

Урок 4. Реализуем каталог товаров

ОГЛАВЛЕНИЕ

[СОЗДАЕМ КАТАЛОГ ТОВАРОВ 3](#_Toc79588292)

[НАСТРАИВАЕМ РАБОТУ КНОПКИ ВЫБОРА ТОВАРОВ ПЕРВОЙ НОМЕНКЛАТУРНОЙ ГРУППЫ 5](#_Toc79588293)

[ВЫВОДИМ ИЗ БАЗЫ ДАННЫХ ТОВАРЫ ПЕРВОЙ НОМЕНКЛАТУРНОЙ ГРУППЫ 7](#_Toc79588294)

[ПОДКЛЮЧАЕМ БАЗУ ДАННЫХ 12](#_Toc79588295)

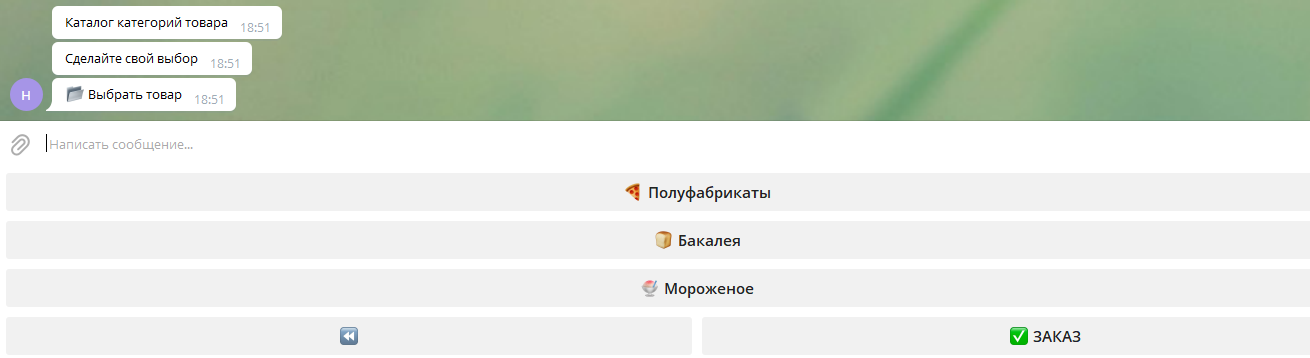
[ЗАКЛЮЧЕНИЕ 16](#_Toc79588296)

[КЛЮЧЕВЫЕ ВЫВОДЫ 16](#_Toc79588297)



# СОЗДАЕМ КАТАЛОГ ТОВАРОВ

Необходимо, чтобы при нажатии на кнопку «Выбрать товар», мы бы получили, например, такую картинку:



Необходимо реализовать обработчик для кнопки «Выбрать товар».

Дополним модуль **handler\_all\_text.py**. Реализуем перехват события нажатия указанной выше кнопки.

**Листинг 1. step\_4/handlers/handler\_all\_text.py**

**if** message.text == config.KEYBOARD[**'CHOOSE\_GOODS'**]:  
 self.pressed\_btn\_category(message)

И добавим обработчик:

**def** pressed\_btn\_category(self, message):  
 *"""  
 Обработка события нажатия на кнопку 'Выбрать товар'. А точне  
 это выбор категории товаров  
 """* self.bot.send\_message(message.chat.id, **"Каталог категорий товара"**,  
 reply\_markup=self.keybords.remove\_menu())  
 self.bot.send\_message(message.chat.id, **"Сделайте свой выбор"**,  
 reply\_markup=self.keybords.category\_menu())

Также дополним модуль **markup.py**.

**Листинг 2. step\_4/markup/markup.py**

@staticmethod  
**def** remove\_menu():  
 *"""  
 Удаляет меню  
 """* **return** ReplyKeyboardRemove()  
  
**def** category\_menu(self):  
 *"""  
 Создает разметку кнопок в меню категорий товара и возвращает разметку  
 """* self.markup = ReplyKeyboardMarkup(**True**, **True**, row\_width=1)  
 self.markup.add(self.set\_btn(**'SEMIPRODUCT'**))  
 self.markup.add(self.set\_btn(**'GROCERY'**))  
 self.markup.add(self.set\_btn(**'ICE\_CREAM'**))  
 self.markup.row(self.set\_btn(**'<<'**), self.set\_btn(**'ORDER'**))  
 **return** self.markup

При нажатии на кнопку «Выбрать товар» удаляем старую разметку:

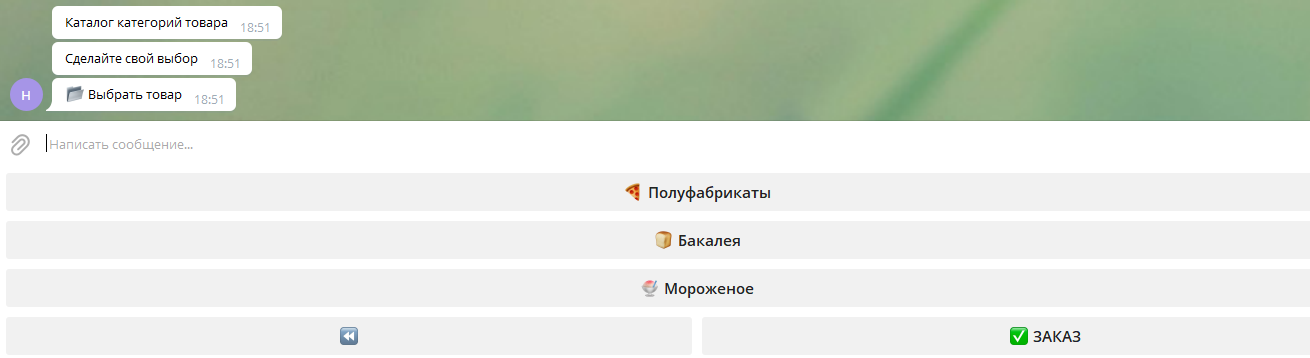
reply\_markup=self.keybords.remove\_menu()

И сразу же создаем, и добавляем новую:

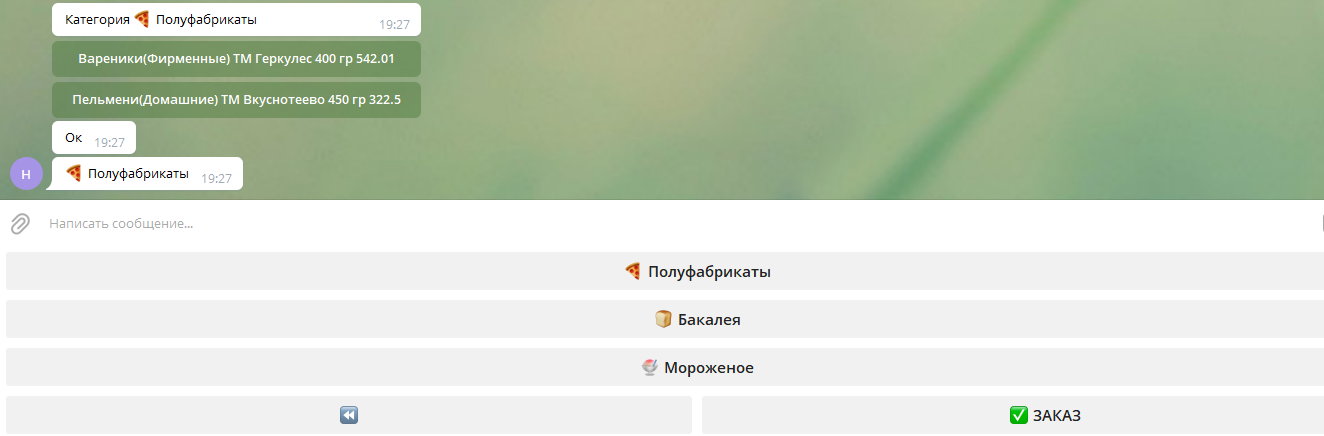
reply\_markup=self.keybords.category\_menu()

В которой есть кнопки для перехода к товарам трех номенклатурных позиций: Полуфабрикаты, Бакалея, Мороженое. А также кнопки «Назад» и «Заказ».

Запустим проект, все работает.



# НАСТРАИВАЕМ РАБОТУ КНОПКИ ВЫБОРА ТОВАРОВ ПЕРВОЙ НОМЕНКЛАТУРНОЙ ГРУППЫ

Например, мы хотим выбрать товары из группы «Полуфабрикаты». И получить что-то вроде:

Для этого дополняем код модуля **handler\_all\_text.py**. В модуль добавим обработчик выбора товаров номенклатурной группы.

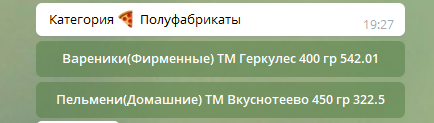
**Листинг 3. step\_4/handlers/ handler\_all\_text.py**

**def** pressed\_btn\_product(self, message, product):  
 *"""  
 Обработка события нажатия на кнопку 'Выбрать товар'. А точнее  
 это выбор товара из категории  
 """* self.bot.send\_message(message.chat.id, **'Категория '** +  
 config.KEYBOARD[product],  
 reply\_markup=  
 self.keybords.set\_select\_category(  
 config.CATEGORY[product]))  
 self.bot.send\_message(message.chat.id, **"Ок"**,  
 reply\_markup=self.keybords.category\_menu())

…

*# \*\*\*\*\*\*\*\*\*\* меню (категории товара, ПФ, Бакалея, Мороженое)\*\*\*\*\*\*\*\*\*\****if** message.text == config.KEYBOARD[**'SEMIPRODUCT'**]:  
 self.pressed\_btn\_product(message,**'SEMIPRODUCT'**)  
  
**if** message.text == config.KEYBOARD[**'GROCERY'**]:  
 self.pressed\_btn\_product(message,**'GROCERY'**)  
  
**if** message.text == config.KEYBOARD[**'ICE\_CREAM'**]:  
 self.pressed\_btn\_product(message,**'ICE\_CREAM'**)

Итак, мы нажимаем на кнопку «Полуфабрикаты». Нужно показать пользователю, что выбрана эта категория.

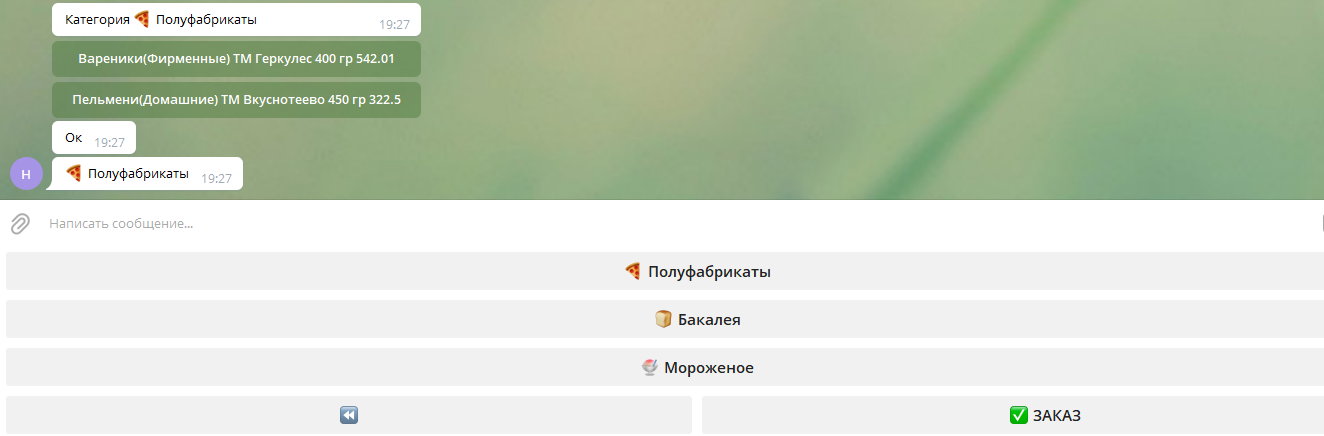


Показываем:

self.bot.send\_message(message.chat.id, **'Категория '** + config.KEYBOARD[product],  
 reply\_markup=self.keybords.set\_select\_category(config.CATEGORY[product]))

А также выводим наше главное меню, тем самым показываем, что все в порядке:

self.bot.send\_message(message.chat.id, **"Ок"**,  
 reply\_markup=self.keybords.category\_menu())



Метод **category\_menu()** (он обеспечивает показ списка категорий) мы уже знаем, а вот метод **set\_select\_category()** пока совершенно непонятен. Нужно с ним разобраться.

# ВЫВОДИМ ИЗ БАЗЫ ДАННЫХ ТОВАРЫ ПЕРВОЙ НОМЕНКЛАТУРНОЙ ГРУППЫ

По заголовку понятно, что метод **set\_select\_category()** каким-то образом связан с базой данных.

Код этого метода мы добавим в модуль **markup.py**.

**Листинг 4. step\_4/markup/markup.py**

**def** set\_select\_category(self, category):  
 *"""  
 Создает разметку инлайн-кнопок в выбранной  
 категории товара и возвращает разметку  
 """* self.markup = InlineKeyboardMarkup(row\_width=1)  
 *# загружаем в названия инлайн-кнопок данные  
 # из БД в соответствие с категорией товара* **for** itm **in** self.BD.select\_all\_products\_category(category):  
 self.markup.add(self.set\_inline\_btn(itm))  
  
 **return** self.markup

В двух словах об этом коде:

**for** itm **in** self.BD.select\_all\_products\_category(category):  
 self.markup.add(self.set\_inline\_btn(itm))

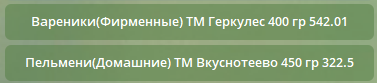
Для каждого товара из текущей номенклатурной группы (Категории) мы создаем кнопку.

self.markup = InlineKeyboardMarkup(row\_width=1)

И добавляем ее в разметку:

self.markup.add(self.set\_inline\_btn(itm))

Поэтому получаем:



Теперь самое сложное. Магический метод **select\_all\_products\_category()**:

self.BD.select\_all\_products\_category(category)

Объект менеджера базы данных мы уже создали ранее также в модуле **markup.py:**

self.BD = DBManager()

А сам класс **DBManager** был пока заглушкой. Этот класс мы описали в модуле **db\_alchemy.py** директории **data\_base**.

Теперь этот класс мы дополним кодом и получится:

**Листинг 4. step\_4/data\_base/dbalchemy.py**

**from** os **import** path  
  
**from** sqlalchemy **import** create\_engine  
**from** sqlalchemy.orm **import** sessionmaker  
**from** data\_base.dbcore **import** Base  
  
**from** settings **import** config  
**from** models.product **import** Products  
  
  
**class** Singleton(type):  
 *"""  
 Патерн Singleton предоставляет механизм создания одного  
 и только одного объекта класса,  
 и предоставление к нему глобальной точки доступа.  
 """* **def** \_\_init\_\_(cls, name, bases, attrs, \*\*kwargs):  
 super().\_\_init\_\_(name, bases, attrs)  
 cls.\_\_instance = **None  
  
 def** \_\_call\_\_(cls, \*args, \*\*kwargs):  
 **if** cls.\_\_instance **is None**:  
 cls.\_\_instance = super().\_\_call\_\_(\*args, \*\*kwargs)  
 **return** cls.\_\_instance  
  
  
**class** DBManager(metaclass=Singleton):  
 *"""   
 Класс-менеджер для работы с БД  
 """* **def** \_\_init\_\_(self):  
 *"""  
 Инициализация сесии и подключения к БД  
 """* self.engine = create\_engine(config.DATABASE)  
 session = sessionmaker(bind=self.engine)  
 self.\_session = session()  
 **if not** path.isfile(config.DATABASE):  
 Base.metadata.create\_all(self.engine)  
  
 **def** select\_all\_products\_category(self, category):  
 *"""  
 Возвращает все все товары категории  
 """* result = self.\_session.query(Products).filter\_by(  
 category\_id=category).all()  
  
 self.close()  
 **return** result  
  
 **def** close(self):  
 *""" Закрывает сессию """* self.\_session.close()

Отдельно по импортам:

**from** os **import** path  
  
**from** sqlalchemy **import** create\_engine  
**from** sqlalchemy.orm **import** sessionmaker  
**from** sqlalchemy.ext.declarative **import** declarative\_base  
  
**from** settings **import** config  
**from** models.product **import** Products

Встроенный подмодуль **path** нам пригодится для формирования пути к файлу базы данных.

Метод **create\_engine()** библиотеки SQLAlchemy необходим для создания подключения к базе данных.

Через метод **sessionmaker()** мы выполняем создание сессии. Что это такое? Через сессии ORM-библиотека SQLAlchemy взаимодействует с базой данных. Следующие два импорта понятные, мы их уже знаем.

Последний импорт очень важен. В нем мы импортируем наш класс-модель **Products**. Поскольку именно через этот класс мы будем взаимодействовать с базой данных. Вы же помните, что мы решили отказаться от использования SQL и работать по более продвинутому подходу – через ORM.

В методе **\_\_init\_\_()** происходит инициализация сессии.

self.engine = create\_engine(config.DATABASE)  
 session = sessionmaker(bind=self.engine)  
 self.\_session = session()  
 **if not** path.isfile(config.DATABASE):  
 Base.metadata.create\_all(self.engine)

Мы подключаемся к базе данных. Ее имя хранится в переменной-константе DATABASE. Далее создаем класс сессии, ее объект (self.session = session()). И почти все готово для работы с нашей базой данных.

Далее мы проверяем, если файл базы данных отсутствует, то по декларативному подходу работы в SQLAlchemy вызываем создание базы данных в ее исходном состоянии:

Base.metadata.create\_all(self.engine)

Дальше переходим к тому самому методу **select\_all\_products\_category()**, который непосредственно делает обращение к базе данных.

**def** select\_all\_products\_category(self, category):  
 *"""  
 Возвращает все товары категории  
 """* result = self.\_session.query(Products).filter\_by(  
 category\_id=category).all()  
 self.close()  
 **return** result

По сути, это альтернатива SQL-запросу. Через объект сессии (\_session) мы делаем запрос (query()) через класс-модель Products к соответствующей таблице базы данных.

Фильтруем полученный массив по указанной категории и получаем товары только для этой категории (category). Возвращаем именно все записи (all()).

Если вы хотя бы немного знакомы с ORM Django, суть этого запроса будет понятна. И важно, что это запрос! Но написанный не средствами SQL, а на привычном Python-синтаксисе в привязке к ORM-библиотеке SQLAlchemy.

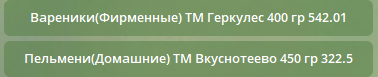
Вот такая удобная библиотека. Работаем с базой данных с помощью привычного Python-синтаксиса.

В конце важно вызвать закрытие сессии. А вот сама функция закрытия:

**def** close(self):  
 *""" Закрывает сесию """* self.\_session.close()

Выше, когда мы разбирали фрагмент:

**for** itm **in** self.BD.select\_all\_products\_category(category):  
 self.markup.add(self.set\_inline\_btn(itm))

То не коснулись функции self.set\_inline\_btn(itm). А эта функции добавляет вывод кнопки для товара текущей номенклатурной группы:

Код этой функции разместим в модуле **markup.py**.

**from** telebot.types **import** KeyboardButton, ReplyKeyboardMarkup, \  
 ReplyKeyboardRemove, InlineKeyboardMarkup, InlineKeyboardButton

**def** set\_inline\_btn(self, name):  
 *"""  
 Создает и возвращает инлайн кнопку по входным параметрам  
 """* **return** InlineKeyboardButton(str(name),  
 callback\_data=str(name.id))

Название товара будет выводиться на кнопке (str(name)). Кроме того, мы хотим перенаправить в другую функцию данные (id товара), полученные с текущей кнопки. Поэтому, мы прописываем callback\_data=str(name.id). Об этой другой функции мы поговорим позднее. Она будет обеспечивать добавление выбранного товара в заказ и нам нужен будет идентификатор этого товара. Т.е. мы будем нажимать на кнопку с товаром, получать его id, чтобы потом корректно сделать запрос к базе данных и добавить товар в заказ.

# ПОДКЛЮЧАЕМ БАЗУ ДАННЫХ

Итак, на этом уроке мы уже реализуем взаимодействие с базой данных. Но где сам файл базы данных?

Вот ссылка на него: **config.DATABASE**.

Если мы запустим проект с нуля, то у нас появится файл с пустой базой данных. А мы ведь уже обращаемся к базе данных, как будто в ней уже есть таблица продуктов с данными. Как быть?

Давайте пока вручную создадим файл базы данных, например, через браузер баз данных SQLiteStudio, вручную создадим всю структуру таблиц и вручную заполним таблицы данными.

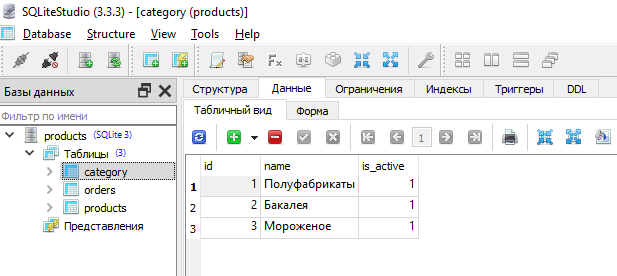
В модуле **config.py** у нас есть вот такая переменная-константа, которая ссылается на имя базы данных.

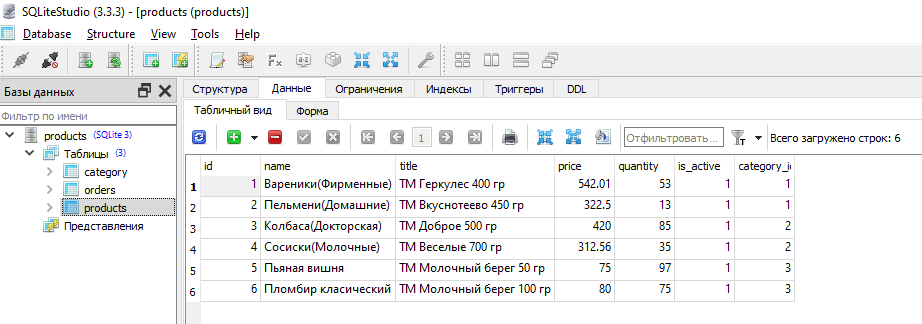
NAME\_DB = **'products.sqlite'**

Пока вы можете пока взять готовый файл **'products.sqlite'** из директории **settings** примера и добавить в свой проект. А потом уже потренироваться с SQLiteStudio. У этой десктопной программы интуитивно понятный интерфейс. Его освоение – дело нескольких минут.

Давайте откроем наш файл **'products.sqlite'** с помощью SQLiteStudio и посмотрим, что в нем.

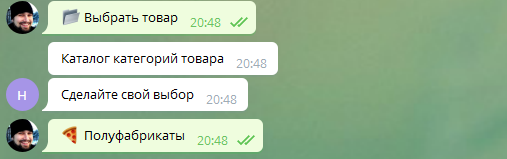
В базе данных на данный момент у нас три таблицы, из которых данные пока будут только в двух: «category» и «products». Эти данные мы заполним через графический интерфейс программы SQLiteStudio:



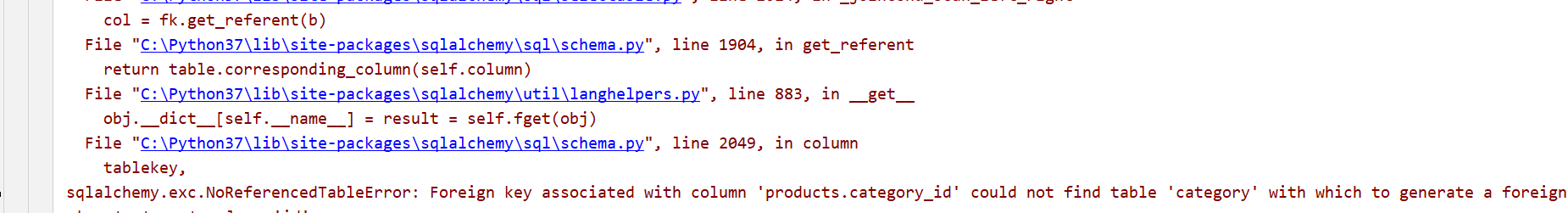


Пока наш бот «умеет» работать только с клиентом и реализован интерфейс только для клиента. Со временем в приложение можно будет добавить и другую «роль», например, Администратор и реализовать интерфейс для него. Админ сможет, используя свой интерфейс, заносить данные в базу.

Вроде мы все настроили и наш проект можно запускать.

Выбираем категорию товара и получаем:

И ничего не происходит. Товары выбранной номенклатурной группы не выводятся. А в PyCharm появляется вот такая ошибка:



Нам говорят, что как будто проблема со связями между таблицами категорий и продуктов. Хотя вроде связи эти в таблицах в SQLiteStudio видны.

В чем тогда дело? Значит мы где-то допустили ошибку?

Вспомним, что мы работаем с базой данных через ORM SQLAlchemy.

И вот такой момент. В каждом файле с моделями мы выполняем вот такую команду:

Base = declarative\_base()

От этого класса наследуются все наши классы-модели. Кроме того, в модуле **dbalchemy.py** тоже есть такая команда, а также команда:

Base.metadata.create\_all(self.engine)

По сути, мы выполняем генерацию структуры всех таблиц базы данных. Это похоже на выполнение миграций в Django ORM, когда структура моделей переносится на структуру базы данных.

В нашем случае важно, чтобы вся работа осуществлялась через один и тот же объект **Base**.

Т.е. нужно не в каждом файле отдельно вызывать

Base = declarative\_base()

А сделать это в одном общем файле, а потом выполнить импорт.

Создадим в директории **data\_base** модуль **dbcore.py**.

**Листинг 5. step\_4/data\_base/dbcore.py**

**from** sqlalchemy.ext.declarative **import** declarative\_base  
*# инициализация декларативного стиля для БД*Base = declarative\_base()

Теперь в модулях **category.py**, **order.py**, **product.py**, **dbalchemy.py** убираем команды:

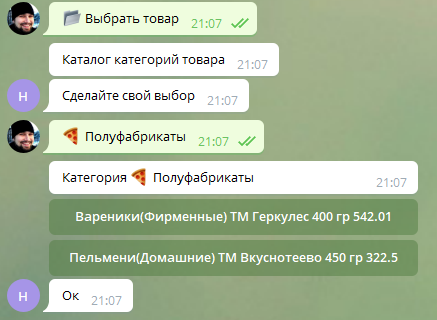
**from** sqlalchemy.ext.declarative **import** declarative\_base

Base = declarative\_base()

И заменяем их на:

**from** data\_base.dbcore **import** Base

Теперь все работает:



Если выполнить нажатие на кнопки «Бакалея» и «Мороженое», увидим список товаров для этих категорий. Все работает.

# Вид сверху на рукопожатие над деловыми документамиЗАКЛЮЧЕНИЕ

Теперь наш проект «умеет» работать с базой данных. Мы можем просматривать список товаров, входящих в каждую категорию. Используя возможности Telegram Bot API и SQLAlchemy, мы реализовали привычный вывод содержимого таблиц через графический интерфейс бота.

## КЛЮЧЕВЫЕ ВЫВОДЫ

* Подход, описанный в уроке, напоминает работу с PyQt, когда при выполнении пользователем действий в графическом интерфейсе происходит удаление одних и размещение других элементов управления (кнопок, полей ввода и т.д.)
* Благодаря SQLAlchemy мы выполняем запросы любой сложности, но избавлены от необходимости работы с языком запросов SQL. Но есть некоторые важные особенности работы с данной библиотекой, о которых мы узнали в конце раздела.
* Пока мы сделали ограничение, что база данных со всеми таблицами у нас уже есть, но если развивать проект, можно настроить деление на роли: покупатель, админ, менеджер. И настроить для админа графический интерфейс, через который он сможет заполнять базу данных. Пока же мы работаем с базой данных вручную, через приложение SQLiteStudio.

Текущая структура разрабатываемого Telegram-бота приведена на на рис. 1.

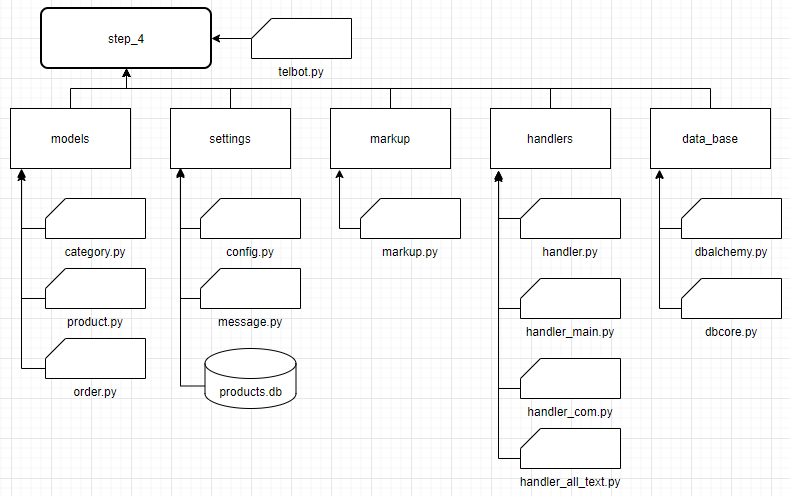


Рис. 1. Структура проекта