Материалы к занятию

При работе бота неизбежно возникновение различных ошибок, связанных не с кодом, а с внешними событиями. Простейший пример: попытка отправить сообщение пользователю, заблокировавшему бота. Чтобы не оборачивать каждый вызов в try..except, в aiogram существует специальный хэндлер для исключений, связанных с Bot API.
Рассмотрим следующий пример кода, имитирующий задержку перед ответом пользователю:

```
@dp.message_handler(commands="block")
async def cmd_block(message: types.Message):
    await asyncio.sleep(10.0)
    await message.reply("Вы заблокированы")
```

За эти 10 секунд пользователь может успеть заблокировать бота со своей стороны и попытка вызвать метод reply приведёт к появлению исключения BotBlocked. Напишем специальный хэндлер для этого исключения

```
from aiogram.utils.exceptions import BotBlocked

@dp.errors_handler(exception=BotBlocked)
async def error_bot_blocked(update: types.Update, exception: BotBlocked):
    # Update: объект события от Telegram. Exception: объект исключения
    # Здесь можно как-то обработать блокировку, например, удалить
пользователя из БД
    print(f"Меня заблокировал пользователь!\nСообщение: {update}\nОшибка:
{exception}")

    # Такой хэндлер должен всегда возвращать True,
    # если дальнейшая обработка не требуется.
    return True
```

Аналогично пишутся обработчики и на другие исключения. Таким образом, если одна и та же непредвиденная ситуация может возникнуть в различных хэндлерах, то можно вынести её обработку в отдельный хэндлер ошибок. Кода будет меньше, а оставшийся станет читабельнее.

Для того, чтобы сделать код чище и читабельнее, aiogram расширяет возможности стандартных объектов Telegram. Например, вместо bot.send_message(...) можно написать message.answer(...) или message.reply(...). В последних двух случаях не нужно подставлять chat_id, подразумевается, что он такой же, как и в исходном сообщении.
Разница между answer и reply простая: первый метод просто отправляет сообщение в тот же чат, второй делает "ответ" на сообщение из message:

```
@dp.message_handler(commands="answer")
async def cmd_answer(message: types.Message):
    await message.answer("Это простой ответ")


@dp.message_handler(commands="reply")
async def cmd_reply(message: types.Message):
    await message.reply('Это ответ с "ответом"')
```

Более того, для большинства типов сообщений есть вспомогательные методы вида "answer_{type}" или "reply_{type}", например:

```
from aiogram import Bot, types
from aiogram.dispatcher import Dispatcher
from aiogram.utils import executor



bot = Bot(token='Ваш токен')
dp = Dispatcher(bot)


@dp.message_handler(commands="dice")
async def cmd_dice(message: types.Message):
   await message.answer_dice(emoji="🎲")


if __name__ == '__main__':
   executor.start_polling(dp)
```

Python является интерпретируемым языком с сильной, но динамической типизацией, поэтому встроенная проверка типов, как, например, в C++ или Java, отсутствует. Однако начиная с версии 3.5 в языке появилась поддержка подсказок типов, благодаря которой различные чекеры и IDE вроде PyCharm анализируют типы используемых значений и подсказывают программисту, если он передаёт что-то не то. В данном случае подсказка types.Message сообщает PyCharm-у, что переменная message имеет тип Message, описанный в модуле types библиотеки aiogram. Благодаря этому IDE может на лету подсказывать атрибуты и функции.

При вызове команды /dice бот отправит в тот же чат игральный кубик. Разумеется, если его надо отправить в какой-то другой чат, то придётся по-старинке вызывать await bot.send_dice(...). Но объект bot (экземпляр класса Bot) может быть недоступен в области видимости конкретной функции. К счастью, объект бота доступен во всех типах апдейтов: Message, CallbackQuery, InlineQuery и т.д. Предположим, вы хотите по команде /dice отправлять кубик не в тот же чат, а в канал с ID -100123456789. Перепишем предыдущую функцию:

```
from aiogram import Bot, types
from aiogram.dispatcher import Dispatcher
from aiogram.utils import executor



bot = Bot(token='Ваш токен')
dp = Dispatcher(bot)


@dp.message_handler(commands="dice")
async def cmd_dice(message: types.Message):
   await message.bot.send_dice(-100123456789, emoji="🎲")


if __name__ == '__main__':
   executor.start_polling(dp)
```

Всё хорошо, но если вдруг вы захотите поделиться с кем-то кодом, то придётся каждый раз помнить об удалении из исходников токена бота, иначе придётся его перевыпускать у @BotFather. Чтобы обезопасить себя, давайте перестанем указывать токен прямо в коде, а вынесем его как переменную окружения.
Замените следующие строчки из начала файла:

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

bot = Bot(token="Ваш токен"
```

на эти:

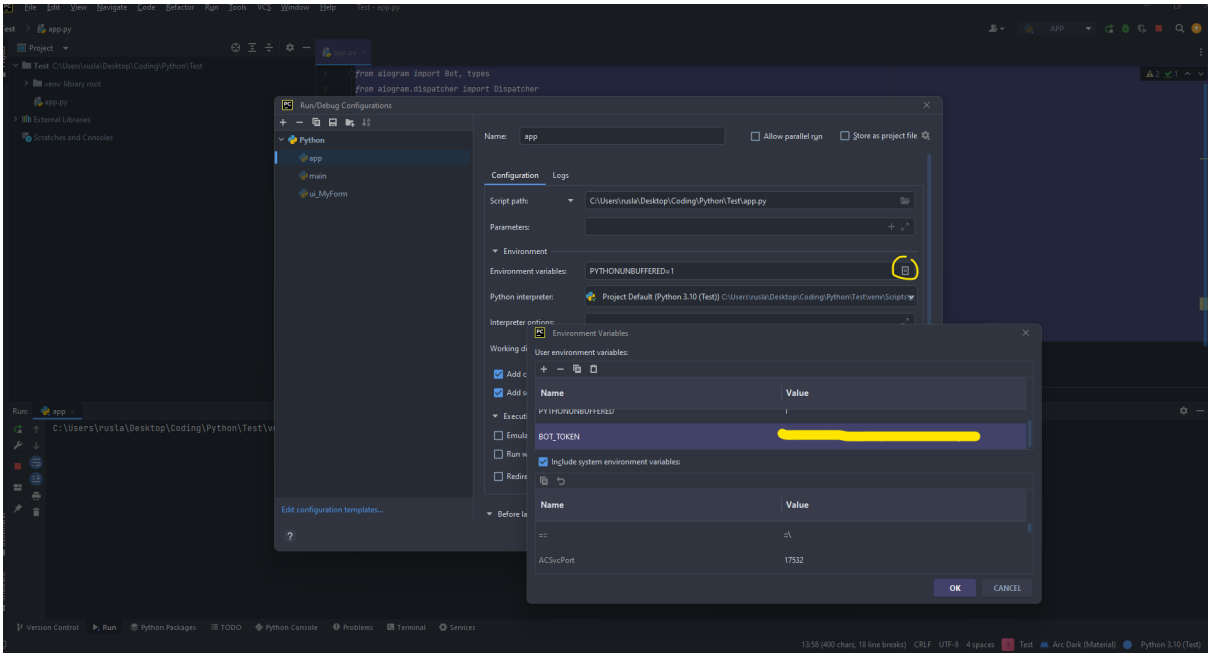
```
import logging
from aiogram import Bot, Dispatcher, executor, types
from os import getenv
from sys import exit

bot_token = getenv("BOT_TOKEN")
if not bot_token:
    exit("Error: no token provided")

bot = Bot(token=bot_token)
```

Но теперь ваш бот не запустится, т.к. будет сразу завершаться с ошибкой Error: no token provided. Чтобы передать переменную окружения в PyCharm, откройте сверху раздел Run ->

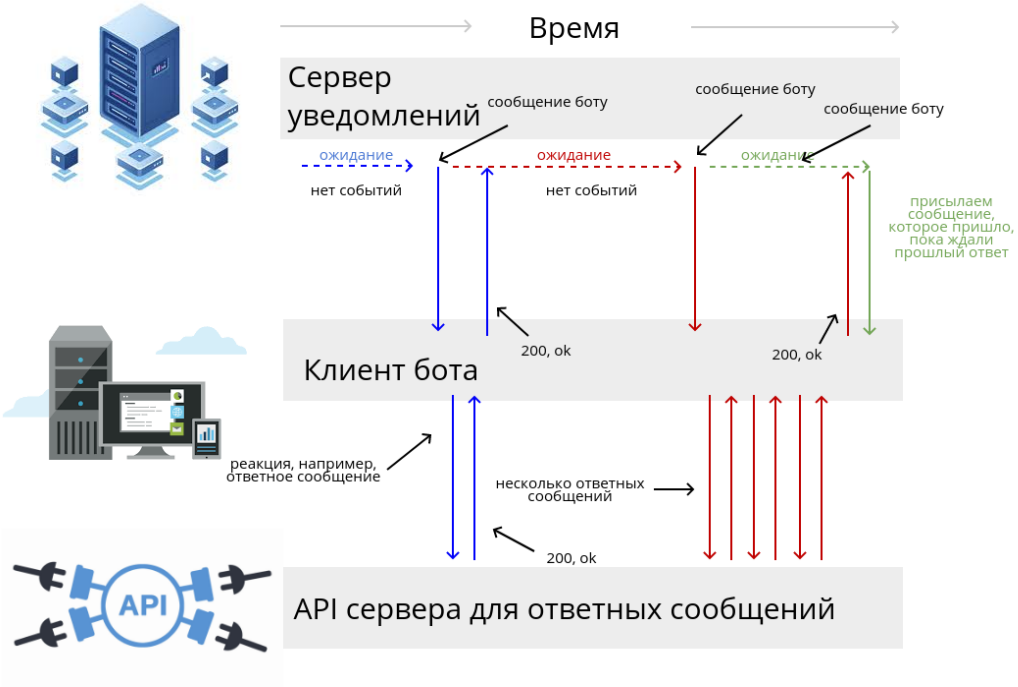
Edit Configurations и добавьте в окне Environment Variables переменную с именем BOT_TOKEN и значением токена.



Запустите снова бота и убедитесь, что он работает. Получившийся код можно смело сохранять в PyCharm в File Templates.

Разрабатывая телеграм ботов мы часто не задумываемся какое количество человек им будет пользоваться. До этого мы писали примеры с использованием технологии pooling.

Polling — это когда клиент всё время спришвает у сервера “есть что-нибудь новенькое?”, а сервер присылает события (например, новые сообщения) или пишет, что ничего не происходило.
Webhook — это то же самое, только наоборот. Теперь если что-то случается (например, новое сообщение) — сервер сам пишет клиенту



Давайте рассмотрим схему подробнее. Есть 2 основных действующих лица: клиент и сервер уведомлений. Как только вашему боту написал пользователь — сервер уведомлений шлёт вам JSON-объект, где указывает кто писал, что писал и так далее. Дальше клиент сам решает, как на это сообщение отреагировать. Он может прислать одно сообщение, может несколько, а может вообще промолчать. В конце он отвечает серверу “Ок, 200”, обозначая этим, что запрос обработан.
Чтобы клиент мог писать в ответ, обычно создаётся отдельное API, где реализованы запросы на отправку сообщений и так далее. API и сервер уведомлений — это обычно одна та же соцсеть, просто разные микрсервисы. Один занимается только рассылкой уведомлений, другой — полноценное API для разработчиков.

Обе технологии занимаются одним и тем же: приносят вам сообщения о событиях. Тем не менее, между ними огромная разница: polling легко реализуется обычной библиотекой requests, в то время как для webhook обязательно нужен сайт, который будет принимать и обрабатывать запросы от сервера уведомлений.

Взамен webhook снижает нагрузку на сеть: вы обмениваетесь сообщениями с сервером уведомлений только “по делу”, в то время как polling постоянно меняется пустыми сообщениями:
Клиент: Обновления есть?
Сервер: Обновлений нет.
Клиент: А теперь?
Сервер: Всё ещё нет
...

При этом вы не теряете мгновенность ответа, как при long polling, потому что сервер по-прежнему пишет вам сразу, как только ему написал пользователь.

Вот стандартный шаблон простого бота с вебхуками

```
import logging

from aiogram import Bot, types
from aiogram.contrib.middlewares.logging import LoggingMiddleware
from aiogram.dispatcher import Dispatcher
from aiogram.dispatcher.webhook import SendMessage
from aiogram.utils.executor import start_webhook

API_TOKEN = 'Ваш  токен'

# webhook settings
WEBHOOK_HOST = 'адрес сайта'
WEBHOOK_PATH = '/path/to/api'
WEBHOOK_URL = f"{WEBHOOK_HOST}{WEBHOOK_PATH}"

# webserver settings
WEBAPP_HOST = 'localhost'  # or ip
WEBAPP_PORT = 3001

logging.basicConfig(level=logging.INFO)

bot = Bot(token=API_TOKEN)
dp = Dispatcher(bot)
dp.middleware.setup(LoggingMiddleware())


@dp.message_handler()
async def echo(message: types.Message):
   # Regular request
   # await bot.send_message(message.chat.id, message.text)

   # or reply INTO webhook
   return SendMessage(message.chat.id, message.text)


async def on_startup(dp):
   await bot.set_webhook(WEBHOOK_URL)
   # insert code here to run it after start


async def on_shutdown(dp):
   logging.warning('Shutting down..')

   # insert code here to run it before shutdown

   # Remove webhook (not acceptable in some cases)
   await bot.delete_webhook()

   # Close DB connection (if used)
   await dp.storage.close()
   await dp.storage.wait_closed()

   logging.warning('Bye!')
```

```
if __name__ == '__main__':
    start_webhook(
        dispatcher=dp,
        webhook_path=WEBHOOK_PATH,
        on_startup=on_startup,
        on_shutdown=on_shutdown,
        skip_updates=True,
        host=WEBAPP_HOST,
        port=WEBAPP_PORT,
    )
```

Как мы видим из данного шаблона, при запуске бота, мы используем не start_polling(), а start_webhook(), в который помимо диспетчера передаем настройки хоста и порта. Также мы указываем, что при падении бот будет перезапускаться.

В примере дополнительно указывается такие понятия как мидлвари и работу с базой данных, которые мы затронем чуть позже. Ввиду того, что для использования вебхуков подразумевает использование стороннего сайта, мы рассмотрим его на примере сервиса Heroku. На данный момент возможны некоторые проблемы с регистрацией на сервисе.

setWebhook - Используйте этот метод, чтобы указать URL-адрес и получать входящие обновления через исходящий webhook. Всякий раз, когда появляется обновление для бота, мы отправляем запрос HTTPS POST на указанный URL-адрес, содержащий сериализованный в JSON файл aiogram.types.update.Update. В случае неудачного запроса мы откажемся от него после разумного количества попыток. Возвращает значение True при успешном выполнении. Если вы хотите убедиться, что запрос Webhook поступает из Telegram, мы рекомендуем использовать секретный путь в URL-адресе, например *https://www.example.com/<токен>*. Поскольку никто другой не знает токен вашего бота, вы можете быть уверены, что это мы.

url - URL-адрес HTTPS для отправки обновлений. Используйте пустую строку для удаления интеграции с webhook
certificate - Загрузите свой сертификат открытого ключа, чтобы можно было проверить используемый корневой сертификат.
ip_address - Фиксированный IP-адрес, который будет использоваться для отправки запросов webhook вместо IP-адреса, разрешенного через DNS
max_connections - Максимально допустимое количество одновременных HTTPS-подключений к webhook для доставки обновлений - 1-100. Значение по умолчанию равно 40. Используйте более низкие значения, чтобы ограничить нагрузку на сервер вашего бота, и более высокие значения, чтобы увеличить пропускную способность вашего бота.

allowed_updates - Сериализованный в формате JSON список типов обновлений, которые вы хотите, чтобы ваш бот получал. Например, укажите [„сообщение“, „edited_channel_post“, „callback_query“], чтобы получать обновления только этих типов. Укажите пустой список для получения всех типов обновлений, кроме chat_member (по умолчанию). Если не указано, будет использоваться предыдущая настройка.
drop_pending_updates - Передайте значение True, чтобы удалить все ожидающие обновления