**Язык *SQL***

Этот язык называется SQL — **Structured Query Language** — язык структурированных запросов.

од, написанный на *SQL*, как правило, сохраняется в файлах вида *database.sql*. Сейчас мы посмотрим, как с помощью этого языка можно описать создание таблиц в базе данных.

Основная команда, которая будет нами использоваться — это *CREATE TABLE*.

Например, заголовок создания таблицы *ORDERS*, хранящей заказы, будет выглядеть так:

**CREATE** **TABLE** ORDERS ();

Далее наша задача состоит в том, чтобы описать атрибуты сущности. Как мы уже говорили, каждый атрибут имеет определённый тип.

Базовые типы схожи с аналогичными в *Python*:

* + INT — целочисленный тип;
  + FLOAT — числа с плавающей точкой;
  + CHAR(size) — хранение строк до 255 символов (длина указывается в *size*);
  + TEXT — хранение больших текстовых данных;
  + DATETIME — атрибуты позволяют хранить дату и время.

Этих типов данных, используемых в *SQL*, нам сейчас будет достаточно. Более того, они являются универсальными.

Чтобы описать атрибут сущности, необходимо пользоваться следующей конструкцией языка внутри конструкции создания таблицы:

**CREATE** **TABLE** **table\_name** (field\_name **TYPE**);

Опишем некоторые (пока что не все) атрибуты сущностей базы данных нашего интернет-магазина*:*

1. **Сущность «Заказ»**

**CREATE** **TABLE** ORDERS (

order\_id INT AUTO\_INCREMENT **NOT** **NULL**,

time\_in DATETIME **NOT** **NULL**,

time\_out DATETIME **NOT** **NULL**,

cost FLOAT **NOT** **NULL**,

pickup INT **NOT** **NULL**

);

Здесь мы дополнительно использовали несколько свойств атрибутов.

AUTO\_INCREMENT указывает базе данных, что она должна сама озаботиться тем, чтобы записать сюда значение. Помните, как *Python* автоматически назначал каждому объекту идентификатор. Именно это же мы указываем базе данных делать с данным атрибутом, он служит уникальным идентификатором, и мы требуем, чтобы БД сама назначала порядковый номер каждой строке.

Также в каждом атрибуте появилось таинственное NOT NULL. И не зря. Когда мы создавали первую большую таблицу, обнаружили в ней пропуски — отсутствие значений. Мы сошлись на том, что так делать не очень хорошо. И чтобы вдруг случайно такой ситуации не случилось в базе данных, мы прямо указываем, что каждое поле должно иметь значение. Иными словами быть ненулевым (*not* *null*).

В этой сущности мы создали атрибуты:

* + order\_id (целочисленный атрибут с автоинкрементом) — уникальный идентификатор заказа, который база данных будет автоматически назначать каждой строке;
  + time\_in (тип даты/времени) — хранит время оформления заказа;
  + time\_out (тип даты/времени) — хранит время выдачи заказа;
  + cost (число с плавающей точкой) — стоимость заказа в рублях;
  + pickup (целочисленный атрибут) — доставка (0) или самовывоз (1). Мы использовали целочисленный тип (который, как вы помните, хорошо приводится к логическому типу) только с двумя значениями.

В данной таблице мы на текущий момент не добавили только сотрудника. Мы обязательно вернёмся к нему в следующем юните, а сейчас продолжим описание других таблиц на языке *SQL*.

## Сущность «Продукт»

Опишем таблицу *PRODUCTS*, соответствующую сущности «Продукт».

**CREATE** **TABLE** PRODUCTS (

product\_id INT AUTO\_INCREMENT **NOT** **NULL**,

name CHAR(**255**) **NOT** **NULL**,

price FLOAT **NOT** **NULL**

);

Здесь мы имеем:

* + product\_id (целочисленный атрибут с автоинкрементом) — уникальный идентификатор товара в каталоге;
  + name (строка до 255 символов) — название товара;
  + price (число с плавающей точкой) — цена товара.

### Задание 2.2.4

Задание на самопроверку.

С помощью языка SQL и написанных ранее примеров, создайте таблицу STAFF, которая имеет следующие атрибуты:

* + staff\_id (целочисленный атрибут с автоинкрементом) — уникальный идентификатор каждого сотрудника;
  + full\_name (строка до 255 символов) — ФИО сотрудника;
  + position (строка до 255 символов) — должность сотрудника;
  + labor\_contract (целочисленный атрибут) — номер трудового договора.

Ответ

**CREATE** **TABLE** STAFF (

staff\_id INT AUTO\_INCREMENT **NOT** **NULL**,

full\_name CHAR(**255**) **NOT** **NULL**,

position CHAR(**255**) **NOT** **NULL**,

labor\_contract INT **NOT** **NULL**

);

Как вы могли заметить, здесь мы описали не все атрибуты, а опустили те из них, которые являются связующими между таблицами. Это связано с тем, что прежде чем создавать связи, нужно понять, какие они бывают и зачем они вообще нужны, этим мы займёмся в следующем юните.

мы имеем **связи между сущностями.**

Базы данных, в которых на этапе проектирования заложены предопределённые связи между сущностями, называют **реляционными.**

Именно такие базы данных мы и будем обсуждать в нашем курсе. Однако стоит понимать, что есть и другие способы хранения данных, и о некоторых из них можно прочитать в [этой статье](https://habr.com/ru/post/103021/). Если вы не имели опыта работы с базами данных, рекомендуется ознакомиться со статьёй после изучения этого юнита.

Существует три основных типа связей (отношений) между сущностями:

* 1. Один к одному.
  2. Один ко многим.
  3. Многие ко многим.

**Один к одному**

Этот тип связи между сущностями присутствует, когда один из объектов одной сущности связан **только с одним** объектом другой сущности.

## ****Создание ключей с помощью****SQL

Сначала посмотрим, как мы должны модицифировать созданные нами в прошлом юните таблицы Orders, Products, Staff. Начнём с таблицы Orders.

Здесь в качестве первичного ключа выступает атрибут order\_id:

**CREATE** **TABLE** ORDERS (

order\_id INT AUTO\_INCREMENT **NOT** **NULL**,

time\_in DATETIME **NOT** **NULL**,

time\_out DATETIME,

cost FLOAT **NOT** **NULL**,

pickup INT **NOT** **NULL**,

**PRIMARY** **KEY** (order\_id)

);

Мы добавили строку PRIMARY KEY (order\_id), которая и выполняет наше желание.

## ****Создание ключей с помощью****SQL

Сначала посмотрим, как мы должны модицифировать созданные нами в прошлом юните таблицы Orders, Products, Staff. Начнём с таблицы Orders.

Здесь в качестве первичного ключа выступает атрибут order\_id:

**CREATE** **TABLE** ORDERS (

order\_id INT AUTO\_INCREMENT **NOT** **NULL**,

time\_in DATETIME **NOT** **NULL**,

time\_out DATETIME,

cost FLOAT **NOT** **NULL**,

pickup INT **NOT** **NULL**,

**PRIMARY** **KEY** (order\_id)

);

Мы добавили строку PRIMARY KEY (order\_id), которая и выполняет наше желание.

### Задание 2.3.3

Возьмите таблицы Products и Staff, созданные в конце прошлого юнита и модифицируйте их, добавив определение первичного ключа.

Ответ

**CREATE** **TABLE** PRODUCTS (

product\_id INT AUTO\_INCREMENT **NOT** **NULL**,

name CHAR(**255**) **NOT** **NULL**,

price FLOAT **NOT** **NULL**,

**PRIMARY** **KEY** (product\_id)

);

**CREATE** **TABLE** STAFF (

staff\_id INT AUTO\_INCREMENT **NOT** **NULL**,

full\_name CHAR(**255**) **NOT** **NULL**,

position CHAR(**255**) **NOT** **NULL**,

labor\_contract INT **NOT** **NULL**,

**PRIMARY** **KEY** (staff\_id)

);

Внешний ключ определяется аналогичным способом. Рассмотрим на примере построения связи «один ко многим». В зависимой таблице (в которой «много» объектов) необходимо добавить строку, похожую на ту, что мы использовали для создания первичного ключа:

**FOREIGN** **KEY** (имя\_атрибута) **REFERENCES** Основная\_Таблица (первичный\_ключ)

Давайте сразу применим это к нашим таблицам. В таблице заказов Orders в нашей изначальной задумке мы имели атрибут staff, который должен указывать на сотрудника, выполнившего заказ. Давайте вновь модифицируем эту таблицу, создав внешний ключ к таблице Staff.

**CREATE** **TABLE** ORDERS (

order\_id INT AUTO\_INCREMENT **NOT** **NULL**,

time\_in DATETIME **NOT** **NULL**,

time\_out DATETIME,

cost FLOAT **NOT** **NULL**,

pickup INT **NOT** **NULL**,  
 staff INT **NOT** **NULL**,

**PRIMARY** **KEY** (order\_id),

FOREIGN **KEY** (staff) **REFERENCES** STAFF (staff\_id)

);

### Задание 2.3.4

При помощи SQL создайте таблицу PRODUCTS\_ORDERS, которая должна:

* 1. Содержать атрибут product\_order\_id, который предполагается целочисленным, автоматически увеличивающимся на 1 и тем самым должен стать первичным ключом этой таблицы.
  2. Содержать атрибут product, который ссылается на первичный ключ таблицы Products.
  3. Содержать атрибут in\_order, который ссылается на первичный ключ таблицы Orders.
  4. Содержать атрибут amount, который определяет количество конкретного продукта в заказе. Мы предполагаем, что это целое число.

Ответ

**CREATE** **TABLE** PRODUCTS\_ORDERS (

product\_order\_id INT AUTO\_INCREMENT **NOT** **NULL**,

product INT **NOT** **NULL**,

in\_order INT **NOT** **NULL**,

amount INT **NOT** **NULL**,

**PRIMARY** **KEY** (product\_order\_id),

FOREIGN **KEY** (product) **REFERENCES** PRODUCTS (product\_id),

FOREIGN **KEY** (in\_order) **REFERENCES** ORDERS (order\_id)

);

И это большой шаг в построении баз данных, потому что только что была реализована не просто таблица с двумя внешними ключами, а самая настоящая промежуточная таблица для связи «многие ко многим». Ведь именно таким образом она и создаётся, если вспомнить определение выше — каждая строка таблицы должна ссылаться на какой-то объект из первой исходной таблицы (в нашем случае — PRODUCTS) и в то же самое время на объект из второй исходной таблицы — ORDERS из примера.

Отлично! Первая связь построена! Сделаем последний рывок, полностью создав таблицу «Products\_Orders» с нуля.

Давайте посмотрим, какие приложения были созданы:

* + - * *accounts* — профили пользователей и всё, что с ними связано;
      * *courses* — о курсах, темах, а также функционал банка заданий;
      * *events* — система создания тестов, а также управления видеоуроками;
      * *study* — приложение, которое управляет образовательным процессом со стороны ученика;
      * *students* — приложение, с помощью которого методисты и преподаватели могут отслеживать прогресс студентов.

Как видите, все они сильно взаимосвязаны, но каждый по отдельности выполняет только свою часть «работы».

## ****Приложения****

Разбавим теоретическую часть небольшой практикой. В одном из скринкастов предыдущего модуля вы создавали пустой проект. Давайте проделаем это снова, чтобы по ходу текущего модуля частично создать проект новостного портала.

Создаём виртуальное окружение:

**$** python3 -m venv venv

Для *Windows*:

python -m venv venv

Заходим в него:

~/django-projects $ source venv/bin/activate

Для *Windows*:

venv\scripts\activate

Устанавливаем Django в свежее виртуальное окружение:

(venv) ~/django-projects $ pip3 install django

Для *Windows*:

pip install django

И запускаем команду создания проекта:

(venv) ~/django-projects $ django-admin startproject NewsPaper

Переходим в директорию проекта:

(venv) ~/django-projects $ cd NewsPaper

Здесь мы видим замечательный файл manage.py, который является точкой входа для управления проектом. Также через консоль запустим следующую команду, которая создаст новое приложение news.

(venv) ~/django-projects/NewsPaper $ python manage.py startapp news

Здесь мы использовали команду startapp из скрипта manage.py. В качестве параметра этой команды мы должны указать название нового приложения — news. Мы можем увидеть новую директорию, в которой есть большое количество файлов.

Django автоматически создал основные необходимые файлы для нового приложения. Чтобы это приложение стало частью этого проекта, мы должны его добавить в установленные приложения.

Перейдем в файл NewsPaper/settings.py и найдём там список INSTALLED\_APPS:

Здесь мы должны добавить новый элемент в этот список — строку с названием приложения, которое совпадает с названием директории(news). Это позволит Django обнаружить созданное нами приложение.

## MVC****или****MTV****— вот в чём вопрос****

Теперь снова немного к теории, чтобы понять, как устроен Django. Проекты в Django, как мы уже говорили, модульные. Однако внутреннее устройство каждого приложения не является произвольным — оно соответствует определённым шаблонам проектирования.

При проектировании приличной части веб-приложений применяется шаблон **M**odel-**V**iew-**C**ontroller — MVC.

Шаблон состоит из трёх основных частей:

* 1. ***Controller*** (контроллер).  
     Эта часть приложения отвечает за **бизнес-логику**. Оно определяет, что можно делать в том или ином веб-приложении: какие действия вообще могут быть, кто их может выполнять (доступ) и т. д.
  2. ***Model*** (модель).  
     Здесь обеспечивается **работа с данными**:взаимодействие с базой данных, выгрузка данных оттуда, подготовка для последующей обработки в контроллере и всё-всё, что может быть связано напрямую с данными
  3. ***View*** (представление).  
     Это визуал веб-приложения — конкретное отображение данных.

С помощью этой структуры можно понять, как происходит взаимодействие с пользователем:

* 1. Пользователь посылает запрос на сервер. Например, «хочу получить последние 10 новостей».
  2. Этот запрос принимает контроллер. Он понимает, что от него просят новости и должен их где-то получить.
  3. Для этого он бежит к модели и сообщает ей: «Меня тут выгрузить новости попросили, но только последние 10, скинь, плиз». Модель обращается к базе данных и выгружает объекты сущности *Articles* (статьи). Естественно, что контроллеру не нужно всё, что хранится в каждой из 10 строк, но ему нужен, наверняка заголовок, текст и дата. Он берёт именно их и отдаёт контроллеру.
  4. Счастливый контроллер идет к представлению, передаёт ему то, что забрал у модели и говорит: «Сделай так, чтобы красиво и понятно».
  5. Представление покрутило  строчками кода и отправило красиво оформленную страничку пользователю.

Однако в *Django* используется не *MVC*, а *MTV*. Тогда зачем мы это разбирали сейчас, спросите вы? Во-первых, потому что *MVC* универсальная архитектура — её используют фреймворки для *PHP*, *Java*, *Ruby* *on* *Rails* и другие веб-фреймворки на *Python* могут быть адаптированы для неё. Однако *MTV* специфична именно для *Django*, хоть и практически ничем не отличается от *MVC*.

Архитектура *MTV* — это ***M****odel*-***T****emplate*-***V****iew*.

Отличия несущественные:

* 1. Модель, как была моделью, так ей и остаётся.
  2. *Template* — шаблон — это то, что называлось представлением в архитектуре *MVC*, отвечает за отображение данных.
  3. *View* — представление — это то, что было контроллером в *MVC,*управляет бизнес-логикой.

Если кратко, в *Django* контроллеры называют представлениями, а представления — шаблонами.