Материалы к занятию

HashTag - Данный фильтр проверяет то, что в сообщении содержится определённый хеш (#) или кеш ($) тег. Может использоваться либо как callable, либо как аргументы hashtags и cashtags.

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



@dp.message_handler(hashtags='money')
@dp.message_handler(cashtags=['eur', 'usd'])
async def hashtag_example(msg: types.Message):
    await msg.answer('Eeee, деньги 😎')
```

Regexp - Фильтр, проверяющий регулярное выражение. Подключён на большинство обработчиков. Для примера напишем обработчик, который будет обрабатывать ссылку на изображения:

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



IMAGE_REGEXP = r'https://.+?\.(jpg|png|jpeg)'


@dp.message_handler(filters.Regexp(IMAGE_REGEXP))
async def regexp_example(msg: types.Message):

    await msg.answer('Похоже на картинку, не так ли?')
```

Этот фильтр может использоваться также и как аргумент. Используйте для этого ключевое слово regexp:

```
@dp.message_handler(regexp=IMAGE_REGEXP)
async def regexp_example(msg: types.Message):
. . .
```

RegexpCommandsFilter - Фильтр, проверяющий команду через регулярное выражение. Может использоваться как аргумент regexp_commands (Именно поэтому у меня функция с двумя декораторами, вам стоит использовать один из приведенных вариантов).

Возьмём идентичный пример. Реализуем команду image с несколькими префиксами, которые проверяют ссылку на изображение:

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


IMAGE_REGEXP = r'https://.+?\.(jpg|png|jpeg)'
COMMAND_IMAGE_REGEXP = r"/image:" + IMAGE_REGEXP


@dp.message_handler(filters.RegexpCommandsFilter([COMMAND_IMAGE_REGEXP]))
@dp.message_handler(regexp_commands=[COMMAND_IMAGE_REGEXP])
async def command_regexp_example(msg: types.Message):
    await msg.answer('По вашей команде докладываю, что данная ссылка
является изображением!')
```

Изначально в фильтре идёт проверка на команду, так что сообщение должно начинаться с косой черты.

StateFilter - Этот фильтр проверяет состояние, в котором находится пользователь. Он может использоваться только как аргумент функции state. Для примера опишем 2 обработчика: для присвоения состояния и для его сброса.

```
from aiogram import types

from aiogram.dispatcher import FSMContext



@dp.message_handler(commands='set_state')
async def set_state(msg: types.Message, state: FSMContext):
    """Присваиваем пользователю состояние для теста"""
    await state.set_state('example_state')
    await msg.answer('Состояние установлено')


@dp.message_handler(state='example_state')
async def state_example(msg: types.Message, state: FSMContext):
    await msg.answer('Ой всё, иди отсюда')
    await state.finish()
```

Text - Ещё один очень популярный фильтр. Он проверяет текст, будь это текст сообщения, или callback_data кнопки. Кроме полной идентичности (equals), данный фильтр также может проверить то, что текст начинается, содержит или заканчивается какой-либо строкой. Вы можете использовать импортированный класс, передавая в него аргументы либо использовать фильтр как аргумент функции. Пройдёмся по аргументам конструктора класса:

equals — строка, которой текст должен быть идентичен
contains — строка, которую должен содержаться текст
startswith — строка, которой текст должен начинаться
endswith — строка, которой текст должен заканчиваться
ignore_case — игнорировать регистр текст. Проверяется с помощью str.lower().

Данный фильтр может использоваться как аргумент. Для этого используйте text, text_contains, text_startswith или text_endswith.
Рассмотрим следующий пример: допустим нам нужно отвечать в чате участнику за какую-нибудь фразу:

```
import asyncio

from aiogram import types
from aiogram.dispatcher import filters



FORBIDDEN_PHRASE = [
    'C++',
    'Python'
]


@dp.message_handler(filters.Text(contains=FORBIDDEN_PHRASE,
ignore_case=True))
async def text_example(msg: types.Message):
    await msg.reply('Ответ!')
```

Таким образом, если сообщение пользователя будет содержать обе строки из списка, бот ответит сообщением.

IDFilter - Ещё один фильтр, который заслуживает внимания — фильтр для проверки идентификатора. Он может использоваться как аргумент user_id, chat_id, так и как callable объект IDFilter(user_id=12345789)

Сам фильтр имеет два аргумента:
user_id — проверяет ID пользователя
chat_id — проверяет ID чата

Представим, что нам нужно написать обработчик, который будет отвечать на все сообщения в диалоге с админом, если обновление не попало ни в один из обработчиков (для этого расположим обработчик в конце файла). Таким образом в конфигурационном файле у нас есть константа типа List с идентификаторами суперпользователей SUPERUSER_IDS. Пример будет иметь следующий вид:

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


from app.config import SUPERUSER_IDS


@dp.message_handler(filters.IDFilter(chat_id=SUPERUSER_IDS))
@dp.message_handler(chat_id=SUPERUSER_IDS)
async def id_filter_example(msg: types.Message):
    await msg.answer('Да, помню тебя')
```

Этот обработчик сработает на сообщение суперпользователя, и только если ни это сообщение не прошло по фильтрам ни в один из предыдущих фильтров.

AdminFilter - Один из любимых фильтров разработчиков чат модераторов. Как понятно из названия проверяет, что запрос прилетел от администратора чата. Может использоваться как callable объект, или аргумент функции.
Допустим мы пишем бота для администрирования чата и нам нужна команда, для изменения изображения чата. Естественно, такую команду не следует доверять обычным пользователям, поэтому пусть ответственность за использование этой команды будет исключительно на плечах администрации чата

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




@dp.message_handler(commands='change_photo', is_chat_admin=True)
@dp.message_handler(filters.Command('change_photo'), filters.AdminFilter())
async def chat_admin_example(msg: types.Message):
```

```
    await msg.answer('Нет')
```

Тут используются два декоратора исключительно для демонстрации. Вам стоит выбрать одно из решений. Также обратите внимание, что вперёд стоит фильтр на команду, потому что AdminFilter делает запрос в API, что требует времени, а значит замедляет работу бота.
Кроме логического типа данных, этот фильтр может также содержать ID чата. В этом случае он будет проверять, что запрос пришел именно от администратора чата с конкретным идентификатором, а не текущего чата.
Этот фильтр не может быть False.
Если при использовании этого фильтра как callable не передавать аргументы, проверяться будет администрация текущего чата.

IsReplyFilter - один из простейших фильтров. Он проверит, является ли сообщение или пост в канале ответом. Может использоваться как аргумент is_reply.
Допустим по команде user_id пусть бот возвращает ID пользователя, ответом на сообщение которого была использована команда.

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




@dp.message_handler(is_reply=True, commands='user_id')
@dp.message_handler(filters.IsReplyFilter(True), commands='user_id')
async def reply_filter_example(msg: types.Message):
    await msg.answer(msg.reply_to_message.from_user.id)
```

IsSenderContact - этот фильтр проверяет, что пользователь отправил именно свой контакт. Может использоваться как callable или как аргумент is_sender_contact.

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




@dp.message_handler(content_types='contact', is_sender_contact=True)
@dp.message_handler(filters.IsSenderContact(True), content_types='contact')
async def sender_contact_example(msg: types.Message):
    await msg.answer('Это вы')
```

ForwardedMessageFilter - этот фильтр проверяет, что отправленное сообщение является пересланным. Может использоваться как аргумент is_forwarded или как объект класса.

Пусть бот ругает пользователя, за попытку отправить чужое сообщение

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




@dp.message_handler(is_forwarded=True)
@dp.message_handler(filters.ForwardedMessageFilter(True))
async def forwarded_example(msg: types.Message):
    await msg.answer('Не пытайся меня обмануть, я же вижу, что это не твоё
сообщение')
```

ChatTypeFilter - Из названия этого фильтра понятно, что он проверяет тип чата. Он может использоваться либо как объект, либо как аргумент chat_type. Принимает либо строку, либо types.ChatType (что тоже является строкой).
Допустим нам нужно, чтобы бот по команде is_pm подтверждал, что команда выполнена в личных сообщениях.

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




@dp.message_handler(chat_type=types.ChatType.PRIVATE, commands='is_pm')
@dp.message_handler(chat_type='private', commands='is_pm')
@dp.message_handler(filters.ChatTypeFilter(types.ChatType.PRIVATE),
commands='is_pm')
async def chat_type_example(msg: types.Message):

    await msg.answer('Да, это личные сообщения')
```

Давайте реализуем простой пример бота с учетом изученных нами фильтров.

bot.py

```
import logging
```

```
from aiogram import Bot, Dispatcher, executor, types
import aiogram.utils.markdown as fmt
from aiogram.dispatcher.filters import CommandHelp, CommandStart, Text

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}")

@dp.message_handler(CommandHelp())
async def bot_start(message: types.Message):
   await message.answer(f"{message.from_user.full_name} вам нужна помощь?")

@dp.message_handler(Text(startswith='Бот'))
async def bot_start(message: types.Message):
   await message.answer(f"ВЫ ВВЕЛИ ТЕКСТ НАЧИНАЮЩИЙСЯ НА БОТ")

@dp.message_handler(Text(equals='телеграм'))
async def bot_start(message: types.Message):
   await message.answer(f"ВЫ ВВЕЛИ слово телеграмм")

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

config.py

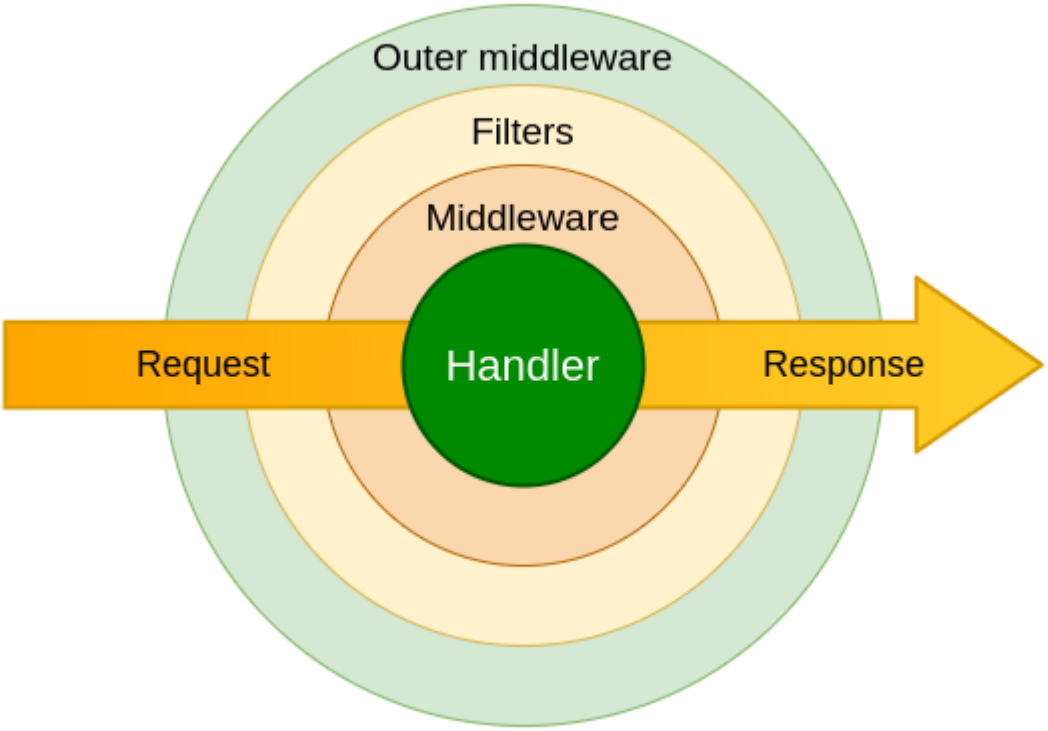
```
import os

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

Middleware - это словно мост, связывающий между собой две части программы или системы. Задача мидлвари вклиниваться в процесс обработки апдейтов для реализации какой-либо логики. Что можно делать внутри мидлварей:

- логировать события;
- передавать в хэндлеры какие-то объекты (например, сессию базы данных из пула сессий);
- подменять обработку апдейтов, не доводя до хэндлеров;

- по-тихому пропускать апдейты, как будто их и не было;
- и т.п

Воспользуемся схемой из официальной документации



Мидлварей два вида: внешние (outer) и внутренние (inner или просто «мидлвари»). В чём разница? Outer выполняются до начала проверки фильтрами, в inner — после. На практике это значит, что апдейт/сообщение/колбэк, проходящий через outer-мидлварь, может так ни в один хэндлер и не попасть, а если он попал в inner, то дальше 100% будет какой-то хэндлер.
Мидлварь обязательно должна быть унаследована от BaseMiddleware
Все методы для точек взаимодействия должны иметь названия:
on_<point>_<event_type>,
То есть *on_pre_process_message, on_post_process_update*

Теперь рассмотрим простейшую мидлварь с точки зрения кода:

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

from aiogram.dispatcher.middlewares import BaseMiddleware
from aiogram.types.base import TelegramObject
```

```
class SomeMiddleware(BaseMiddleware):
   async def __call__(
       self,
       handler: Callable[[TelegramObject, Dict[str, Any]], Awaitable[Any]],
       event: TelegramObject,
       data: Dict[str, Any]
   ) -> Any:
       print("Before handler")
       result = await handler(event, data)
       print("After handler")
       return result
```

Каждая мидлварь, построенная на классах (впрочем, возможны и иные варианты), должна реализовывать метод __call__() с тремя аргументами:

1. handler — собственно, объект хэнлдлера, который будет выполнен. Имеет смысл только для inner-мидлварей, т.к. outer-мидлварь ещё не знает, в какой хэндлер попадёт апдейт.
2. event — тип Telegram-объекта, который обрабатываем. Обычно это Update, Message, CallbackQuery или InlineQuery (но не только). Если точно знаете, какого типа объекты обрабатываете, смело пишите, например, Message вместо TelegramObject.
3. data — связанные с текущим апдейтом данные: FSM, переданные доп. поля из фильтров, флаги (о них позже) и т.д. В этот же data мы можем класть из мидлварей какие-то свои данные, которые будут доступны в виде аргументов в хэндлерах (так же, как в фильтрах).

С телом функции ещё интереснее.

- Всё, что вы напишете до строки:
  result = await handler(event, data)
  будет выполнено до передачи управления нижестоящему обработчику (это может быть другая мидлварь или непосредственно хэндлер).
- Всё, что вы напишете после строки:
  result = await handler(event, data)
  будет выполнено уже после выхода из нижестоящего обработчика.
- Если вы хотите, чтобы обработка продолжилась, вы должны вызвать await handler(event, data). Если хотите «дропнуть» апдейт, просто не вызывайте его.
- Если вам не интересно, что происходит после выполнения функции handler(...), можно сделать return await handler(event, data) и выйти из функции __call__().

Все привычные нам объекты (Message, и т.д.) являются апдейтами (Update), поэтому для Message сначала выполнятся мидлвари для Update, а уже затем для самого Message. Оставим на месте наши print() из примера выше и проследим, как будут выполняться мидлвари, если мы зарегистрируем по одной outer- и inner-мидлвари для типов Update и Message.

Есть несколько точек, куда мидлварью можно вклиниваться:
- pre_process: выполняется каждый раз перед началом фильтрации
- process: выполняется каждый раз после прохождения фильтра перед запуском хендлера
- post_process: выполняется каждый раз после обработки всего

Если сообщение (Message) в конечном счёте обработалось каким-то хэндлером:

1. *[Update Outer] Before handler*
2. *[Update Inner] Before handler*
3. *[Message Outer] Before handler*
4. *[Message Inner] Before handler*
5. *[Message Inner] After handler*
6. *[Message Outer] After handler*
7. *[Update Inner] After handler*
8. *[Update Outer] After handler*

Если сообщение не нашло нужный хэндлер:

1. *[Update Outer] Before handler*
2. *[Update Inner] Before handler*
3. *[Message Outer] Before handler*
4. *[Message Outer] After handler*
5. *[Update Inner] After handler*
6. *[Update Outer] After handler*

Пример миддлеваре

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

from aiogram.dispatcher.middlewares import BaseMiddleware
from aiogram.types import Message, CallbackQuery


def _is_weekend() -> bool:
   # 5 - суббота, 6 - воскресенье
   return datetime.utcnow().weekday() in (5, 6)


# Это будет inner-мидлварь на сообщения
class WeekendMessageMiddleware(BaseMiddleware):
   async def __call__(
       self,
       handler: Callable[[Message, Dict[str, Any]], Awaitable[Any]],
       event: Message,
       data: Dict[str, Any]
   ) -> Any:
       # Если сегодня не суббота и не воскресенье,
       # то продолжаем обработку.
       if not _is_weekend():
           return await handler(event, data)
```

```
        # В противном случае просто вернётся None
        # и обработка прекратится


# Это будет outer-мидлварь на любые колбэки
class WeekendCallbackMiddleware(BaseMiddleware):
    async def __call__(
        self,
        handler: Callable[[CallbackQuery, Dict[str, Any]], Awaitable[Any]],
        event: CallbackQuery,
        data: Dict[str, Any]
    ) -> Any:
        # Если сегодня не суббота и не воскресенье,
        # то продолжаем обработку.
        if not _is_weekend():
            return await handler(event, data)
        # В противном случае отвечаем на колбэк самостоятельно
        # и прекращаем дальнейшую обработку
        await event.answer(
            "Бот по выходным не работает!",
            show_alert=True
        )
        return
```