiogram.utils.formatting import as_list, as_marked_section, Bold
...
@user_private_router.message(F.text.lower() == 'варианты оплаты')
@user_private_router.message(Command(cmd.PAYMENT.command))
async def payment_cmd(message: types.Message):
    payment_options = [
        'Картой в боте',
        'При получении карта/кеш',
        'В заведении'
    ]

    # Вернется класс текста
```

```
    text = as_marked_section(
        # Title. Текст сделается жирным
        Bold(f'{cmd.PAYMENT.description}:'),
        # Body
        *payment_options,
        marker='* '
    )
    # Нужно указать как именно мы будем парсить текст.
    await message.answer(text.as_html())
    ...
```

```
...
@user_private_router.message(
    (F.text.lower().contains('достав')) |
    (F.text.lower().contains('варианты доставки'))
)
@user_private_router.message(Command(cmd.SHIPPING.command))
async def shipping_cmd(message: types.Message):
    shipping_options = [
        'Курьер',
        'Самовывоз',
        'Покушаю у вас',
    ]
    forbidden = [
        'Почта', 'Голуби',
    ]
    # Список маркированных объектов с разделителем sep.
    text = as_list(
        as_marked_section(
            Bold(f'{cmd.SHIPPING.description}:'),
            *shipping_options,
            marker='* '
        ),
        as_marked_section(
            Bold('Нельзя:'),
            *forbidden,
            marker='X '
        ),
        sep='\n_____________________\n'
    )
    await message.answer(
        text=text.as_html(),
    )
    ...
```


Машина состояний (Finite State Machine), Админка в боте, Диалоги, фильтр `IsAdmin`

Фильтр `IsAdmin`, работа с админкой

Задумка в том, чтобы администраторы группы были так же и админами бота. Необходимо получить список админов группы и потом этот список применить в фильтре.

Навесить экземпляру бота еще одно свойство `admins` со значением пустого списка:

```
# Класс самого бота - инициализация.
bot = Bot(token=TOKEN, default=DefaultBotProperties(parse_mode='HTML'))
# Такой конструкцией можно создавать новые свойства.
bot.admins = []
```

Наполнить список данными из админов группы:

Так как в личке нет такого понятия как админ, то команду для сбора списка админов будем брать в хендлерах для группы ***user_group.py***

Для этого понадобится секретная команда, которая считает и сохранит список админов. Добавить хендлер ***user_group.py***:

```
...
# Секретная команда
@user_group_router.message(Command('admin'))
async def get_admins(message: types.Message, bot: Bot):
    # Делаем запрос на сервер телеграмма
    admins = await bot.get_chat_administrators(message.chat.id)
    admins = [
        member.user.id
        for member in admins
        if member.status == 'creator' or member.status == 'administrator'
    ]
    bot.admins = admins
    # Можно удалить сразу сообщение, чтобы никто не увидел
    if message.from_user.id in admins:
        await message.delete()
...
```

Сам фильтр `IsAdmin`

```
...
class IsAdmin(Filter):
    def __init__(self) -> None:
        pass

    async def __call__(self, message: types.Message, bot: Bot):
        return message.from_user.id in bot.admins
```

Машина состояний

> *Специальный алгоритм который позволяет провести пользователя по четким пунктам диалога взять от него ответы и использовать в дальнейшем в приложении.*

FSM-стратегия и хранилища

> *[!attention] Всё это настраивается при определении диспетчера*

Storage

Нужно разобраться где будет храниться вся информация (машины состояния).

По умолчанию данные хранятся в оперативной памяти. Далее можно указать БД.

FSM-strategy

Какие есть стратегии формирования данных - пользователей много, и у каждого пользователя могут быть свои состояния/диалоги (начал процесс заполнения формы и ее нужно закончить, после чего куда-то сохранить). - ?

По умолчанию используется стратегия `USER_IN_CHAT` - для каждого отдельного пользователя, в любом чате будет вестись его отдельное состояние

Есть и другие...

1) Описать шаги FSM: класс `StatesGroup`

```
from aiogram.fsm.state import StatesGroup, State
...
# Код ниже для машины состояний (FSM)
class AddProduct(StatesGroup):
    # Каждый пункт это состояние на котором
    # может находиться каждый пользователь (?).
    name = State()
    description = State()
    price = State()
    image = State()
...
```

2) Передать в хендлеры состояние пользователя: `FSMContext`

Помимо события message передать состояние пользователя Aiogram автоматически прокинет этот параметр

```
from aiogram.fsm.context import FSMContext
...
@admin_router.message(F.text == 'Добавить товар')
async def add_product_fsm(message: types.Message, state: FSMContext):
    await message.answer(
        text='Введите название товара',
        reply_markup=BACK_CANCEL_KB
    )
...
```

> *[!attention] Для каждого хендлера FSM нужно прокинуть параметр `state` Чтобы мы могли оперировать состояниями пользователя*

3) Процесс FSM

Перед началом процесса состояния проверить, что у пользователя нет активных состояний...

```
# КОД НИЖЕ ДЛЯ МАШИНЫ СОСТОЯНИЙ (FSM)
class AddProduct(StatesGroup):
    # Каждый пункт это состояние на котором
    # может находиться каждый пользователь (?).    name = State()
    description = State()
    price = State()
    image = State()
```

```python
@admin_router.message(Command('отмена'))
@admin_router.message(F.text.casefold() == 'отмена')
async def cancel(message: types.Message, state: FSMContext) -> None:
    await message.answer(
        text='Действия отменены',
        reply_markup=ADMIN_KB
    )


@admin_router.message(Command('назад'))
@admin_router.message(F.text.casefold() == 'назад')
async def back(message: types.Message, state: FSMContext) -> None:
    await message.answer(
        text='ок, вы вернулись к прошлому шагу',
        reply_markup=BACK_CANCEL_KB
    )


# 1) FSM начнется, если у пользователя нет активных состояний - StateFilter(None).
# Это точка входа в состояние.
@admin_router.message(StateFilter(None), F.text == 'Добавить товар')
async def add_product_fsm(message: types.Message, state: FSMContext):
    await message.answer(
        text='Введите название товара',
        reply_markup=BACK_CANCEL_KB
    )
    # Нужно указать в какое состояние нужно стать
    await state.set_state(AddProduct.name)


# 2) Если пользователь в состоянии AddProduct.name и ввел текст,
# то продолжаем FSM
@admin_router.message(AddProduct.name, F.text)
async def add_name(message: types.Message, state: FSMContext):
    # Записываем name из предыдущего шага
    await state.update_data(name=message.text)
    await message.answer(
        text='Введите описание товара',
        reply_markup=BACK_CANCEL_KB
    )
    # Меняем состояние на description
    await state.set_state(AddProduct.description)


# 3) Если пользователь в состоянии AddProduct.description и ввел текст,
# то продолжаем FSM
@admin_router.message(AddProduct.description, F.text)
async def add_description(message: types.Message, state: FSMContext):
    # Записываем description
    await state.update_data(description=message.text)
```

```
    await message.answer(
        text='Введите стоимость товара',
        reply_markup=BACK_CANCEL_KB
    )
    # Меняем состояние на price
    await state.set_state(AddProduct.price)


# 4) Если пользователь в состоянии AddProduct.price и ввел текст,
# то продолжаем FSM
@admin_router.message(AddProduct.price, F.text)
async def add_price(message: types.Message, state: FSMContext):
    # Записываем price
    await state.update_data(price=message.text)
    await message.answer(
        text='Загрузите изображение товара',
        reply_markup=BACK_CANCEL_KB
    )
    # Меняем состояние на image
    await state.set_state(AddProduct.image)


# 5) Если пользователь в состоянии AddProduct.image и загрузил фото,
# то продолжаем FSM
@admin_router.message(AddProduct.image, F.photo)
async def add_image(message: types.Message, state: FSMContext):
    # Записываем в словарь фото.
    # photo[-1] - означает самое высокое качество,    # так как на серверах хранится
несколько вариантов фото.    # К каждому изображению ТГ присваивает уникальные id
await state.update_data(image=message.photo[-1].file_id)
    await message.answer(
        text='Товар добавлен',
        reply_markup=ADMIN_KB
    )
    # Теперь полученные данные можно куда-нибудь сохранить.
    data = await state.get_data()
    await message.answer(str(data))
    # Когда пользователь прошел все пункты -
    # очистить состояние пользователя и удалить все данные из машины состояния!
await state.clear()
```

Система отмены или возврата на предыдущие шаги

Отмена

```
# Сбросить состояние пользователя.
# '*' - обозначает любое состояние пользователя.
@admin_router.message(StateFilter('*'), Command('отмена'))
@admin_router.message(StateFilter('*'), F.text.casefold() == 'отмена')
async def cancel(message: types.Message, state: FSMContext) -> None:
    # Сохраним текущее состояние в переменную.
    current_state = state.get_state()
    # Если у пользователя нет никакого состояния ..
```

```
    if current_state is None:
        # Завершаем работу хендлера.
        return
    # В ином случае очищаем данные и убираем все состояния.
    await state.clear()

    await message.answer(
        text='Все действия отменены',
        reply_markup=ADMIN_KB
    )
```

Возврат на предыдущие шаги

```
class AddProduct(StatesGroup):
    name = State()
    description = State()
    price = State()
    image = State()
    # Новая конструкция
    texts = {
        'AddProduct:name': 'Введите название заново:',
        'AddProduct:description': 'Введите описание заново:',
        'AddProduct:price': 'Введите стоимость заново:',
        'AddProduct:image': 'Этот стейт последний, поэтому...',
    }

@admin_router.message(StateFilter('*'), Command('назад'))
@admin_router.message(StateFilter('*'), F.text.casefold() == 'назад')
async def back(message: types.Message, state: FSMContext) -> None:
    current_state = await state.get_state()
    if current_state == AddProduct.name:
        await message.answer(
            'Предыдущего шага нет, '
            'или введите название товара или нажмите "отмена"'
        )
        return
    previous = None
    for step in AddProduct.__all_states__:
        if step.state == current_state:
            print(f'{previous=} {current_state=}')
            await state.set_state(previous)
            await message.answer(
                'Вы вернулись к прошлому шагу '
                f'\n {AddProduct.texts[previous.state]}'
            )
            return
        previous = step
```

Обработка некорректного ввода

> *[!attention] ПОСЛЕДОВАТЕЛЬНОСТЬ СТРОГО ВАЖНА!*

```
...
# 2) Если пользователь в состоянии AddProduct.name и ввел текст,
# то продолжаем FSM
@admin_router.message(AddProduct.name, F.text)
async def add_name(message: types.Message, state: FSMContext):
    # Записываем name из предыдущего шага
    await state.update_data(name=message.text)
    await message.answer(
        text='Введите описание товара'
    )
    # Меняем состояние на description
    await state.set_state(AddProduct.description)


# Если вместо текста будет другое событие,
# то выполнится этот хендлер, но состояние останется прежним.
@admin_router.message(AddProduct.name)
async def fix_name(message: types.Message):
    await message.reply(
        text=(
            'Название товара было введено некорректно, '
            'введите название еще раз'
        )
    )
...
```

И так далее для каждого события!

Middleware и SQLAlchemy. Inline-кнопки и CallbackQuery

Middleware

Как в хендлеры передаются такие объекты как `message` или `state`? - все это передается благодаря **промежуточным слоям** (Точно так же нужно будет передавать сессию для работы с БД.).

Промежуточные слои бывают:

- **Outer middleware** (работают до фильтров)
- **Middleware** (Работают после фильтров)

> *[!faq] Middleware можно передавать в объект роутера после его определения или на корневой - диспетчер*

Пример из доки: класс `BaseMiddleware`

Пример кода Middleware

```
from typing import Callable, Dict, Any, Awaitable

from aiogram import BaseMiddleware
from aiogram.types import Message
```

```python
class CounterMiddleware(BaseMiddleware):
    def __init__(self) -> None:
        self.counter = 0

    async def __call__(
        self,
        # Автоматически пробрасывается экземпляр хендлера
        handler: Callable[[Message, Dict[str, Any]], Awaitable[Any]],
        # Какое было событие? Message, CallbackQuery ...
        event: Message,
        # data - Специальный словарь, который в себе собирает все,
        # что может передаваться в хендлер -
        # промежуточные слои (state, session (скоро), и так далее)
        data: Dict[str, Any]
    ) -> Any:
        self.counter += 1
        data['counter'] = self.counter
        return await handler(event, data)
```

Применение к роутеру

```python
...
admin_router = Router()
admin_router.message.filter(ChatTypeFilter(['private']), IsAdmin())
# Сработает после всех фильтраций
admin_router.message.middleware(CounterMiddleware())
...
```

Каждый раз, когда будет срабатывать admin_router, будет вызываться `CounterMiddleware`.

Применить middleware к диспетчеру: глобальное событие `update` и предок всех типов событий - `TelegramObject`

> *Помимо события message и так далее, есть глобальное событие `update` - представляет любое событие которое может быть.*

```python
...
dispatcher = Dispatcher()
# Такой промежуточный слой сработает раньше всех, так как он над всеми апдейтами манипулирует.
dispatcher.update.outer_middleware(CounterMiddleware())
dispatcher.include_routers(*ROUTERS)
...
```

> *[!attention] Но, если цеплять промежуточный слой на такой ранний этап для обработки всех событий непосредственно самим фреймворком, то в промежуточном слое нужно указать предка всех типов событий - `TelegramObject`*

Изменить middleware для работы со всеми типами событий

```python
from typing import Callable, Dict, Any, Awaitable

from aiogram import BaseMiddleware
from aiogram.types import Message, TelegramObject
```

```python
class CounterMiddleware(BaseMiddleware):
    def __init__(self) -> None:
        self.counter = 0

    async def __call__(
        self,
        handler: Callable[[TelegramObject, Dict[str, Any]], Awaitable[Any]],
        # Какое было событие? Message, CallbackQuery ...
        event: TelegramObject,
        # data - Специальный словарь, который в себе собирает все,
        # что может передаваться в хендлер -
        # промежуточные слои (state, session (скоро), и так далее)
        data: Dict[str, Any]
    ) -> Any:
        self.counter += 1
        data['counter'] = self.counter
        return await handler(event, data)
```

Inline-кнопки

Данный вид кнопок прикрепляется к сообщению!

Код кнопок

```python
from aiogram.types import InlineKeyboardButton
from aiogram.utils.keyboard import InlineKeyboardBuilder


def get_callback_btns(
        # * - автоматический запрет на передачу неименованных аргументов.
        *,
        # В словаре будет указываться text как ключ,
        # а в value - строка с данными, которые отправятся боту.
        btns: dict[str, str],
        sizes: tuple[int] = (2,)
):
    keyboard = InlineKeyboardBuilder()
    # Проходимся по словарю
    for text, data in btns.items():
        # Reply-кнопки отправляют только то,
        # что на них написано в чат, а inline нет.
        # Тут создаем именно кнопку.
        # Параметр text - только для отображения,
        # а в callback_data передаются данные,
        # которые бот сможет как то обработать
        # хендлером отлавливающим CallbackQuery-события.
        # PS: callback_data не отправляется в чат,
        # то есть эти данные невидимы для пользователя.
        # PPS: Вместо callback_data можно отправить url -
        # тогда произойдет типа редиректа.
        keyboard.add(InlineKeyboardButton(text=text, callback_data=data))
```

```
    return keyboard.adjust(*sizes).as_markup()


# Если понадобятся кнопки с URL
def get_url_btns(
        *,
        btns: dict[str, str],
        sizes: tuple[int] = (2,)):
    keyboard = InlineKeyboardBuilder()

    for text, url in btns.items():
        keyboard.add(InlineKeyboardButton(text=text, url=url))

    return keyboard.adjust(*sizes).as_markup()


# Создать микс из CallBack и URL кнопок
def get_inlineMix_btns(
        *,
        btns: dict[str, str],
        sizes: tuple[int] = (2,)):
    keyboard = InlineKeyboardBuilder()

    for text, value in btns.items():
        if '://' in value:
            keyboard.add(InlineKeyboardButton(text=text, url=value))
        else:
            keyboard.add(InlineKeyboardButton(text=text, callback_data=value))

    return keyboard.adjust(*sizes).as_markup()
```

Применение кнопок

Каждая отправленная картинка будет обладать описанием и двумя кнопками, которые под капотом содержат УЖЕ нужные данные.

```
...
@admin_router.message(F.text == 'Список товаров')
async def get_products_list(message: types.Message, session: AsyncSession):
    products = await product_crud.get_multi(session=session)
    for product in products:
        await message.answer_photo(
            product.image,
            # caption - это описание картинки
            caption=(
                f'<strong>{product.name}</strong>\n'
                f'{product.description}\n'
                f'Стоимость: {round(product.price, 2)}'
            ),
            # Передается как обычная клавиатура
            reply_markup=get_callback_btns(
                btns={
```

```
                    # Передаем айди каждого продукта,
                    # потом, в другом хендлере (с событием CallbackQuery)
                    # нужно будет распарсить айди.
                    'Удалить': f'delete_{product.id}',
                    'Изменить': f'change_{product.id}',
                }
            )
        )
    await message.answer('ОК, вот список товаров:')
...
```

Ловим CallbackQuery-события (тип апдейта `callback_query`): Удаление товара

1) Разрешить `callback_query`

Нужно прописать новый разрешенный тип апдейта в `ALLOWED_UPDATES` - `callback_query`

```
ALLOWED_UPDATES = ['message', 'edited_message', 'callback_query']
```

> *[!info] Что-бы не прописывать каждый тип апдейта*
>
> ```
> # Здесь бот начнет слушать сервер ТГ,
> # и спрашивать у него о наличии обновлений.
> await dispatcher.start_polling(
>     bot,
>     # allowed_updates=ALLOWED_UPDATES,
>     # Что-бы не прописывать каждый тип апдейта.
>     # Те апдецты которые мы используем - автоматически будут передаваться сюда.
>     allowed_updates=dispatcher.resolve_used_update_types()
> )
> ```

2) Прописать хендлер с типом события `callback_query`

```
@admin_router.callback_query(F.data.startswith('delete_'))
async def delete_product(
        callback: types.CallbackQuery,
        session: AsyncSession
):
    product_id = int(callback.data.split('_')[-1])
    await product_crud.remove(obj_id=product_id, session=session)

    # Данная конструкция нужна для того,
    # что-бы по центру отобразился полупрозрачный всплывающий текст "Товар удален".
    # Нужно указывать ОБЯЗАТЕЛЬНО, так как нужно
    # серверу дать сигнал о том, что мы приняли callback_query!
    # PS: Скобки можно оставить пустыми.
    await callback.answer('Товар удален')

    # А здесь просто придет сообщение в личку, что товар удален.
    await callback.message.answer('Товар удален')
```

Ловим CallbackQuery-события (тип апдейта `callback_query`): Изменение товара

> *Нужно немного переделать уже имеющуюся машину состояния на **добавление** и на **изменение** товара.*

Модифицировать класс состояния `AddProduct`

Добавить новое свойство для временного хранения ID продукта

```
class AddProduct(StatesGroup):
    # Каждый пункт это состояние на котором
    # может находиться каждый пользователь (?).
    name = State()
    description = State()
    price = State()
    image = State()

    change_product_id = None

    texts = {
        'AddProduct:name': 'Введите название заново:',
        'AddProduct:description': 'Введите описание заново:',
        'AddProduct:price': 'Введите стоимость заново:',
        'AddProduct:image': 'Этот стейт последний, поэтому...',
    }
```

Хендлер `change_product`

```
@admin_router.callback_query(StateFilter(None), F.data.startswith('change_'))
async def change_product(
        callback: types.CallbackQuery,
        state: FSMContext,
        # session: AsyncSession
):
    product_id = int(callback.data.split('_')[-1])
    # product = await product_crud.get(obj_id=product_id, session=session)
    AddProduct.change_product_id = product_id
    await callback.answer()
    await callback.message.answer(
        text='Введите название товара',
        reply_markup=FSM_MANAGEMENT_KB
    )
    # Нужно указать в какое состояние нужно стать
    await state.set_state(AddProduct.name)
```

Затем все дальше пойдет своим чередом

Кнопка "пропустить"

```
@admin_router.message(
    AddProduct.name,
```

```
    or_f(F.text.strip().lower() == 'пропустить', F.text)
)
async def add_name_fsm(message: types.Message, state: FSMContext):
    if 'пропустить' in message.text.lower():
        pass
    else:
        # Записываем name из предыдущего шага
        await state.update_data(name=message.text)
    await message.answer(
        text='Введите описание товара'
    )
    # Меняем состояние на description
    await state.set_state(AddProduct.description)
```

Код завершения изменения продукта

```
@admin_router.message(
    AddProduct.image,
    or_f(F.text.strip().lower() == 'пропустить', F.photo)
)
async def add_image_fsm(
        message: types.Message,
        state: FSMContext,
        session: AsyncSession
):
    if message.text and 'пропустить' in message.text.lower():
        pass
    else:
        await state.update_data(image=message.photo[-1].file_id)
    await message.answer(
        text='Товар добавлен/изменен',
        reply_markup=ADMIN_KB
    )
    data = await state.get_data()
    if AddProduct.change_product_id:
        await product_crud.update(
            obj_id=int(AddProduct.change_product_id),
            data=data, session=session
        )
        AddProduct.change_product_id = None
    else:
        await product_crud.create(obj_in=data, session=session)
    # Когда пользователь прошел все пункты -
    # очистить состояние пользователя и удалить все данные из машины состояния!
    await state.clear()
```

Про `edit_`: изменять сообщения вместо удаления и отправки нового - для красоты и создания интерфейса.

```
...
# Список разрешенных типов чатов.
CHAT_TYPES = [
    'private',
]

user_private_router = Router()
user_private_router.message.filter(
    ChatTypeFilter(CHAT_TYPES)
)


@user_private_router.message(CommandStart())
async def start_cmd(message: types.Message):
    await message.answer("Привет, я виртуальный помощник",
                         reply_markup=get_callback_btns(btns={
                             'Нажми меня': 'some_1'
                         }))


# Отлавливаем callback и зменяем.
# PS: Вместе с callback передается сообщение целиком,
# к которому была прикреплена callback клавиатура.
@user_private_router.callback_query(F.data.startswith('some_'))
async def counter(callback: types.CallbackQuery):
    number = int(callback.data.split('_')[-1])
    # 1) Нельзя изменить текст на изображение!
    # Преимущества редактирования сообщений вместо его удаления:
    #   - Меньше хендлеров
    #   - Интерфейс лояльный
    #   - У ТГ есть ограничения: на редактирование нет временных ограничений,
    #       а вот удалить, после истечения 48 часов - не удастся,
    #       и чат захломится старыми неактуальными сообщениями.
    #
    #
     await callback.message.edit_text(
        text=f"Нажатий - {number}",
        reply_markup=get_callback_btns(btns={
            'Нажми еще раз': f'some_{number + 1}'
        }))
```

Запуск чего-то во время старта/завершения бота

```
...

# Обязательно нужно передать бота (?)
async def on_startup_func(bot):
    print('Запустили бот')
```

```python
async def on_shutdown_func(bot):
    print('Завершили бот')


async def main():
    # Вот эти функции: одна запкстится при старте бота,
    # другоая, когда бот завершит работу.
    dispatcher.startup.register(on_startup_func)
    dispatcher.shutdown.register(on_shutdown_func)

    await bot.delete_webhook(drop_pending_updates=True)
    await bot.set_my_commands(
        commands=PRIVATE,
        scope=BotCommandScopeAllPrivateChats()
    )
     await dispatcher.start_polling(
        bot,
        allowed_updates=ALLOWED_UPDATES
    )


if __name__ == '__main__':
    asyncio.run(main())
```