Материалы к занятию

Давайте сегодня немного разберем написание собственных фильтров. Возьмем примеры из прошлого занятия

```
class AdminFilter(BoundFilter):
    async def check(self, message: types.Message):
        member = await message.chat.get_member(message.from_user.id)
        return member.is_chat_admin()


class IsGroup(BoundFilter):
    async def check(self, message: types.Message):
        return message.chat.type in (
            types.ChatType.GROUP,
            types.ChatType.SUPERGROUP,
        )
```

В хендлерах можно прописывать условия, эти условия называются фильтрами. Для написания собственного фильтра мы должны импортировать класс BoundFilter.

Разберем построчно. Мы создаем класс, который наследуется от импортированного класса

```
class AdminFilter(BoundFilter):
```

Далее в функции чек мы указываем тип апдейта, который нам должен прилететь. Например если в обработчике используется @dp.message_handler, то и в функцию попадет нам апдейт сообщения message:types.Message.

```
async def check(self, message: types.Message):
```

далее мы получаем id нашего пользователя и сохраняем его в переменной member

```
member = await message.chat.get_member(message.from_user.id)
```

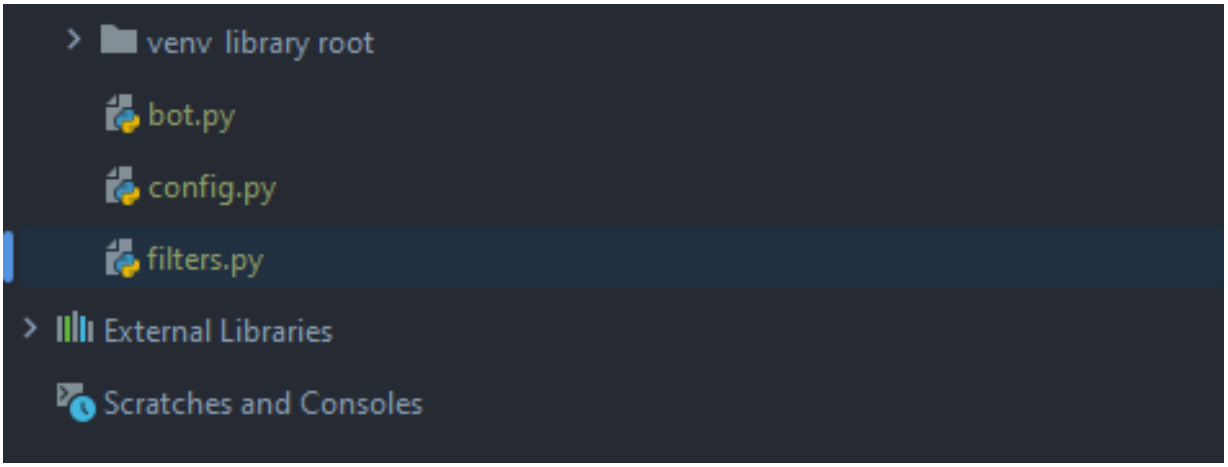
после получения id пользователя, мы возвращаем значение, которое вернула нам is_chat_admin(). Возвратом в данном случае будет true или false. По итогу если наша функция фильтр возвращает false, то обработчик пропускает данное сообщение.

Давайте теперь попробуем реализовать еще один собственный фильтр, который будет проверять написано ли сообщение в группе или это личное сообщение

Создадим заготовку бота

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

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)



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

Хорошим тоном является распределение логически связанных участков кода в разных файлах. Для наших фильтров создадим файл filters.py



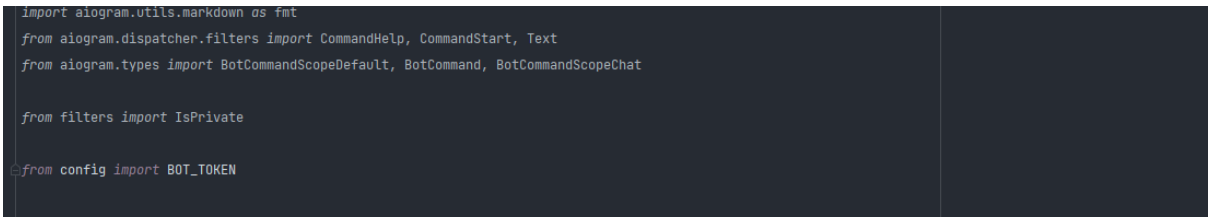
Теперь необходимо импортировать types для подсказок от среды разработки и BoundFilter

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

Далее мы создадим класс IsPrivate, который наследуется от BoundFilter. В классе создадим асинхронную функцию check, которая будет возвращать булево значение сравнения типа чата в котором написал пользователь и приватным чатом.

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


class IsPrivate(BoundFilter):
   async def check(self, message: types.Message):
       return message.chat.type == types.ChatType.PRIVATE
```

Теперь импортируем данный класс из filters.py в наш основной бот



Напишем обработчик

```
@dp.message_handler(IsPrivate(),  Command(commands='private'))
async def start_bot(message: types.Message):
   await message.answer('Личный чат')
```

Теперь добавим наш фильтр для групповых чатов

```
class IsGroup(BoundFilter):
    async def check(self, message: types.Message):
        return message.chat.type in (
            types.ChatType.GROUP,
            types.ChatType.SUPERGROUP,
        )
```

Этот фильтр работает аналогично. И напишем для него обработчик.

```
@dp.message_handler(IsGroup(), Command(commands='group'))
async def start_bot(message: types.Message):
```

```
   await message.answer('Групповой чат')
```

При вводе команды /private выводится сообщение - ‘Личный чат’, при вводе /group ничего не происходит соответственно, так как наша проверка возвращает false.

```
import logging
from aiogram import Bot, Dispatcher, executor, types
import aiogram.utils.markdown as fmt
from aiogram.dispatcher.filters import CommandHelp, CommandStart, Text,
Command
from aiogram.types import BotCommandScopeDefault, BotCommand,
BotCommandScopeChat

from filters import IsPrivate, IsGroup

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(IsPrivate(), Command(commands='private'))
async def start_bot(message: types.Message):
   await message.answer('Личный чат')


@dp.message_handler(IsGroup(), Command(commands='group'))
async def start_bot(message: types.Message):
   await message.answer('Групповой чат')


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

Сегодня мы немного вернемся к теме хендлеров и поговорим о error_handler(хендлере, перехватывающем ошибки). Конечно же мы можем обработать ошибки с помощью инструкции try..except, но в случае все же возникновения какой либо ошибки мы можем как то поработать с ошибкой.
Для начала создадим шаблон нашего бота

```
import logging
```

```
from aiogram import Bot, Dispatcher, executor, types

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)

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

Давайте создадим обработчик, в котором заведомо допустим пару ошибок и обработаем их с помощью try..except

```
@dp.message_handler(CommandStart())
async def bot_start(message: types.Message):
    non_existing_user = 111111
    # не попадает в error handler, так как обрабатывается
try..except
    try:
        await message.answer("Неправильно закрыт <b>тег<b>")
    except Exception as err:
        await message.answer(f'Не попало в error handler. Ошибка
{err}')

    # не попадает в error handler, так как обрабатывается
try..except
    try:
        await bot.send_message(chat_id=non_existing_user,
text='Несуществующий пользователь')
    except Exception as err:
        await message.answer(f'Не попало в error handler. Ошибка
{err}')

    # попадает в error handler
    await message.answer('Не сущетствует <fff>тега</fff>')
    logging.info('Это не выполнится, но бот не упадет')

    # все что ниже не выполнится
    await message.answer('hello')
```

Теперь пропишем наш error handler

```
@dp.errors_handler()
async def errors_handler(update, exception):
    """
    Exceptions handler. Catches all exceptions within task
factory tasks.
    :param update:
    :param exception:
    :return: stdout logging
    """
    from aiogram.utils.exceptions import (Unauthorized,
InvalidQueryID, TelegramAPIError,
                                          CantDemoteChatCreator,
MessageNotModified, MessageToDeleteNotFound,
                                          MessageTextIsEmpty,
RetryAfter,
                                          CantParseEntities,
MessageCantBeDeleted, BadRequest)

    if isinstance(exception, CantDemoteChatCreator):
        logging.debug("Can't demote chat creator")
        return True

    if isinstance(exception, MessageNotModified):
        logging.debug('Message is not modified')
        return True
    if isinstance(exception, MessageCantBeDeleted):
        logging.debug('Message cant be deleted')
        return True

    if isinstance(exception, MessageToDeleteNotFound):
        logging.debug('Message to delete not found')
        return True

    if isinstance(exception, MessageTextIsEmpty):
        logging.debug('MessageTextIsEmpty')
        return True

    if isinstance(exception, Unauthorized):
        logging.info(f'Unauthorized: {exception}')
        return True

    if isinstance(exception, InvalidQueryID):
        logging.exception(f'InvalidQueryID: {exception} \nUpdate:
{update}')
        return True

    if isinstance(exception, TelegramAPIError):
        logging.exception(f'TelegramAPIError: {exception}
\nUpdate: {update}')
        return True
    if isinstance(exception, RetryAfter):
        logging.exception(f'RetryAfter: {exception} \nUpdate:
{update}')
        return True
    if isinstance(exception, CantParseEntities):
```

```
        logging.exception(f'CantParseEntities: {exception}
\nUpdate: {update}')
        return True
    if isinstance(exception, BadRequest):
        logging.exception(f'CantParseEntities: {exception}
\nUpdate: {update}')
        return True
    logging.exception(f'Update: {update} \n{exception}')
```

В данной функции мы по сути получаем update и исключение, которое возникло. Далее мы импортируем все возможные исключения из aiogram и потом сравниваем его с нашим. Ошибку же выводим с помощью модуля logging для логирования.

Давайте добавим собственное условие.

```
  from aiogram.types import Update

    if isinstance(exception, CantParseEntities):
        await Update.get_current().message.answer(f'Попало в
error handler. CantParseEntities: {exception.args}')
        return True
```

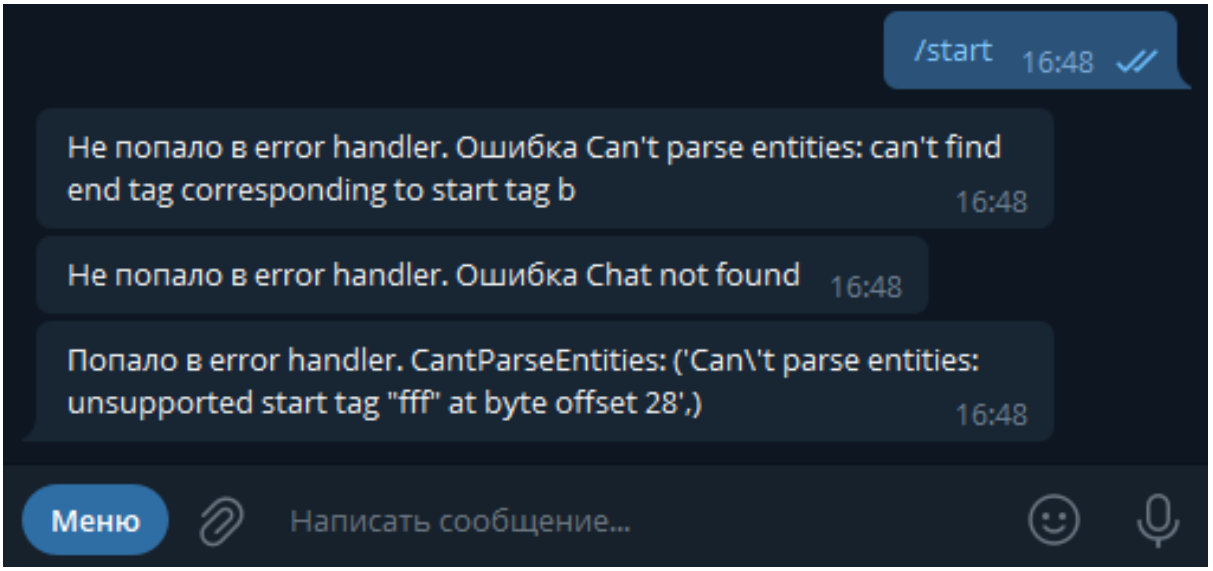
CantParseEntities - исключение, которое выбрасывается при направильных тегах.

Разберем строчку

await Update.get_current().message.answer(f'Попало в error handler. CantParseEntities: {exception.args}')

В данном случае мы говорим, что получаем текущий update, достаем из него message и отвечаем на него с помощью answer. Далее мы возвращаем True, чтобы “завершить данную ошибку”

В результате ошибка попала в error handler и мы отправили ее пользователю вместе с ошибками обработанными ранее.



Теперь давайте напишем бота, который заносит нас в базу данных и проверяет присутствуем ли мы в ней уже или нет.
Создадим заготовку бота

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

from filters import IsPrivate, IsGroup

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)



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

Теперь займемся базой данных.

```
class Database:
    def __init__(self, path_to_db='test.db'):
        self.path_to_db = path_to_db

    @property
```

```
    def connection(self):
        return sqlite3.connect(self.path_to_db)

    def execute(self, sql: str, parameters: tuple=None, fetchone=False,
fetchall=False, commit=False):
        if not parameters:
            parameters = tuple()
        connection = self.connection
        connection.set_trace_callback(logger)
        cursor = connection.cursor()
        data = None
        cursor.execute(sql, parameters)
        if commit:
            connection.commit()
        if fetchone:
            data = cursor.fetchone()
        if fetchall:
            data = cursor.fetchall()
        connection.close()

        return data

    def create_table_users(self):
        sql = """
        CREATE TABLE Users(
        id int NOT NULL,
        name varchar(255) NOT NULL,
        email varchar(255),
        PRIMARY KEY (id)
        );
        """
        return self.execute(sql)


    def add_user(self, id:int, name: str, email: str=None):
        sql = "INSERT INTO Users(id, name, email) VALUES  (?, ?, ?)"
        parameters = (id, name, email)
        self.execute(sql, parameters=parameters, commit=True)

    @staticmethod
    def format_args(sql, parameters: dict):
        sql += " AND ".join([
            f'{item} = ?' for item in parameters
        ])
        return sql, tuple(parameters.values())
    def  select_all_users(self):
        sql = "SELECT * FROM Users"
        return self.execute(sql, fetchall=True)

    def select_user(self, **kwargs):
        sql = "SELECT * FROM Users WHERE "
        sql, parameters = self.format_args(sql, kwargs)
        return self.execute(sql, parameters, fetchone=True)

    def count_users(self):
        return self.execute("SELECT COUNT(*) FROM Users;", fetchone=True)

    def update_email(self, email, id):
        sql = "UPDATE Users SET email=? WHERE id=?"
        return self.execute(sql, parameters=(email, id), commit=True)
```

```
    def delete_all_users(self):
        self.execute("DELETE FROM Users WHERE TRUE")
```

В данном коде мы создали класс Database, в котором конструкторе класса указываем в качестве аргумента путь до базы данных.
В методе connection возвращается соединение с нужной базой данных. Декоратор property в данном случае нужен лишь для удобства, чтобы мы могли обращаться к данному методу через точку без скобок
connection = self.connection

Метод execute выполняет необходимые запросы, которые передаются при вызове метода,также мы указали
fetchone=False, fetchall=False, commit=False
если нам надо получить из базы данных одно, все значения и если нам надо сохранить в базу данных
Строчка
connection.set_trace_callback(logger)
требуется всего лишь для вывода состояния обращения к базе данных.Добавим функцию logger для нашего бота

```
def logger(statement):
    print(f"""
_____________________________
EXECUTING
{statement}
_____________________________
""")
```

Также в классе Database мы создали методы для создания таблицы, добавления пользователя, выборку одного пользователя, всех пользователей, получения количества пользователей, обновления почты, удаления всех пользователей.

Создадим экземпляр класса базы данных и воспользуемся методом создания таблицы, предварительно обработав исключения

```
db = Database()

try:
    db.create_table_users()
    print('Таблица создана')
```

```
except Exception as e:
    print(e)
```

Теперь можно написать обработчик с фильтром команды /start, который будет проверять есть ли пользователь в базе данных, если нет, то добавит, также мы выведем количетсво зарегистрированных пользователей

```
@dp.message_handler(CommandStart())
async def bot_start(message: types.Message):
    name = message.from_user.full_name
    try:
        db.add_user(id=message.from_user.id, name=name)
    except sqlite3.IntegrityError as err:
        print(err)
    count_users = db.count_users()[0]
    await message.answer(
        '\n'.join([
            f'Привет, {message.from_user.full_name}!',
            'Ты был занесен в базу данных',
            f'В базе данных {count_users} пользователей'
        ])
    )
```

Таким образом мы можем сохранять необходимые данные в базе данных.
Полный код бота

```
import logging
import sqlite3

from aiogram import Bot, Dispatcher, executor, types
import aiogram.utils.markdown as fmt
from aiogram.dispatcher.filters import CommandHelp, CommandStart, Text,
Command
from aiogram.types import BotCommandScopeDefault, BotCommand,
BotCommandScopeChat

from filters import IsPrivate, IsGroup

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)
```

```
class Database:
   def __init__(self, path_to_db='test.db'):
       self.path_to_db = path_to_db

   @property
   def connection(self):
       return sqlite3.connect(self.path_to_db)

   def execute(self, sql: str, parameters: tuple = None, fetchone=False,
fetchall=False, commit=False):
       if not parameters:
           parameters = tuple()
       connection = self.connection
       connection.set_trace_callback(logger)
       cursor = connection.cursor()
       data = None
       cursor.execute(sql, parameters)
       if commit:
           connection.commit()
       if fetchone:
           data = cursor.fetchone()
       if fetchall:
           data = cursor.fetchall()
       connection.close()

       return data

   def create_table_users(self):
       sql = """
       CREATE TABLE Users(
       id int NOT NULL,
       name varchar(255) NOT NULL,
       email varchar(255),
       PRIMARY KEY (id)
       );
       """
       return self.execute(sql)

   def add_user(self, id: int, name: str, email: str = None):
       sql = "INSERT INTO Users(id, name, email) VALUES  (?, ?, ?)"
       parameters = (id, name, email)
       self.execute(sql, parameters=parameters, commit=True)

   @staticmethod
   def format_args(sql, parameters: dict):
       sql += " AND ".join([
           f'{item} = ?' for item in parameters
       ])
       return sql, tuple(parameters.values())

   def select_all_users(self):
       sql = "SELECT * FROM Users"
       return self.execute(sql, fetchall=True)

   def select_user(self, **kwargs):
       sql = "SELECT * FROM Users WHERE "
       sql, parameters = self.format_args(sql, kwargs)
       return self.execute(sql, parameters, fetchone=True)

   def count_users(self):
```

```
        return self.execute("SELECT COUNT(*) FROM Users;", fetchone=True)

    def update_email(self, email, id):
        sql = "UPDATE Users SET email=? WHERE id=?"
        return self.execute(sql, parameters=(email, id), commit=True)

    def delete_all_users(self):
        self.execute("DELETE FROM Users WHERE TRUE")


def logger(statement):
    print(f"""
_____________________________
EXECUTING
{statement}
_____________________________
""")


db = Database()

try:
    db.create_table_users()
    print('Таблица создана')
except Exception as e:
    print(e)


@dp.message_handler(CommandStart())
async def bot_start(message: types.Message):
    name = message.from_user.full_name
    try:
        db.add_user(id=message.from_user.id, name=name)
    except sqlite3.IntegrityError as err:
        print(err)
    count_users = db.count_users()[0]
    await message.answer(
        '\n'.join([
            f'Привет, {message.from_user.full_name}!',
            'Ты был занесен в базу данных',
            f'В базе данных {count_users} пользователей'
        ])
    )


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