

Урок 1. Создаем модели-сущности предметной области

ОГЛАВЛЕНИЕ

[ДЛЯ ЧЕГО НУЖНЫ МОДЕЛИ 3](#_Toc79586699)

[СОЗДАЕМ ПЕРВЫЕ КЛАССЫ-МОДЕЛИ 4](#_Toc79586700)

[ЗАКЛЮЧЕНИЕ 8](#_Toc79586701)

[КЛЮЧЕВЫЕ ВЫВОДЫ 8](#_Toc79586702)



# ДЛЯ ЧЕГО НУЖНЫ МОДЕЛИ

Когда вы работали с известными фреймворками Django и Flask для разработки веб-приложений, то помните, что в процессе решения этой задачи мы создавали классы-модели и размещали их в отдельном модуле **models.py**. Каждый класс-модель соответствовал определенной сущности вашего проекта, например, категории, товару, пользователю. Каждая из таких моделей, по сути, описывала структуру соответствующей таблицы базы данных. Использование моделей является частью механизма ORM, когда вместо подготовки и запуска SQL-запросов, мы работаем с базой данных посредством привычного Python-синтаксиса. Но это было в веб-приложениях, а у нас задача – написание Telegram-бота.

Но ведь наш бот тоже посвящен некоторой предметной области и предполагает хранение информации. В нашем случае это информация о товарах в каталоге, их количестве, единицах измерения, информация о заказе и т.д. Объем хранимой в базе данных информации будет расширяться с ростом проекта. Конечно, у нас есть вариант использовать подход DB-API, но тогда придется засесть за изучение команд языка SQL. А значит нам тоже необходим ORM-подход и подготовка классов-моделей для сущностей нашей предметной области. В Django применяется встроенный ORM-механизм, но никто не мешает нам воспользоваться сторонней ORM-библиотекой, например, SQLAlchemy. Это свободно распространяемый инструмент для работы с базами данных в Python-приложениях. Полноценное освоение «Алхимии» требует времени, но сейчас нам нет необходимости бросать на это все силы. Достаточно освоить базовые возможности инструмента. Установку SQLAlchemy можно выполнить через PyCharm или командную строку:

pip install SQLAlchemy

# СОЗДАЕМ ПЕРВЫЕ КЛАССЫ-МОДЕЛИ

Так как мы храним информацию о каталоге товаров, нам необходимо для начала описать сущность – «Категория товара».

**Листинг 1. step\_1/models/category.py**

|  |
| --- |
| *# компоненты библиотеки для описания структуры таблицы* **from** sqlalchemy **import** Column, String, Integer, Boolean *# класс-конструктор для работы с декларативным стилем работы с SQLAlchemy* **from** sqlalchemy.ext.declarative **import** declarative\_base *# инициализация декларативного стиля* Base = declarative\_base()   **class** Category(Base):  *"""  Класс-модель для описания таблицы "Категория товара",  основан на декларативном стиле SQLAlchemy  """  # название таблицы* \_\_tablename\_\_ = **'category'** *# поля таблицы* id = Column(Integer, primary\_key=**True**)  name = Column(String, index=**True**)  is\_active = Column(Boolean)   **def** \_\_str\_\_(self):  *"""  Метод возвращает строковое представление объекта класса  """* **return** self.name |

Важно отметить, что в SQLAlchemy существует так называемый традиционный и декларативный стили описания сущностей. Декларативный стиль считается более лаконичным и предполагает создание специального класса с помощью конструктора **declarative\_base()**. Созданный класс будет родителем для всех классов-сущностей нашего проекта. В нашем классе-категории, как и в соответствующей таблице базы данных будет три поля, определение которых вам будет интуитивно понятно, если вы ранее работали с базами данных. Перегрузка метода **\_\_str\_\_()** здесь используется, чтобы определить строковое представление объекта класса-категории, т.е. вместо указания на область памяти, где хранится объект, мы получим вывод строки-названия категории.

**Листинг 2. step\_1/models/product.py**

|  |
| --- |
| *# компоненты библиотеки для описания структуры таблицы* **from** sqlalchemy **import** Column, String, Integer, Float, Boolean, ForeignKey *# импортируем модуль для связки таблиц* **from** sqlalchemy.orm **import** relationship, backref *# класс-конструктор для работы с декларативным стилем работы с SQLAlchemy* **from** sqlalchemy.ext.declarative **import** declarative\_base *# импортируем модель Категория для связки моделей* **from** models.category **import** Category   *# инициализация декларативного стиля* Base = declarative\_base()   **class** Products(Base):  *"""  Класс для создания таблицы "Товар",  основан на декларативном стиле SQLAlchemy  """  # название таблицы* \_\_tablename\_\_ = **'products'** *# поля таблицы* id = Column(Integer, primary\_key=**True**)  name = Column(String, index=**True**)  title = Column(String)  price = Column(Float)  quantity = Column(Integer)  is\_active = Column(Boolean)  category\_id = Column(Integer, ForeignKey(**'category.id'**))  *# для каскадного удаления данных из таблицы* category = relationship(  Category,  backref=backref(**'products'**,  uselist=**True**,  cascade=**'delete, all'**))   **def** \_\_str\_\_(self):  **return f"{**self.name**} {**self.title**} {**self.price**}"** |

В приведенном листинге класс-модель описывает сущность «Товар». Структура класса во многом идентична предыдущему листингу. В классе есть указание на привязку к внешней таблице, где хранятся категории товаров. Стоит обратить внимание, что поле **name** применяется для указания названия товара, а поле **title** – для указания названия компании-производителя. Важно также отметить функцию **relationship()**, которая определяет действия, выполняемые при удалении категории. В данном случае товары, связанные с указанной категорией также будут удалены. Значение **uselist=True** означает, что указанный механизм будет применен ко всем найденным отношениям.

**Листинг 3. step\_1/models/order.py**

|  |
| --- |
| *# компоненты библиотеки для описания структуры таблицы* **from** sqlalchemy **import** Column, DateTime, Integer, ForeignKey *# импортируем модуль для связки таблиц* **from** sqlalchemy.orm **import** relationship, backref *# класс-конструктор для работы с декларативным стилем работы с SQLAlchemy* **from** sqlalchemy.ext.declarative **import** declarative\_base *# импортируем модель продуктов для связки моделей* **from** models.product **import** Products   *# инициализация декларативного стиля* Base = declarative\_base()   **class** Order(Base):  *"""  Класс для создания таблицы "Заказ",  основан на декларативном стиле SQLAlchemy  """  # название таблицы* \_\_tablename\_\_ = **'orders'** *# поля таблицы* id = Column(Integer, primary\_key=**True**)  quantity = Column(Integer)  data = Column(DateTime)  product\_id = Column(Integer, ForeignKey(**'products.id'**))  user\_id = Column(Integer)   *# для каскадного удаления данных из таблицы* products = relationship(  Products,  backref=backref(**'orders'**,  uselist=**True**,  cascade=**'delete,all'**))   **def** \_\_str\_\_(self):  **return f"{**self.quantity**} {**self.data**}"** |

В приведенном классе также есть указание на внешнюю таблицу, реализовано каскадное удаление. В нашем курсе реализуется MVP-версия, поэтому список моделей пока будет ограничен представленными классами, но постепенно его можно будет дополнить другими сущностями.

# Вид сверху на рукопожатие над деловыми документамиЗАКЛЮЧЕНИЕ

Друзья, наш первый урок подошел к концу. Мы решили начать с системы хранения данных и реализовали классы, описывающие сущности предметной области, в которой будет работать наш Telegram-бот. Со временем список сущностей может быть расширен.

## КЛЮЧЕВЫЕ ВЫВОДЫ

* Классы-модели – важная составляющая любого проекта на Python, не только веб-приложения. Благодаря таким классам и применению ORM-библиотек работа с базой данных ведется без необходимости написания SQL-кода.
* Сторонние ORM-библиотеки, например, SQLAlchemy требуют тщательного изучения, но дают разработчику мощный инструментарий для работы с базами данных из Python-приложения.
* Декларативный подход описания моделей-сущностей отличается понятностью и удобством применения.

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

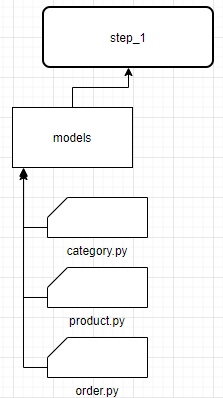


Рис. 1. Структура проекта