

Проект WebServer. Введение в БД

План урока

- 1 Введение в базы данных
- 2 Создание первого веб-приложения, работающего с базой данных
- 3 Создание таблиц в базе данных
- 4 Работаем с базой данных из Python
- 5 Создаём формы для веб-приложения
- 6 Авторизация пользователя в веб-приложении
- 7 Идеи для улучшения веб-приложения
- 8 Первый взгляд на sqlalchemy

Аннотация

На этом занятии мы создадим наше первое веб-приложение, работающее с базой данных, а также рассмотрим несколько вариантов работы с БД: напрямую через SQL и через ORM.

1. Введение в базы данных

Хранить данные в приложении можно несколькими способами. Можно использовать даже обычные текстовые файлы (если приложение однопользовательское или пользователей мало и они не конкурируют за доступ), но в большинстве случаев это — не самая лучшая идея, потому что при одновременном редактировании пользователями файла будут происходить коллизии.

Для хранения данных веб-приложений принято использовать базы данных. Видов баз данных достаточно много, но условно их можно поделить на две большие группы:

1. Реляционные.
2. Нереляционные.

Из нереляционных баз данных последнее время большую популярность приобрела **mongoDB**, которая позволяет хранить **коллекции** информации в хорошо известном нам формате json. MongoDB идеально подходит для случаев, когда мы точно не знаем, какие данные нам надо обрабатывать, они неполные или, наоборот, избыточные. Она идеально подходит для областей, связанных с анализом данных, и приложений с разнородными данными. Кроме того, ей достаточно удобно пользоваться из Python, и мы рекомендуем вам познакомиться с ней самостоятельно. Подробности можно посмотреть на [официальном сайте](#).

Реляционные базы данных представляют собой таблицы (их можно представить себе как набор листов в Excel) и связи между этими таблицами. Их удобнее использовать в тех областях, где структура данных (набор информации, которую мы собираемся хранить и обрабатывать) определена достаточно жёстко.

Для обеспечения работы базы данных существует специальное программное обеспечение, называемое **СУБД** (Система управления базами данных). Практически всегда для этого необходимо устанавливать дополнительное ПО на локальный компьютер (или присоединяться к удалённому, уже настроенному), но существуют и встраиваемые базы данных, для которых такой необходимости нет.

Одной из таких баз данных является **SQLite**. К её преимуществам можно отнести ещё и то, что Python может работать с ней «из коробки», и нет необходимости устанавливать дополнительные модули. SQLite содержит некоторые ограничения по сравнению со «взрослыми» базами данных, но всё, о чем мы поговорим сегодня, будет справедливо и для MySQL, и для PostgreSQL, и для других реляционных баз данных.

Сам Flask не имеет собственных инструментов для работы с базами данных (в отличие, например, от Django) и предоставляет нам полную свободу выбора инструментов для работы.

2. Создание первого веб-приложения, работающего с базой данных

Давайте создадим наше первое веб-приложение, которое будет использовать базу данных. Для начала давайте определимся с функциональностью. У нас будет простое веб-приложение, в котором пользователи могут авторизоваться, просматривать свои новости, добавлять и удалять их — такой аналог личного дневника.

Исходя из постановки задачи, нам надо хранить информацию о двух сущностях:

1. Пользователи.
2. Новости.

3. Создание таблиц в базе данных

За каждую сущность в нашей базе данных будет отвечать одна таблица, поэтому нам надо будет создать всего две таблицы: **users** и **news** соответственно. Как и при проектировании классов и объектов, нам важны не все данные о наших сущностях, а только некоторые. Для пользователей это будут: уникальный идентификатор пользователя, его логин и пароль (хранить пароль в открытом виде в базе данных очень-очень плохая идея, но пока оставим так, а чуть позже вместо паролей будем хранить только **хэши** паролей). А для новости — уникальный идентификатор новости, заголовок, текст новости и её автор.

На языке **SQL** (язык запросов для реляционных баз данных) запрос создания таблицы с новостями будет выглядеть вот так:

```
CREATE TABLE IF NOT EXISTS news
(
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title VARCHAR(100),
  content VARCHAR(1000),
  user_id INTEGER
)
```

Вы обязательно обратите внимание, что несложные запросы на SQL очень похожи на обычную английскую речь. SQL — **декларативный** язык, то есть на нём мы пишем не «как надо сделать», а «что надо сделать». Наш запрос очень легко прочитать: мы просим SQLite создать таблицу, если она не существует, с именем news, которая состоит из 4-х полей:

1. id — уникальный идентификатор новости, который будет целым числом, первичным ключом (с уникальным значением), а также автоинкрементным, то есть будет увеличиваться на единицу каждый раз при добавлении новой новости. За увеличением id будет следить сама база данных, и нам про это думать не обязательно.

2. `title` — заголовок новости, `VARCHAR` — аналог строки в Python. В скобках мы указываем максимальную длину поля, чтобы в базе данных не резервировалось много места.
3. `content` — содержание нашей новости.
4. `user_id` — автор новости. Для удобства мы будем хранить в новости не имя пользователя, который написал её, а его уникальный номер.

Аналогично для таблицы пользователей:

```
CREATE TABLE IF NOT EXISTS users
(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_name VARCHAR(50),
    password_hash VARCHAR(128)
)
```

Обратите внимание: хотя мы пока будем хранить пароли в открытом виде, поле в БД мы назвали «на вырост».

4. Работаем с базой данных из Python

Перейдём от SQL к Python. Давайте создадим класс, который будет управлять подключением к базе данных. Назовем его `db.py`:

```
import sqlite3

class DB:
    def __init__(self):
        conn = sqlite3.connect('news.db', check_same_thread=False)
        self.conn = conn

    def get_connection(self):
        return self.conn

    def __del__(self):
        self.conn.close()
```

В конструкторе класса мы присоединяемся к базе данных с именем **news.db** (хорошая идея — вынести это значение в настройки приложения, а не зашивать его в код). Если база данных не существует, она будет создана автоматически. Запоминаем в классе подключение к этой

базе данных. По запросу мы будем отдавать ссылку на подключение всем, кто к нам обращается, а при уничтожении объекта — закрывать соединение с базой данных.

Для управления данными в таблицах давайте создадим ещё два класса: **NewsModel** и **UsersModel**. Для **UsersModel** нам надо будет создать несколько методов.

Конструктор, который будет принимать параметром соединение к базе данных:

```
def __init__(self, connection):
    self.connection = connection
```

В этот же класс можно добавить метод первоначального создания таблицы:

```
def init_table(self):
    cursor = self.connection.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users
                      (id INTEGER PRIMARY KEY AUTOINCREMENT,
                       user_name VARCHAR(50),
                       password_hash VARCHAR(128)
                      )''')

    cursor.close()
    self.connection.commit()
```

По этому методу можно проследить общий принцип работы с базой данных. Мы создаём **курсор** (специальный объект для доступа к базе данных), затем исполняем какой-либо SQL-скрипт, закрываем курсор, и, наконец, вызываем метод **commit** у соединения. Это делается для того, чтобы все изменения были записаны в базу данных.

Сюда же давайте добавим метод добавления нового пользователя, который принимает на вход имя пользователя и хэш пароля:

```
def insert(self, user_name, password_hash):
    cursor = self.connection.cursor()
    cursor.execute('''INSERT INTO users
                      (user_name, password_hash)
                      VALUES (?,?)''', (user_name, password_hash))

    cursor.close()
    self.connection.commit()
```

Для добавления записей в базу данных используется команда **INSERT**. Наш скрипт можно прочитать следующим образом: вставим в таблицу `users` в поля `user_name` и `password_hash` значения, которые принимают переменные `user` и `password_hash`.

Допишем методы для получения одного пользователя и нескольких пользователей:

```

def get(self, user_id):
    cursor = self.connection.cursor()
    cursor.execute("SELECT * FROM users WHERE id = ?", (str(user_id),))
    row = cursor.fetchone()
    return row

def get_all(self):
    cursor = self.connection.cursor()
    cursor.execute("SELECT * FROM users")
    rows = cursor.fetchall()
    return rows

```

Для извлечения данных используется команда **SELECT**. Наша команда для выборки одного пользователя расшифровывается так: выбрать все поля (вместо «*» можно просто перечислить поля через запятую) из таблицы users, где поле id принимает значение user_id. Часть WHERE — это часть условий, которая поддерживает логические команды AND, NOT и OR в любой комбинации.

cursor.fetchone() — пытается достать одну запись из возвращаемого значения и преобразует её в кортеж значений полей. Порядок полей в кортеже совпадает с порядком полей в SELECT или совпадает с порядком полей при создании таблицы (если была указана «*»). Если наш запрос не вернул ни одного поля, то вернётся None.

cursor.fetchall() — делает тоже самое, но только для нескольких записей, то есть возвращается список кортежей значений полей.

Давайте добавим в модель для пользователей ещё метод проверки существования пользователя с предоставленным логином и паролем:

```

def exists(self, user_name, password_hash):
    cursor = self.connection.cursor()
    cursor.execute("SELECT * FROM users WHERE user_name = ? AND password_hash = ?"
                  (user_name, password_hash))
    row = cursor.fetchone()
    return (True, row[0]) if row else (False,)

```

В этом методе мы будем возвращать True и уникальный идентификатор пользователя, если он найден, и False в противном случае.

Модель для новостей будет выглядеть практически аналогичным образом.

Конструктор:

```

def __init__(self, connection):
    self.connection = connection

```

Первоначальное создание таблицы:

```
def init_table(self):
    cursor = self.connection.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS news
                      (id INTEGER PRIMARY KEY AUTOINCREMENT,
                       title VARCHAR(100),
                       content VARCHAR(1000),
                       user_id INTEGER
                      )''')

    cursor.close()
    self.connection.commit()
```

Добавление новости:

```
def insert(self, title, content, user_id):
    cursor = self.connection.cursor()
    cursor.execute('''INSERT INTO news
                      (title, content, user_id)
                      VALUES (?, ?, ?)''', (title, content, str(user_id)))

    cursor.close()
    self.connection.commit()
```

Получение одной и нескольких новостей соответственно:

```
def get(self, news_id):
    cursor = self.connection.cursor()
    cursor.execute("SELECT * FROM news WHERE id = ?", (str(news_id),))
    row = cursor.fetchone()
    return row

def get_all(self, user_id = None):
    cursor = self.connection.cursor()
    if user_id:
        cursor.execute("SELECT * FROM news WHERE user_id = ?",
                       (str(user_id),))
    else:
        cursor.execute("SELECT * FROM news")
    rows = cursor.fetchall()
    return rows
```

И давайте добавим ещё удаление новости:

```
def delete(self, news_id):
    cursor = self.connection.cursor()
    cursor.execute('DELETE FROM news WHERE id = ?', (str(news_id),))
    cursor.close()
    self.connection.commit()
```

Наш скрипт для удаления можно прочитать следующим образом: удалить из таблицы news все записи, у которых id равен news_id. Будьте внимательны, если вы выполните DELETE без указания блока WHERE, то удалятся все записи в таблице.

5. Создаём формы для веб-приложения

Ещё нам понадобится форма для добавления новости. Сама форма add_news.py:

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField, TextAreaField
from wtforms.validators import DataRequired

class AddNewsForm(FlaskForm):
    title = StringField('Заголовок новости', validators=[DataRequired()])
    content = TextAreaField('Текст новости', validators=[DataRequired()])
    submit = SubmitField('Добавить')
```

Для многострочного поля ввода лучше использовать класс **TextAreaField**.

И шаблон для формы в файле add_news.html:

```
{% extends "base.html" %}

{% block content %}
    <h1>Добавление новости</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.title.label }}<br>
            {{ form.title }}<br>
            {% for error in form.title.errors %}
                <div class="alert alert-danger" role="alert">
                    {{ error }}
                </div>
            {% endfor %}
        </p>
    </form>
{% endblock %}
```



```

        {% endfor %}
    </p>
    <p>
        {{ form.content.label }}<br>
        {{ form.content }}<br>
        {% for error in form.content.errors %}
            <div class="alert alert-danger" role="alert">
                {{ error }}
            </div>
        {% endfor %}
    </p>
    <p>{{ form.submit() }}</p>
</form>
{% endblock %}

```

← Я 127.0.0.1:8080 Добавление новости

★ Bookmarks Медиацентр ▾ Афиши и билеты ▾ Билинги и банки ▾ Кино и сериалы ▾ Музыка ▾ ОАО "ОЭП" ▾ Книги ▾ Ништяки ▾

Наше приложение

Добавление новости

Заголовок новости

Текст новости

Добавить

6. Авторизация пользователя в веб-приложении

Давайте внесём небольшие изменения в базовый шаблон. Будем отображать в нём имя пользователя, который зашёл в нашу систему, и сделаем так, что при клике на имя пользователя его бы перенаправляло на страницу выхода. Для этого достаточно в шапку шаблона (или ещё куда-нибудь на ваш выбор) добавить конструкцию вроде:

```

{% if "username" in session %}
    <a class="navbar-brand" href="/logout">{{session['username']}}</a>
{% endif %}

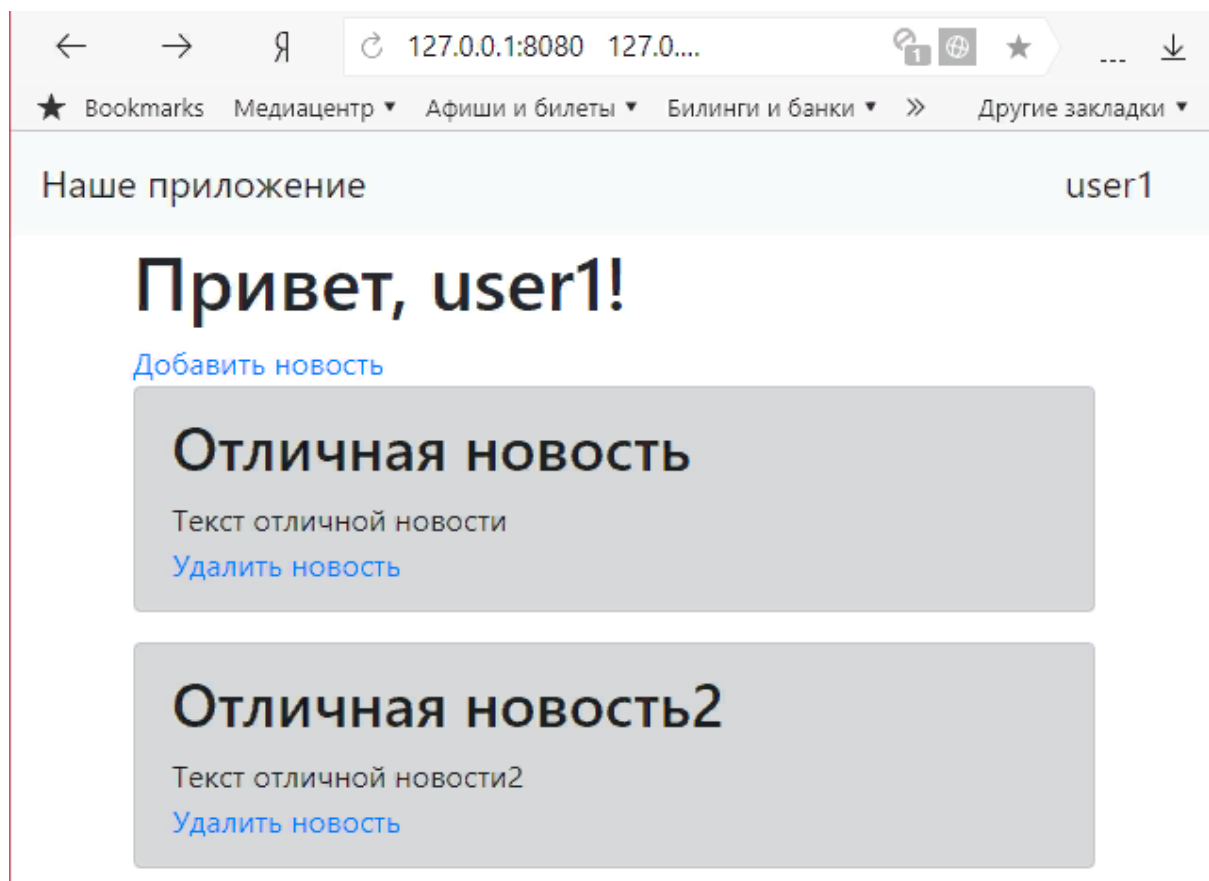
```

Мы уже говорили о том, что в шаблонах есть доступ к некоторым особым объектам Flask. К ним

относится и объект сессии, в котором мы можем хранить некоторую специфичную для подключенного пользователя информацию. Мы проверяем, есть ли информация о том, что пользователь залогинен и, если он залогинен (мы сами чуть позже запишем эту информацию), то отображаем его имя на форме.

Изменим ещё и главную страницу. Будем выводить на ней новости, которые добавлены залогиненным пользователем. А также добавим ссылку на страницу с созданием новости.

```
<a href="/add_news">Добавить новость</a>
{% for item in news %}
    <div class="alert alert-dark" role="alert">
        <h2>{{item[1]}}</h2>
        <div>{{item[2]}}</div>
        <a href="/delete_news/{{item[0]}}">Удалить новость</a>
    </div>
{% endfor %}
```



У нас почти всё готово, осталось добавить только обработчики для наших адресов. Пожалуй, начнём с обработки авторизации пользователя. На предыдущем уроке мы почти всё подготовили, осталось только проверить, есть ли пользователь с введённым логином и паролем в базе данных. Если есть, нужно записать его имя и id (для удобства) в сессию и отправить его на страницу с новостями. Объект **session** надо предварительно импортировать непосредственно из flask:

```

user_name = form.username.data
password = form.password.data
user_model = UsersModel(db.get_connection())
exists = user_model.exists(user_name, password)
if (exists[0]):
    session['username'] = user_name
    session['user_id'] = exists[1]
return redirect("/index")

```

Заодно давайте сделаем возможность выхода для пользователя, для этого нам достаточно «забыть» информацию о нём в сессии:

```

@app.route('/logout')
def logout():
    session.pop('username',0)
    session.pop('user_id',0)
    return redirect('/login')

```

На главной странице приложения будем отображать новости нашего пользователя, причём пускать на неё будем только авторизованного пользователя:

```

@app.route('/')
@app.route('/index')
def index():
    if 'username' not in session:
        return redirect('/login')
    news = NewsModel(db.get_connection()).get_all(session['user_id'])
    return render_template('index.html', username=session['username'],
                           news=news)

```

Если мы не находим информацию о пользователе, то мы просто перенаправляем его на страницу с формой авторизации.

Обратите внимание: чтобы пользователь мог войти в нашу систему, он вначале должен появиться в базе данных, а так как регистрации в веб-приложении пока нет, появиться он там может несколькими способами:

1. С помощью метода **insert** класса **UsersModel**
2. После выполнения SQL-скрипта на добавление пользователя в таблицу **users**

Кроме того, мы можем просмотреть содержимое базы данных в привычном для нас табличном представлении. Такая функция встроена в Professional редакцию PyCharm, но кроме этого есть достаточно много бесплатных инструментов, например, [DB Browser for SQLite](#).

Добавьте пользователя любым из способов, чтобы у него появилась возможность авторизоваться в нашем веб-приложении.

Создадим страницу добавления новости и страницу удаления новости, на которые есть доступ также только у авторизованных пользователей:

```
@app.route('/add_news', methods=['GET', 'POST'])
def add_news():
    if 'username' not in session:
        return redirect('/login')
    form = AddNewsForm()
    if form.validate_on_submit():
        title = form.title.data
        content = form.content.data
        nm = NewsModel(db.get_connection())
        nm.insert(title, content, session['user_id'])
        return redirect("/index")
    return render_template('add_news.html', title='Добавление новости',
                           form=form, username=session['username'])

@app.route('/delete_news/<int:news_id>', methods=['GET'])
def delete_news(news_id):
    if 'username' not in session:
        return redirect('/login')
    nm = NewsModel(db.get_connection())
    nm.delete(news_id)
    return redirect("/index")
```

Как вы можете заметить, при удалении мы не используем метод DELETE протокола HTTP, так как не все браузеры умеют корректно отправлять такой запрос. Но он нам обязательно пригодится на следующем уроке.

7. Идеи для улучшения веб-приложения

Авторизация, которую мы с вами реализовали, довольно примитивна, но для Flask есть готовые модули, которые помогут вам сделать её более профессионально и даже подключить авторизацию через различные социальные сети. Кроме того, как мы уже упоминали раньше, надо скорее избавляться от хранения паролей в открытом виде в базе данных. Это можно сделать как через стандартные функции хэширования, так и через модуль, который устанавливается вместе с Flask (как зависимость, он не является частью Flask):

```
from werkzeug.security import generate_password_hash

hash = generate_password_hash('yandexlyceum')
```

Следует отметить, что мы работали с базой данных напрямую через язык SQL, что вполне оправдано на небольших проектах вроде нашего. Однако в больших веб-приложениях достаточно часто используется **ORM** — прослойка, позволяющая работать с базой данных через объекты языка. Большинство ORM позволяют генерировать скрипты миграции базы данных для поддержания версионности (отдалённо можно сравнить с git, но для баз данных), а также ещё немало другой полезной функциональности.

Рекомендуем посмотреть в сторону [sqlalchemy](#), а для упрощения знакомства рассмотрим небольшой пример.

8. Первый взгляд на sqlalchemy

Поскольку мы говорим о Flask, то с sqlalchemy познакомимся на примере библиотеки **flask-sqlalchemy**. Эта библиотека — обёртка над sqlalchemy, которая добавляет ещё немного удобства к том, что sqlalchemy предоставляет сама по себе. К тому же, **flask-sqlalchemy** прекрасно встраивается в ваши Flask-приложения. Для начала надо установить библиотеку:

```
pip3 install flask_sqlalchemy
```

После установки из библиотеки надо импортировать класс **SQLAlchemy**:

```
from flask_sqlalchemy import SQLAlchemy
```

Давайте создадим класс студента Яндекс.Лицея, который будет содержать следующую информацию:

1. уникальный идентификатор студента,
2. уникальное имя пользователя,
3. уникальный почтовый ящик,
4. имя студента,
5. фамилию,
6. название группы,

7. год обучения.

Для начала надо сказать, где нам искать нашу базу данных. Эту информацию flask-sqlalchemy берёт из параметра **SQLALCHEMY_DATABASE_URI**. Добавим этот параметр в наше Flask-приложение, а также установим **SQLALCHEMY_TRACK_MODIFICATIONS** в значение False (за что отвечает этот параметр — посмотрите самостоятельно), а также создадим объект класса **SQLAlchemy**:

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

Во flask-sqlalchemy для создания сущности нам достаточно будет унаследовать наш класс от класса **db.Model** и написать вот такую конструкцию:

```
class YandexLyceumStudent(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    name = db.Column(db.String(80), unique=False, nullable=False)
    surname = db.Column(db.String(80), unique=False, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    group = db.Column(db.String(80), unique=False, nullable=False)
    year = db.Column(db.Integer, unique=False, nullable=False)

    def __repr__(self):
        return '<YandexLyceumStudent {} {} {} {}>'.format(
            self.id, self.username, self.name, self.surname)
```

Мы создали класс, который содержит всю необходимую для нас информацию. Свойство **unique** означает, что значения в этом поле не могут повторяться для разных записей, а **nullable** — может ли быть поле пустым. SQLAlchemy сделает за нас всё остальное. Создадим сущности в нашей базе данных:

```
db.create_all()
```

После этого у нас появится файл с базой данных (если его до этого не было), а в этой базе создастся таблица вот такой структуры:

▼	📄	yandex_lyceum_student
	🔑	id INTEGER
	📄	username VARCHAR(80)
	📄	name VARCHAR(80)
	📄	surname VARCHAR(80)
	📄	email VARCHAR(120)
	📄	group VARCHAR(80)
	📄	year INTEGER
	🔑	<unnamed> (id)
	🔑	<unnamed> (username)
	🔑	<unnamed> (email)
	📄	sqlite_autoindex_yandex_lyceum_student_1 (username) UNIQUE
	📄	sqlite_autoindex_yandex_lyceum_student_2 (email) UNIQUE

Чтобы добавить нового ученика Яндекс.Лицея, достаточно создать объект класса **YandexLyceumStudent**:

```
user1 = YandexLyceumStudent(username='student1',
                             email='student1@yandexlyceum.ru',
                             name='Иван',
                             surname='Иванов',
                             group='Пенза, Лицей 2',
                             year=2)

user2 = YandexLyceumStudent(username='student2',
                             email='student2@yandexlyceum.ru',
                             name='Петр',
                             surname='Петров',
                             group='Москва, Лицей 1234',
                             year=1)
```

Чтобы объекты появились в базе, надо записать их в сессию и применить изменения:

```
db.session.add(user1)
db.session.add(user2)
db.session.commit()
```

Теперь мы легко можем получить всех студентов или, например, получить первого студента с именем «Петр»:

```
print(YandexLyceumStudent.query.all())
print(YandexLyceumStudent.query.filter_by(name='Петр').first())
```

```
[<YandexLyceumStudent 1 student1 Иван Иванов>, <YandexLyceumStudent 2 student2 Пе  
<YandexLyceumStudent 2 student2 Петр Петров>
```

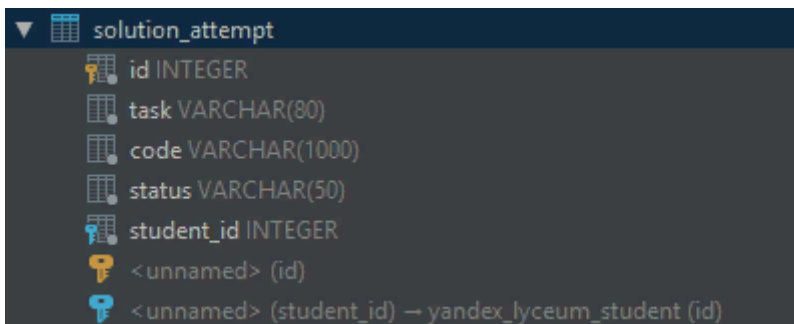
Давайте немного усложним пример и добавим для каждого нашего ученика посылки задач, которые будут содержать следующую информацию:

1. уникальный номер посылки,
2. задача, к которой относится посылка,
3. код посылки,
4. статус посылки.

```
class SolutionAttempt(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    task = db.Column(db.String(80), unique=False, nullable=False)
    code = db.Column(db.String(1000), unique=False, nullable=False)
    status = db.Column(db.String(50), unique=False, nullable=False)
    student_id = db.Column(db.Integer,
                           db.ForeignKey('yandex_lyceum_student.id'),
                           nullable=False)
    student = db.relationship('YandexLyceumStudent',
                              backref=db.backref('SolutionAttempts',
                                                    lazy=True))

    def __repr__(self):
        return '<SolutionAttempt {} {} {}>'.format(
            self.id, self.task, self.status)
```

student_id и student нужны для создания связи между сущностями «Ученик» и «Посылка решения». Запустим наше приложение, в нашей базе данных появится сущность **solution_attempt**:



solution_attempt	
id	INTEGER
task	VARCHAR(80)
code	VARCHAR(1000)
status	VARCHAR(50)
student_id	INTEGER
<unnamed> (id)	
<unnamed> (student_id) → yandex_lyceum_student (id)	

Теперь давайте получим из базы данных первого Петра и добавим ему посылку:


```
user = YandexLyceumStudent.query.filter_by(name='Петр').first()
attempt = SolutionAttempt(task='Первая задача',
                           code='print("Привет, Яндекс!")',
                           status='OK')
user.SolutionAttempts.append(attempt)
db.session.commit()
```

В нашей базе появилась посылка, относящаяся к пользователю Петр:

	id	task	code	status	student_id
1	1	Первая задача	print("Привет, Яндекс!")	OK	2

Это только самое базовое знакомство, возможности **flask_sqlalchemy** намного шире. Об этом можно почитать в официальной документации, которая находится [здесь](#). Удачи в освоении!