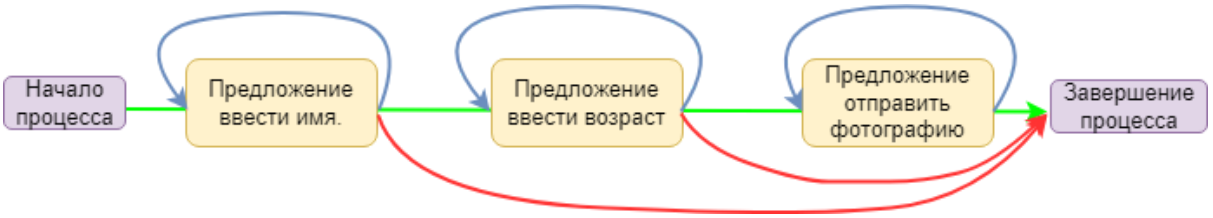
**Теоретические материалы к занятию 5**

Сегодня мы поговорим о, пожалуй, самой важной возможности ботов: о системе диалогов. К сожалению, далеко не все действия в боте можно выполнить за одно сообщение или команду. Предположим, есть бот для знакомств, где при регистрации нужно указать имя, возраст и отправить фотографию с лицом. Можно, конечно, попросить пользователя отправить фотографию, а в подписи к ней указать все данные, но это неудобно для обработки и запроса повторного ввода.
Теперь представим пошаговый ввод данных, где в начале бот «включает» режим ожидания определенной информации от конкретного юзера, далее на каждом этапе проверяет вводимые данные, а по команде /cancel прекращает ожидать очередной шаг и возвращается в основной режим




Зелёной стрелкой обозначен процесс перехода по шагам без ошибок, синие стрелки означают сохранение текущего состояния и ожидание повторного ввода (например, если юзер указал, что ему 250 лет, следует запросить возраст заново), а красные показывают выход из всего процесса из-за команды /cancel или любой другой, означающей отмену. Процесс со схемы выше в теории алгоритмов называется конечным автоматом или машиной состояний.

В oaiogram встроена поддержка различных бэкендов для хранения состояний между перезапусками бота, а помимо, собственно, состояний можно хранить произвольные данные, например, вышеописанные имя и возраст для последующего использования где-либо. Мы будем пользоваться самым простейшим бэкендом MemoryStorage, который хранит все данные в оперативной памяти. Он идеально подходит для примеров, но не рекомендуется использовать его в реальных проектах, т.к. MemoryStorage хранит все данные в оперативной памяти без сброса на диск. Также стоит отметить, что конечные автоматы можно использовать не только с обработчиками сообщений (message_handler, edited_message_handler), но также с колбэками и инлайн-режимом.
В качестве примера мы напишем имитатор заказа еды и напитков в кафе.

Для начала создадим наш шаблон бота.

```
import asyncio
import logging

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

```
from config import BOT_TOKEN
from filters import AdminFilter


bot = Bot(token=BOT_TOKEN, parse_mode=types.ParseMode.HTML)
dp = Dispatcher(bot, storage=MemoryStorage())


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

Импортируем необходимые модули для хранения данных и непосредственно саму “машину состояний”

```
from aiogram.dispatcher import FSMContext
from aiogram.dispatcher.filters.state import State, StatesGroup
```

Укажем списки блюд и их размеров (в реальной жизни эта информация может динамически подгружаться из какой-либо БД)

```
available_food_names = ["суши", "спагетти", "хачапури"]
available_food_sizes = ["маленькую", "среднюю", "большую"]
```

Теперь опишем все возможные “состояния” конкретного процесса (выбор еды). На словах можно описать так: пользователь вызывает команду /food, бот отвечает сообщением с просьбой выбрать блюдо и встаёт в состояние *ожидает выбор блюда* для конкретного пользователя. Как только юзер делает выбор, бот, находясь в этом состоянии, проверяет корректность ввода, а затем принимает решение, запросить ввод повторно (без смены состояния) или перейти к следующему шагу *ожидает выбор размера порции*. Когда пользователь и здесь вводит корректные данные, бот отображает итоговый результат (содержимое заказа) и сбрасывает состояние. Чуть позже мы научимся делать принудительный сброс состояния на любом этапе командой /cancel.

Итак, перейдём непосредственно к описанию состояний. Желательно их указывать именно в том порядке, в котором предполагается переход пользователя, это позволит немного упростить код. Для хранения состояний необходимо создать класс, наследующийся от класса StatesGroup, внутри него нужно создать переменные, присвоив им экземпляры класса State:

```
class OrderFood(StatesGroup):
   waiting_for_food_name = State()
   waiting_for_food_size = State()
```

Напишем обработчик первого шага, реагирующий на команду /food

```
async def food_start(message: types.Message):
    keyboard = types.ReplyKeyboardMarkup(resize_keyboard=True)
    for name in available_food_names:
        keyboard.add(name)
    await message.answer("Выберите блюдо:", reply_markup=keyboard)
    await OrderFood.waiting_for_food_name.set()
```

В последней строке мы явно говорим боту встать в состояние waiting_for_food_name из группы OrderFood. Следующая функция будет вызываться только из указанного состояния, сохранять полученный от пользователя текст (если он валидный) и переходить к следующему шагу:

```
async def food_chosen(message: types.Message, state: FSMContext):
    if message.text.lower() not in available_food_names:
        await message.answer("Пожалуйста, выберите блюдо,
используя клавиатуру ниже.")
        return
    await state.update_data(chosen_food=message.text.lower())

    keyboard = types.ReplyKeyboardMarkup(resize_keyboard=True)
    for size in available_food_sizes:
        keyboard.add(size)
    # Для последовательных шагов можно не указывать название
состояния, обходясь next()
    await OrderFood.next()
    await message.answer("Теперь выберите размер порции:",
reply_markup=keyboard)
```

Разберём некоторые строки из блока выше отдельно. В определении функции food_chosen появился второй аргумент state типа FSMContext. Через него можно получить данные от FSM-бэкенда. В предыдущей функции food_start() никакие такие данные не требовались, поэтому аргумент был пропущен. Далее производится проверка текста от пользователя. Если он ввёл произвольный текст, а не использовал кнопки, то необходимо сообщить об ошибке и досрочно завершить выполнение функции. При этом состояние пользователя останется тем же и бот снова будет ожидать выбор блюда.
К моменту перехода к строке
*await state.update_data(chosen_food=message.text.lower())*
мы уже уверены, что пользователь указал корректное название блюда, поэтому можно спокойно сохранить полученный текст в хранилище данных FSM. Концептуально это словарь, поэтому воспользуемся функцией update_data() и сохраним текст сообщения под ключом chosen_food и со значением message.text.lower().

В строке
*await OrderFood.next()*
мы готовы продвинуть пользователя на следующий шаг, поэтому просто вызываем метод next() у класса OrderFood. Именно для использования next() ранее предлагалось объявлять шаги в нужном порядке. В противном случае пришлось бы указывать целиком await OrderFood.waiting_for_food_size.set().
Осталось реализовать последнюю функцию, которая отвечает за получение размера порции (с аналогичной проверкой ввода) и вывод результатов пользователю:

```
async def food_size_chosen(message: types.Message, state:
FSMContext):
   if message.text.lower() not in available_food_sizes:
       await message.answer("Пожалуйста, выберите размер порции,
используя клавиатуру ниже.")
       return
   user_data = await state.get_data()
   await message.answer(f"Вы заказали {message.text.lower()}
порцию {user_data['chosen_food']}.\n"
                        f"Попробуйте теперь заказать напитки:
/drinks", reply_markup=types.ReplyKeyboardRemove())
   await state.finish()
```

На что стоит обратить внимание: во-первых, получить хранимые данные из FSM можно методом get_data() у state. Во-вторых, т.к. это словарь, то и извлечение содержимого аналогично. В-третьих, вызов метода finish() сбрасывает не только состояние, но и хранящиеся данные. Если надо сбросить только состояние, Используем
*await state.reset_state(with_data=False)*

Наконец, напишем обычную функцию для регистрации вышестоящих обработчиков, на вход она будет принимать диспетчер. Значение состояния "*" при регистрации food_start() означает срабатывание при любом состоянии. Грубо говоря, если пользователь находится на шаге *выбор размера порции*, но решает начать заново и вводит /food, бот вызовет food_start() и начнёт весь процесс заново.

```
def register_handlers_food(dp: Dispatcher):
    dp.register_message_handler(food_start, commands="food",
state="*")
    dp.register_message_handler(food_chosen,
state=OrderFood.waiting_for_food_name)
    dp.register_message_handler(food_size_chosen,
state=OrderFood.waiting_for_food_size)
```

Шаги для выбора напитков делаются совершенно аналогично.

**Учитель:** Давайте рассмотрим возможность сброса состояний.Реализуем обработчики команды /start и действия «отмены». Первая должна показывать некий приветственный/справочный текст, а вторая просто пишет "действие отменено". Обе функции сбрасывают состояние и данные и убирают обычную клавиатуру, если вдруг она есть:

```
async def cmd_start(message: types.Message, state: FSMContext):
    await state.finish()
    await message.answer(
        "Выберите, что хотите заказать: напитки (/drinks) или
блюда (/food).",
        reply_markup=types.ReplyKeyboardRemove()
    )

async def cmd_cancel(message: types.Message, state: FSMContext):
    await state.finish()
    await message.answer("Действие отменено",
reply_markup=types.ReplyKeyboardRemove())
```

Зарегистрируем эти два обработчика:

```
def register_handlers_common(dp: Dispatcher):
    dp.register_message_handler(cmd_start, commands="start",
state="*")
    dp.register_message_handler(cmd_cancel, commands="cancel",
state="*")
    dp.register_message_handler(cmd_cancel, Text(equals="отмена",
ignore_case=True), state="*")
```

Почему строк три, а не две? Дело в том, что один и тот же обработчик можно вызвать по разным событиям. Вот мы и зарегистрируем функцию cmd_cancel() для вызова как по команде /cancel, так и по отправке сообщения "Отмена" (в любом регистре). К слову, если вы навешиваете декораторы напрямую на функцию, то выглядеть это будет следующим образом:

```
@dp.message_handler(commands="cancel", state="*")
@dp.message_handler(Text(equals="отмена", ignore_case=True),
state="*")
async def cmd_cancel(message: types.Message, state: FSMContext):
```

Теперь мы можем хранить состояния(правда в памяти) и можем реализовывать некую систему диалогов. Также очень часто машина состояний используется для различных опросов и викторин, чтобы сохранять шаги пользователя. Давайте рассмотрим пример бота, проводящего опрос.

```
import logging

from aiogram import Bot, Dispatcher, executor, types
from aiogram.contrib.fsm_storage.memory import MemoryStorage
from aiogram.dispatcher import FSMContext
from aiogram.dispatcher.filters.state import State, StatesGroup
from config import BOT_TOKEN

bot = Bot(token=BOT_TOKEN, parse_mode=types.ParseMode.HTML)
storage = MemoryStorage()
dp = Dispatcher(bot, storage=storage)

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

Добавим команды по умолчанию.

```
async def set_default_commands(dp):
   await dp.bot.set_my_commands(
       [
           types.BotCommand("start", "Запустить бота"),
           types.BotCommand("help", "Вывести справку"),
           types.BotCommand("onstarttest", "Пройти первый
опрос"),
       ]
   )


async def on_startup(dispatcher):
   # Устанавливаем дефолтные команды
   await set_default_commands(dispatcher)
```

Добавим необходимые состояния

```
class CallbackOnStart(StatesGroup):
    Q1 = State()
    Q2 = State()
    Q3 = State()
```

Далее создадим клавиатуру.

```
def towers():
   list_button_name = [['Москва', 'Санкт Петербург', 'Нижний
Новгород', 'Ростов'],
```

```
                        ['Новосибирск', 'Екатеринбург', 'Казань',
'Челябинск']]

   buttons_list = []
   for item in list_button_name:
       l = []
       for i in item:
           l.append(InlineKeyboardButton(text=i,
callback_data=i))
       buttons_list.append(l)

   keyboard_inline_buttons =
InlineKeyboardMarkup(inline_keyboard=buttons_list)
   return keyboard_inline_buttons
```

В данном случае мы использовали генератор кнопок, иначе нам бы пришлось каждую кнопку прописывать руками.Функция возвращает массив кнопок для работы с ними.

Перейдем к обработчикам. Первым делом мы проверяем проходил ли пользователь опрос раньше. Для этого открываем базу данных в JSON-файле и перебираем все элементы в поисках ID пользователя. Если его нет, запускаем тест, переводя пользователя в FSM. Если пользователь уже есть в базе - отправляем сообщение, что он проходил опрос ранее.

```
@dp.message_handler(Command('onstarttest'))
async def on_start_test(message: types.Message):
   id = message.from_user.id
   with open('users_test_one.json', encoding='utf-8') as
json_file:
       data = json.load(json_file)
       for i in data:
           if int(i) == id:
               user = False
               break
           else:
               user = True
   if user:
       await message.answer("Описание опросника")
       await message.answer('Вопрос №1\nСколько вам
лет?\nНапишите ответ (только число)',
                            reply_markup=ReplyKeyboardRemove())
       await CallbackOnStart.Q1.set()
   else:
       await message.answer(text="Вы уже проходили тест")
```

Следующий хендлер срабатывает при состоянии Q1. Впоследствии генерируется набор кнопок. После ответа пользователя, полученные данные сохраняются в FSM кэш. После всех действий бот отправляет новый вопрос и переводит человека в новую фазу состояния:

```
@dp.message_handler(state=CallbackOnStart.Q1)
async def tower(message: types.Message, state: FSMContext):
   b = towers()
   answer = message.text
   await state.update_data(name=answer)
   await message.answer(text="Вопрос №2\nВ каком городе вы
живете?\nВыберите ответ из предложенных",
                        reply_markup=b)
   await CallbackOnStart.next()
```

Answer - это данные из скрытого ответа inline кнопки. Data запрашивает все данные, сохраненные в FSM. Дальше мы готовим информацию к сохранению в JSON-файл (под ID пользователя мы добавляем полученные ответы). Когда всё готово, отправляем пользователю его выбранные ответы:

```
@dp.callback_query_handler(state=CallbackOnStart.Q2)
async def end(call: types.Message, state: FSMContext):
   answer = call.data
   await state.update_data(full_name=call.from_user.full_name)
   await state.update_data(repost=answer)
   data = await state.get_data()
   user = {call.from_user.id: data}
   text = []
   for i in data:
       text.append(f'{data[i]}\n')
   await call.message.answer(text="Ваши ответы:",
reply_markup=ReplyKeyboardRemove())
   await call.message.answer('\n'.join(text))
   with open('users_test_one.json', encoding='utf-8') as file:
       data = json.load(file)
       data.update(user)
       with open('users_test_one.json', 'w', encoding='utf-8') as
outfile:
           json.dump(data, outfile, indent=4, ensure_ascii=False)
   await state.finish()
```