

## **Лабораторная работа 7**

### **Тема: Разработка веб-приложений на основе фреймворка Flask**

Цель. Научиться создавать веб-приложения на основе фреймворка Flask, подключаться и выполнять запросы к базе данных в веб-приложении.

#### *Задания*

Разработать веб-приложение, предназначенное для обработки данных в предметной области, соответствующей закреплённой за магистром теме выпускной квалификационной работы. В веб-приложении должны быть реализованы **следующие** задачи:

- 1 **Аутентификация и авторизация** пользователей веб-приложения.
- 2 Реализация навигационного меню по страницам: «Главная», «Аналитика», «О программе», «Регистрация».
- 3 На странице «Главная» должна отображаться общая информация об объекте автоматизации.
- 4 На странице «Аналитика» реализовать подключение функций из лабораторной работы 6, выполняющих регрессию, классификацию или кластеризацию объектов предметной области выпускной квалификационной работы магистра.
- 5 На странице «О программе» представить краткую информацию о приложении и об авторе.
- 6 На странице «Регистрация» реализовать функционал по регистрации пользователей веб-приложения.

#### **1.1 Порядок выполнения лабораторной работы**

- 1 Ознакомиться с целью, задачами, содержанием лабораторной работы, общими методическими указаниями к выполнению.

2 Изучить основы работы с библиотеками Flask, flask\_wtf, sqlite3 и - синтаксическим анализатором шаблонов Jinja2.

3 Подготовить исходные данные к работе. Для этого нужно придумать одномерные или двумерные наборы данных по предметной области ВКР и сохранить их в файле в формате .csv.

4 В системе Visual Studio Code или PyCharm выполнить **практические задания** по разработке веб-приложения как общие, представленные в тексте теоретических предпосылок, так и согласно, соответствующей закреплённой за магистром теме выпускной квалификационной работы.

5 Оформить отчет.

### Краткие теоретические предпосылки

Flask – легкий веб-фреймворк, предлагающий полезные инструменты и функции для облегчения процесса создания веб-приложений с использованием Python. Flask – расширяемая система, которая не обязывает использовать конкретную структуру директорий и не требует сложного шаблонного кода перед началом использования.

Flask использует механизм шаблонов **Jinja** для динамического создания HTML-страниц с использованием знакомых понятий в Python, таких как переменные, циклы, списки и т. д.

**Упражнение 1. Установить Flask**, используя установщик pip:

[pip install Flask](#)

**Упражнение 2. Создание минимального приложения**

Создать файл main.py с кодом

Номер строки	Код
1	<code>from flask import Flask</code>
2	<code>app = Flask(__name__)</code>
3	<code>@app.route("/")</code>
4	<code>def home():</code>
5	<code>    return "Hello, World!"</code>

6	<code>if __name__ == "__main__":</code>
7	<code>app.run(debug=True)</code>

Строка 1: импорт модуля Flask и создание веб-сервера Flask из модуля Flask.

Строка 2: В данном случае это будет main.py.

создаем новое веб-приложение путем создания экземпляра класса Flask с названием app. **\_\_name\_\_** означает текущий файл, который будет представлять веб-приложение .

Строка 3: представляет страницу по умолчанию.

Строка 4–5: функция **home** активируется когда пользователь переходит на веб-сайт и переходит на страницу по умолчанию

Строка 6: **условие проверяет имя запущенного скрипта** Python, Скрипту присваивается автоматически имя «\_\_main\_\_» при выполнении.

Если импортировать другой скрипт, оператор if предотвратит запуск других скриптов. При запуске main.py, **он** изменит свое имя на \_\_main\_\_ и только тогда активируется оператор if.

Строка 7: запустит приложение. Наличие debug=True позволяет отображать возможные ошибки Python на веб-странице для отслеживания ошибок программистом.

### **Упражнение 3. Запуск веб-приложения**

В терминале или командной строке нужно перейти в папку, содержащую созданный файл *main.py*. Затем выполнить

**python main.py**

В терминале или командной строке появится вывод.

```
Администратор: D:\PYTHON_progr\flask_app1\cmd.exe - python main.py
Microsoft Windows [Version 6.9.9600]
(c) Корпорация Майкрософт (Microsoft Corporation), 2013. Все права защищены.

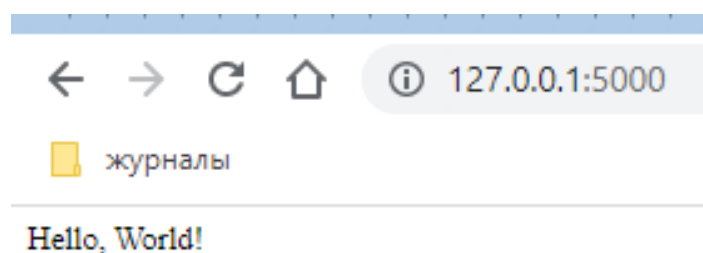
d:\PYTHON_progr\flask_app1>python main.py
* Serving Flask app 'main' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 134-632-069
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Важная часть – в строке:

`Running on http://127.0.0.1:5000/.`

127.0.0.1 означает локальный компьютер, на котором запущен скрипт. Т.е. 127.0.0.1 и localhost относятся к данному локальному компьютеру.

Перейдите по этому адресу, и **вы** должны увидеть следующее:



## HTML и шаблоны в Flask

Созданная первая функция `home` возвращает текст на страницу. Для оформления сайта, нужно добавить HTML и CSS.

### Упражнение 4. Оформление страниц веб-приложения

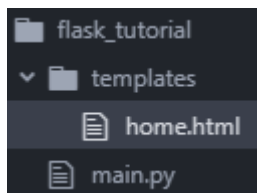
1) Сначала **создайте** новый файл HTML файл – **home.html**.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Flask Tutorial</title>
  </head>
  <body>
    <h1> My First Try Using Flask </h1>
    <p> Flask is Fun </p>
```

```
</body>
</html>
```

*Важный момент*, который следует помнить: Flask Framework ищет HTML-файлы в папке с именем `templates`.

2) Поэтому нужно создать папку шаблонов `templates` и поместить туда все файлы HTML.



Не **забывайте** всегда хранить `main.py` вне папки с **вашими** шаблонами.

3) Теперь нужно изменить файл `main.py`, чтобы можно было просматривать созданный нами файл HTML.

№ строки	Код	
1.	<code>from flask import Flask</code>	<code>from flask import Flask, render_template</code>
2.	<code>app = Flask(__name__)</code>	<code>app = Flask(__name__)</code>
3.	<code>@app.route("/")</code>	<code>@app.route("/")</code>
4.	<code>def home():</code>	<code>def home():</code>
5.	<code>    return "Hello, World!"</code>	<code>    return render_template("home.html")</code>
6.	<code>if __name__ == "__main__":</code>	<code>if __name__ == "__main__":</code>
7.	<code>    app.run(debug=True)</code>	<code>    app.run(debug=True)</code>

Строка 1: импортировали метод `render_template()` из фреймворка `flask`, который ищет указанный в качестве входного параметра шаблон (файл HTML) в папке шаблонов, затем отображает этот шаблон. **Узнайте больше о функции `render_templates()`**.

Строка 5: Изменяем `return` так, чтобы теперь он возвращал `render_template("home.html")`. Это позволит просмотреть указанный в параметрах HTML-файл.

**Запуск веб-приложения.** Если сервер не запущен, его нужно запустить и **посмотреть** изменения: <http://localhost:5000/>.



## Добавление страниц

Если веб-приложение состоит более чем из одной страницы, то нужна навигация чтобы соединить все страницы и удобно перемещаться между ними. Можно создать меню навигации вверху страницы.

Кроме того, все страницы должны иметь одинаковый стиль, общее меню навигации, а это приводит к повтору кода. Чтобы избежать повтора кода на основе синтаксического анализатора шаблонов для Flask Jinja2 создают страницу-шаблон, от которой наследуют общие части в других страницах.

**Упражнение 5. Работа с наследованием шаблонов.** Нужно создать шаблоны: *template.html* – *родитель*, *about.html* и *home.html* – *потомки*. Затем изменить их обработчики.

Во-первых, создадим файл **template.html**. Этот *template.html* будет служить родительским шаблоном. Два наших дочерних шаблона будут наследовать от него код.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Flask Parent Template</title>
  </head>
  <body>
    <header>
      <div class="container">
        <h1 class="logo">First Web App</h1>
        <strong><nav>
          <ul class="menu">
            <li><a href="{{ url_for('home') }}">Home</a></li>
            <li><a href="{{ url_for('about') }}">About</a></li>
```

```

        </ul>
    </nav></strong>
</div>
</header>

```

```

{% block content %}
{% endblock %}

```

```

</body>
</html>

```

Строка 13–14: мы используем функцию `url_for()` с именем `home`. Он принимает в качестве аргумента имя функции. [Подробнее о функции `url\_for\(\)`](#).

Две строки в фигурных скобках будут заменены содержимым `home.html` и `about.html`.

Эти изменения позволяют дочерним страницам (`home.html` и `about.html`) подключаться к родительской (`template.html`). Это позволяет не копировать код меню навигации в `about.html` и `home.html`.

**Упражнение 5. Работа с наследованием шаблонов.** Создадим потомки шаблонов `about.html` и `home.html` в папке шаблонов. Старый `home.html` нужно удалить.

Содержимое `about.html`:

```

{% extends 'template.html' %}

{% block content %}

{{ super() }}

Содержимое страницы о Сайте

{% endblock %}

```

Содержимое `home.html`:

```

{% extends 'template.html' %}

{% block content %}

{{ super() }}

```

Содержимое главной страницы

```
{% endblock % }
```

Также нужно внести изменения в наш *main.py* для новой страницы *about.html*.

Номер строки	Код
1	<code>from flask import Flask, render_template</code>
2	<code>app = Flask(__name__)</code>
3	<code>@app.route("/")</code>
4	<code>def home():</code>
5	<code>    return render_template("home.html")</code>
6	<code>@app.route("/about")</code>
7	<code>def about():</code>
8	<code>    return render_template("about.html")</code>
9	<code>if __name__ == "__main__":</code>
10	<code>    app.run(debug=True)</code>

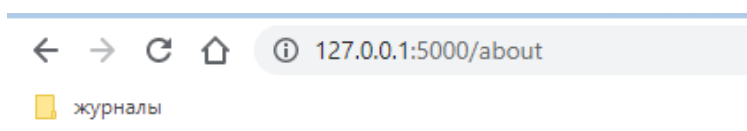
Внесли изменения:

Строка 6: Указан маршрут на `"/about"`

Строка 7: Определили функцию `def about():`

Строка 8: `return` возвращает `render_template("about.html")`.

Теперь смотрите изменения: <http://localhost:5000/about> .



## First Web App

- [Home](#)
- [About](#)

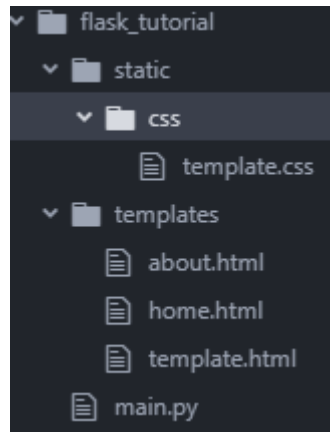
Содержимое страницы о Сайте

### Добавление CSS на сайт

Для хранения всех HTML-шаблонов, как правило создается папка с именем `static` .

В `static` будут храниться CSS, JavaScript, изображения и другие необходимые файлы. После этого папка вашего проекта должна выглядеть так:





**Упражнение 7. Добавление CSS.** В папке templates создать файл template.css с содержимым

```
body {  
    font-family: 'Montserrat', sans-serif;  
}  
  
.logo {  
    background-color: #0D122B;  
    padding: 10px;  
    color: white;  
}
```

### **Упражнение 8. Связывание CSS с файлом HTML**

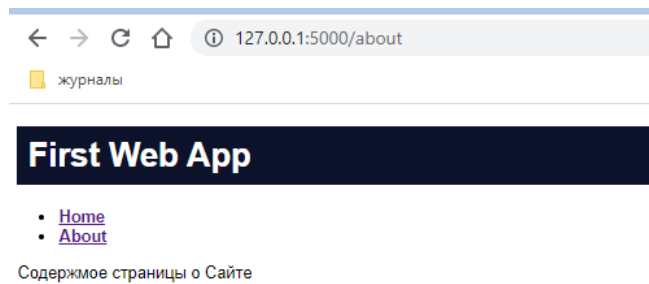
Шаблон template.html связывает все страницы. Можно вставить после тега:

```
<title>Flask Parent Template</title>
```

Строку для указания пути к местоположению template.css, который будет применим ко всем дочерним страницам.

```
<!DOCTYPE html>  
<html lang="en" dir="ltr">  
  <head>  
    <meta charset="utf-8">  
    <title>Flask Parent Template</title>  
    <link rel="stylesheet" href="{{ url_for('static', filename='css/template.css') }}">
```

**Запуск веб-приложения.** Теперь можно посмотреть изменения: <http://localhost:5000/about> .



## Веб-формы

Классически в HTML формы описываются на странице с помощью тега `form`, в параметре `action` указывается URL, который должен принимать данные от формы, а параметр `method` определяет способ передачи данных. В основном, используются два вида передачи:

- GET – в виде строки запроса:  
"/handler?name=Alex&old=18&profit=1000";
- POST – в виде бинарных данных (используется для передачи больших объемов данных: изображений, звуков, документов и т.п., а также закрытых сведений: паролей, логинов и т.п.).

### Упражнение 9. Работа с классической веб-формой.

1) Добавить шаблон **contact.html** с формой:

```
{% extends 'template.html' %}

{% block content %}
{{ super() }}
<form action="/contact" method="post" class="form-contact">
<p><label>Имя: </label> <input type="text" name="username" value="" required />
<p><label>Email: </label> <input type="text" name="email" value="" required />
<p><label>Сообщение:</label>
<p><textarea name="message" rows=7 cols=40></textarea>
<p><input type="submit" value="Отправить" />
</form>
{% endblock %}
```

Здесь указали способ отправки данных в виде POST-запроса и обработчик «/contact», которому будут переданы данные из формы.

2) Добавить стили для оформления формы (в файле `template.css`):

```
.form-contact label {
    display: inline-block;
    min-width: 80px;
}

.form-contact p {margin: 10px 0 10px 0;}
.form-contact input[type=submit], textarea {
```

```
font-size: 16px;  
}
```

### 3) Добавим функцию обработчик в файл main.py

В обработчике нужно явно указать: может ли он принимать данные методом **POST**. Для этого нужно прописать параметр **methods** со значением **POST** как элемент списка

```
@app.route("/contact", methods=["POST", "GET"])  
def contact() :  
    if request.method=='POST':  
        print(request.form)  
    return render_template("contact.html")
```

### 4) Добавить пункт contact в меню в template.html

```
<li><a href="{{ url_for('contact') }}">Contact</a></li>
```

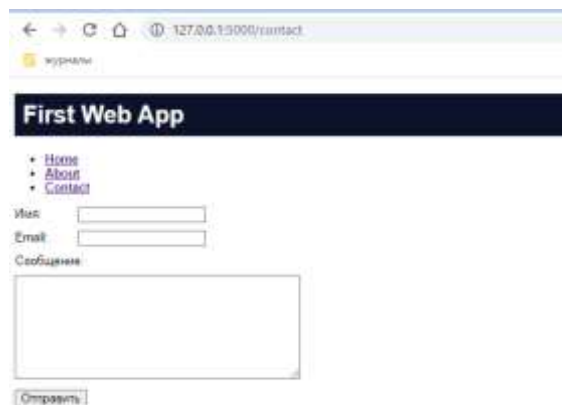
импортировать объекты:

```
from flask import Flask, render_template, request
```

### 5) Запустить программу, откроем в браузере страницу

<http://127.0.0.1:5000/contact>

Результат:



В обработчике проверяется: пришел ли именно POST-запрос, а не какой-либо другой и выводятся данные в консоль:

```
if request.method == 'POST':  
    print(request.form)
```

Все данные формы доступны по свойству `form`, которое представляет своеобразный словарь, то есть, для доступа к конкретному полю можно использовать следующую запись:

```
print(request.form['username'])
```

<https://proproprogs.ru/flask/primenenie-wtforms-dlya-raboty-s-formami-sayta>

## Библиотека WTForms

WTForms – это библиотека, написанная на Python и независимая от фреймворков. WTForms позволяет достаточно просто оперировать формами, способна генерировать формы, проверять их, наполнять начальной информацией, работать с reCaptcha, в нее встроена защита от CSRF – межсайтовая подделка запросов и многое другое.

Расширение WTForms для Flask называется Flask-WTF и устанавливается с помощью команды:

```
pip install flask_wtf
```

**FlaskForm** – это обертка, содержащая полезные методы для оригинального класса `wtform.Form`, который является основной для создания форм. Формы создают путем расширения базового класса `FlaskForm` из пакета `flask_wtf`. Внутри класса формы, элементы формы определяются в виде полей этого класса, ссылающихся на объекты, образованные из встроенных типов (классов), таких как:

- `StringField` – для работы с полем ввода;
- `PasswordField` – для работы с полем ввода пароля;
- `BooleanField` – для checkbox полей;
- `TextAreaField` – для работы с вводом текста;
- `SelectField` – для работы со списком;
- `SubmitField` – для кнопки submit.

Это лишь часть классов. Полную документацию можно посмотреть на [официальном сайте](#).

## Упражнение 10. Создание класса формы flask\_wtf

Установить библиотеку `flask_wtf`. Создать в проекте вспомогательный файл `forms.py`, в котором будут определены все классы форм и создать класс `ContactForm`:

<https://pythonru.com/uroki/11-rabota-s-formami-vo-flask>

Для начала нужно создать файл **forms.py** с кодом.

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

class ContactForm(FlaskForm):
    name = StringField("Name: ", validators=[DataRequired()])
    email = StringField("Email: ", validators=[Email()])
    message = TextAreaField("Message", validators=[DataRequired()])
    submit = SubmitField("Submit")
```

Здесь определен класс формы `ContactForm` с четырьмя полями: *name*, *email*, *message* и *submit*. Эти переменные будут использоваться, чтобы *отрендерить* поля формы, а также назначать и получать информацию из них. Эта форма создана с помощью двух объектов *StringField*, *TextAreaField* и *SubmitField*.

Конструктору объекта-поля передаются **аргументы**. Первый аргумент — строка, содержащая метку, которая будет отображаться внутри тега `<label>` в тот момент, когда поле отрендерится. Второй аргумент — список валидаторов (элементов системы проверки), которые передаются конструктору в виде аргументов-ключевых слов через запятую. **Валидаторы** определяют, корректна ли введенная в поле информация. Модуль **wtforms.validators** предлагает базовые валидаторы, но их можно создавать самостоятельно. В этой форме используются два встроенных валидатора: **DataRequired** и **Email**.

**DataRequired:** он проверяет, ввел ли пользователь хоть какую-то информацию в поле.

**Email:** проверяет, является ли введенный электронный адрес действующим. Валидатор Email может требовать отдельной дополнительной установки:

`pip install email-validator`

Введенные данные не будут приняты до тех пор, пока валидатор не подтвердит соответствие данных.

**Примечание:** Полный список полей форм и валидаторов доступен в официальном руководстве по ссылке <https://wtforms.readthedocs.io>.

### Установка SECRET\_KEY

По умолчанию Flask-WTF предотвращает любые варианты CSFR-атак. Это делается с помощью встраивания специального токена в скрытый элемент `<input>` внутри формы. Затем этот токен используется для проверки подлинности запроса. До того как Flask-WTF сможет сгенерировать csrf-токен, необходимо добавить секретный ключ.

**Упражнение 11.** Установить SECRET\_KEY в файле `main.py` следующим образом:

```
#...
app.debug = True
app.config['SECRET_KEY'] = 'a really really really really long secret key'

manager = Manager(app)
#...
```

Здесь используется атрибут `config` объекта `Flask`. Атрибут `config` работает как словарь и используется для размещения параметров настройки `Flask` и расширений `Flask`, но их можно добавлять и самостоятельно.

Секретный ключ должен быть строкой – такой, которую сложно разгадать и, желательно, длинной. SECRET\_KEY используется не только для создания CSFR-токенов. Он применяется и в других расширениях `Flask`. Секретный ключ должен быть безопасно сохранен. Вместо того чтобы хранить его в приложении, лучше разместить в переменной окружения.

Важно заметить, что вне зависимости от используемого метода нужно вручную добавлять тег `<form>`, чтобы обернуть поля формы.

**Упражнение 11.** Нужно удалить старый и **создать** новый шаблон **contact.html** со следующим кодом:

```
<!DOCTYPE html>

{% extends 'template.html' %}

{% block content %}

{{ super() }}

<form action="" method="post">

    {{ form.csrf_token() }}

    {% for field in form if field.name != "csrf_token" %}

        <p>{{ field.label() }}</p>

        <p>{{ field }}

            {% for error in field.errors %}

                {{ error }}

            {% endfor %}

        </p>

    {% endfor %}

</form>

{% endblock %}
```

Функция представления будет создана далее.

### Работа с подтверждением формы

При разработке формы важной задачей является проверка корректности введенных пользователем данных формы. Это можно сделать методом **validate\_on\_submit()**.

**Упражнение 12.** Изменение обработчика **contact()**

Изменить в **main.py** код обработчика **contact()** следующим образом:

```
from flask import Flask, render_template, request, redirect, url_for
from flask_script import Manager, Command, Shell
```

```

from forms import ContactForm

#...

@app.route('/contact/', methods=['get', 'post'])
def contact():
    form = ContactForm()

    if form.validate_on_submit():
        name = form.name.data
        email = form.email.data
        message = form.message.data

        print(name)
        print(email)
        print(message)

        # здесь логика базы данных

        print("\nData received. Now redirecting ...")

        return redirect(url_for('contact'))

    return render_template('contact.html', form=form)

#...

```

В 7 строке создается объект формы. На 8 строке проверяется значение, которое вернул метод **validate\_on\_submit()** для исполнения кода внутри инструкции **if**.


Процедура проверки запускается только в том случае, если данные были отправлены с помощью метода POST. В противном случае вернется *False*. Метод **validate\_on\_submit()** вызывает внутри себя метод **validate()**, который всего лишь проверяет, корректны ли данные формы. Он не проверяет, был ли запрос отправлен с помощью метода POST. Также нужно обратить внимание, что при создании экземпляра объекта формы данные не передаются, потому что когда форма отправляется с помощью запроса POST, WTForm считывает данные формы из атрибута **request.form**.



```
form.name.data # доступ к данным в поле name.
form.email.data # доступ к данным в поле email.
```

```
form.data # доступ ко всем данным
```

Когда форма отправляется с помощью запроса POST, `validate_on_submit()` возвращает `True`, предполагая, что данные верны. Вызовы `print()` внутри блока `if` выведут данные, введенные пользователем, а функция `redirect()` перенаправит пользователя на страницу `/contact/`. С другой стороны, если `validate_on_submit()` вернет `False`, выполнение инструкций внутри тела `if` будет пропущено, и появится сообщение об ошибке валидации.



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:5000/contact'. The page title is 'First Web App'. Below the title is a navigation menu with links: 'Home', 'About', and 'Contact'. The 'Contact' link is highlighted in purple. Below the menu are three form fields: 'Name:' with a text input containing '2101010101', 'Email:' with an empty text input, and 'Message:' with a larger text area. At the bottom, there is a 'Submit' button and a 'Go back' button.

## Работа с БД в веб-приложении

Изменяемая информация веб-приложения сохраняется в БД, затем, используется при формировании ответов на запросы пользователей.

Для учебных целей достаточно использовать СУБД SQLite, которая поставляется вместе с Python3.

Первое, что нужно сделать при работе с базой данных SQLite (и большинством других библиотек баз данных Python), – это создать к ней подключение.

### **Упражнение 13. Создание подключения к БД.**

1) Для начала работы с SQLite выполнить импорт дополнительных пакетов:

```
import sqlite3
import os
from flask import Flask, render_template, request
```

2) Выполнить **конфигурацию** веб-приложения. Во Flask принято **соглашение**: все переменные, записанные *заглавными* буквами, относятся к *конфигурационной* информации. В дальнейшем конфигурацию нужно определять в отдельном файле. Пользуясь соглашениями, определить следующие настройки:

```
# конфигурация
DATABASE = '/tmp/siteCS.db'
DEBUG = True
SECRET_KEY = 'fdgfh78@#5?>gfhf89dx,v06k'
USERNAME = 'admin'
PASSWORD = '123'
```

SECRET\_KEY. Он необходим для безопасной работы сессий на стороне клиента. С помощью этого секретного ключа выполняется шифрование данных, которые, затем, сохраняются в куках браузера.

3) Создать само приложение и загрузить конфигурацию из текущего модуля:

```
app = Flask(__name__)
app.config.from_object(__name__)
```

4) Далее, переопределить в конфигурации значение DATABASE, расположив БД в текущем каталоге приложения:

```
app.config.update(dict(DATABASE=os.path.join(app.root_path, 'flsite.db')))
```

5) Определить в файле **main.py** общую функцию для установления соединения с БД:

```
def connect_db():
    conn = sqlite3.connect(app.config['DATABASE'])
    conn.row_factory = sqlite3.Row
    return conn
```

### Упражнение 14. Создание БД

Для примера будет рассмотрено создание таблицы **device**, в которой будет храниться информация об устройствах.

Сохранить его в файле **'sq\_db.sql'** SQL-запрос для формирования таблицы **device** с полями: **id**, **name** и **IPaddr**:

```
CREATE TABLE IF NOT EXISTS device (
    id integer PRIMARY KEY AUTOINCREMENT,
    name text NOT NULL,
    IPaddr text NOT NULL,
    time integer NOT NULL
);
```

Для создания начальной БД с набором необходимых таблиц можно сначала объявить вспомогательную функцию **create\_db**:

```
def create_db():
    """Вспомогательная функция для создания таблиц БД"""
    db = connect_db()
    with app.open_resource('sq_db.sql', mode='r') as f:
        db.cursor().executescript(f.read())
    db.commit()
    db.close()
```

В функции использован метод **open\_resource**, который открывает файл **'sq\_db.sql'** на чтение, расположенный в рабочем каталоге созданного приложения. Затем, для открытой БД выполняется скрипт, записанный в файле **'sq\_db.sql'**. В конце вызывается метод **commit**, чтобы изменения применились к текущей БД, и метод **close** закрывает установленное соединение.

Затем уже создать начальную БД можно выполнив в консоле Python импорт функции **create\_db** из файла **main.py**:

```
from main import create_db
```

И, затем, вызвать эту функцию:

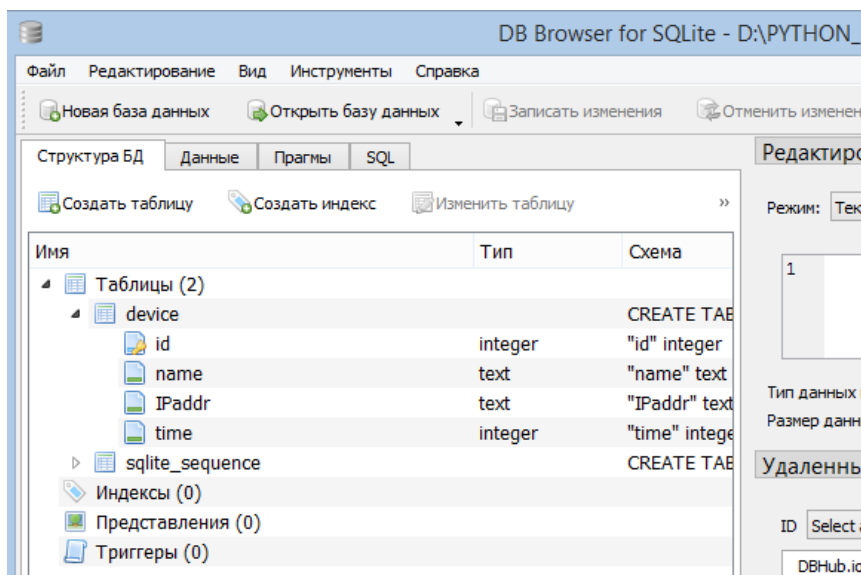
```
create_db()
```

```
d:\PYTHON_progr\flask_app12\cmd.exe
d:\PYTHON_progr\flask_app12>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> from main4 import create_db
>>> create_db()
>>> exit()

d:\PYTHON_progr\flask_app12>
```

Имя	Тип	Размер	Дата
..			<Папка> 18 04 2022
__pycache__			<Папка> 18 04 2022
static			<Папка> 12 04 2022
templates			<Папка> 12 04 2022
cmd	exe	357 376	21 11 2014
forms	py	405	12 04 2022
index	html	228	16 01 2022
main	py	169	16 01 2022
readme	txt	58	12 04 2022
siteCS	db	12 288	18 04 2022
sq_db	sql	147	18 04 2022

Все, в результате, в рабочем каталоге разрабатываемого приложения появится файл **siteCS.db** с таблицей **device**. В этом можно легко убедиться, если открыть эту БД с помощью специальной программы DB Browser SQLite.



## Подключение к БД в запросах

Любые запросы и операции выполняются с использованием соединения, которое закрывается после завершения работы.

В веб-приложениях это соединение обычно привязано к запросу. Он создается в какой-то момент при обработке запроса и закрывается перед отправкой ответа.

Момент поступления запроса можуж «поймать» непосредственно в обработчике. Для этого можно оформить функцию **get\_db**, которая возвращает активное соединение с БД.

#### **Упражнение 16. Подключение к БД в запросах.**

1) Добавить в **main.py** функцию **get\_db**:

```
def get_db():
    '''Соединение с БД, если оно еще не установлено'''
    if not hasattr(g, 'link_db'):
        g.link_db = connect_db()
    return g.link_db
```

В функции **get\_db** использован объект **g** контекста приложения, которое создается в момент поступления запроса. В этом объекте можно сохранять любую пользовательскую информацию, которая будет доступна в любой функции и шаблонах, в пределах этого запроса. Причем, для разных запросов, объект **g** будет разным, то есть, он уникален в пределах текущего запроса. Далее, нужно проверить: было ли соединение уже установлено (существует ли атрибут **link\_db**, который создается в момент соединения с БД. Если соединения еще нет, то устанавливают его и, затем, возвращают. Т.о. если где-либо в функциях (или шаблонах) будет повторное обращение к этой функции, то она просто возвратит ранее установленное соединение.

2) Нужно импортировать **g**.

```
from flask import Flask, render_template, url_for, request, flash, session,
redirect, abort, g
```

После установления связи с БД можно производить различные запросы, а затем **разорвать связь** в момент завершения запроса. Для в Flask есть специальный декоратор **teardown\_appcontext**, который позволяет определять

функцию, вызываемую в момент уничтожения контекста приложения. А это, обычно, происходит в момент завершения обработки запроса.

3) Добавить в **main.py** обработчик для завершения соединения с БД:

```
@app.teardown_appcontext
def close_db(error):
    '''Закрываем соединение с БД, если оно было установлено'''
    if hasattr(g, 'link_db'):
        g.link_db.close()
```

Готова БД, к которой можно получить доступ в момент возникновения запроса и автоматическое завершение соединения при окончании работы с запросом.

Для удобства работы с запросами к БД можно создать вспомогательный класс, например, **FDataBase**, который запоминает ссылку на БД и, затем, с помощью специальных методов возвращает данные для отображения в шаблонах \*.html.

4) В новом файле **FDataBase.py**, добавить класс FDataBase:

```
import sqlite3

class FDataBase:
    def __init__(self, db):
        self.__db = db
        self.__cur = db.cursor()

    def getDevice(self):
        sql = '''SELECT * FROM device'''
        try:
            self.__cur.execute(sql)
            res = self.__cur.fetchall()
            if res: return res
        except:
            print("Ошибка чтения из БД")
        return []
```

Здесь в конструкторе запоминается ссылка на БД и ссылка на класс курсор, через который осуществляется взаимодействие с таблицами этой БД. Далее, идет метод **getDevice** и в блоке try/except осуществляется *выборка всех записей* из таблицы **device**. Если операция прошла успешно, то возвращается **список словарей из записей, а иначе – пустой список.**

5) Добавить метод **addDevice** в класс **FDataBase**:

```
def addDevice (self, name, IPaddr):
    try:
        tm = math.floor(time.time())
```

```

        self.__cur.execute("INSERT INTO device VALUES (NULL, ?, ?, ?)",
(name, IPaddr, tm))
        self.__db.commit()
    except sqlite3.Error as e:
        print("Ошибка добавления статьи в БД "+str(e))
        return False

    return True

```

Методу **addDevice** передаются два аргумента: *name* и *IPaddr*. Затем, вычисляется текущее время добавления записи (в секундах) и выполняется запрос на добавление переданных данных. После этого обязательно нужно вызвать метод **commit** для *физического сохранения изменений* в БД. Также этот метод использует два вспомогательных модуля для вычисления текущего времени:

```

import time
import math

```

Чтобы воспользоваться этим классом в файле **main.py**, его нужно импортировать:

```

from FDataBase import FDataBase

```

6) В программе **main.py** импортировать FDataBase и добавить обработчик для адреса `/add_device`:

```

@app.route("/add_device", methods=["POST", "GET"])
def addDevice():
    db = get_db()
    dbase = FDataBase(db)

    if request.method == "POST":
        if len(request.form['name']) > 4 and len(request.form['IPaddr'])
> 6:
            res=dbase.addDevice(request.form['name'],request.form['IPaddr'
])

            if not res:
                flash('Ошибка добавления записи', category = 'error')
            else:
                flash('Запись добавлена успешно', category='success')
        else:
            flash('Ошибка добавления записи ', category='error')

    return render_template('add_device.html', title="Добавление
устройства")

```

Вначале идет подключение к БД, и после этого проверка: если были переданы данные от формы методом POST, то нужно осуществить добавление записи в таблицу **device**. Для этого вначале проверяется наличие данных в полях *name* и *IPaddr* и, если все нормально, то вызывается метод **addDevice** класса

**FD DataBase.** Кроме того, формируются мгновенные сообщения об успешности или ошибке при добавлении **устройства**. В конце возвращается шаблон **'add\_device.html'**, который выглядит так:

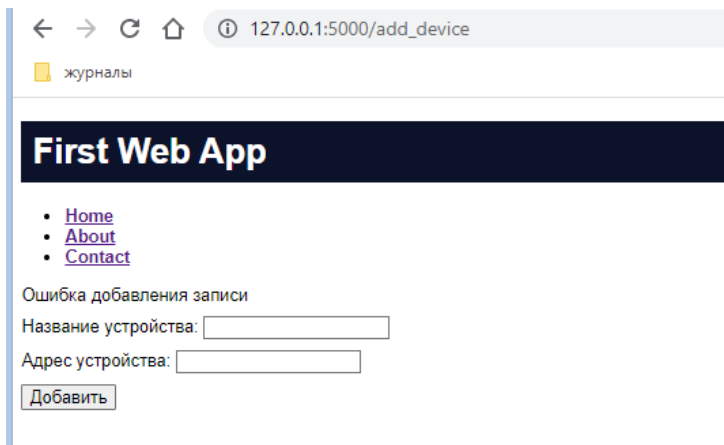
```
{% extends 'template.html' %}

{% block content %}
{{ super() }}
{% for cat, msg in get_flashed_messages(True) %}
<div class="flash {{cat}}">{{msg}}</div>
{% endfor %}

<form action="{{url_for('addDevice')}}" method="post" class="form-
contact">
<p><label>Название устройства: </label> <input type="text" name="name"
value="" required />
<p><label>Адрес устройства: </label> <input type="text" name="IPaddr"
value="" required />

<p><input type="submit" value="Добавить" />
</form>
{% endblock %}
```

Результат: страница добавления устройство выглядит следующим образом:



При нажатии на кнопку «Добавить» данные будут переданы обработчику `/add_device` и при успешной проверке принятых значений, запись будет добавлена в таблицу **device**.

### **Упражнение 17. Отображение записей таблицы device**

После того, как **устройства** добавлены в БД, их можно отобразить.

1) Для начала добавить обработчик в **main.py** для следующего URL-адреса:

```
@app.route("/device/<int:id_device>")
```



```
def showDevice(id_device):
    db = get_db()
    dbase = FDataBase(db)
    name, device = dbase.getDevice(id_device)
    if not title:
        abort(404)

    return render_template('device.html', name=name, device = device)
```

С его помощью будут отображаться устройства с указанным **id\_device**. Вначале также устанавливается соединение с БД, затем, вызывается метод **getDevice** класса FDataBase, который возвращает заголовок статьи и ее текст, а, иначе, при ошибке считывания данных из таблицы **device**, формируется ответ сервера 404 – страница не найдена. В конце возвращается шаблон '**device.html**' с данными об *устройстве*.

## 2) Добавить шаблон '**device.html**':

```
{% extends 'template.html' %}

{% block content %}
{{ super() }}
{{ device }}
{% endblock %}
```

3) Добавить в класс FDataBase метод **getDevice**, имеющий следующую реализацию:

```
def getDevice(self, deviceId):
    try:
        self.__cur.execute(f"SELECT name, IPaddr FROM device WHERE id = {deviceId} LIMIT 1")
        res = self.__cur.fetchone()
        if res:
            return res
    except sqlite3.Error as e:
        print("Ошибка получения статьи из БД "+str(e))

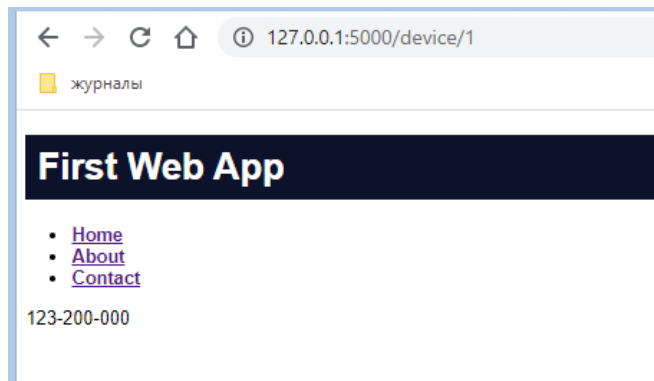
    return (False, False)
```

В методе **getDevice** сначала выбираются поля **name** и **IPaddr** для устройства, у которой **id** равен **deviceId**. Если метод **fetchone** возвращает не *None*, то есть, запись была найдена в БД, то возвращается кортеж из *названия и адреса устройства*. Иначе, возвращается кортеж из значений *False*.

## 4) Перейти в браузер и набрать запрос:

http://127.0.0.1:5000/**device**/1

Будет произведена попытка отобразить устройство с id равным 1.



**Упражнение 18. Отображение списка статей.** Отобразить список устройств на главной странице сайта.

1) В обработчике главной страницы `index()` нужно записать следующий код:

```
@app.route("/")
def index():
    db = get_db()
    dbase = FDataBase(db)

    return render_template('index.html', devices=dbase.getDevicesAnonce())
```

То есть, в функции `index` добавили подключение к БД, в `render_template` добавили один параметр **devices**, который будет ссылаться на список кортежей *устройств*.

2) В класс `FDataBase` добавить метод `getDevicesAnonce`:

```
def getDevicesAnonce(self):
    try:
        self.__cur.execute(f"SELECT id, name, IPaddr FROM device ORDER BY time DESC")
        res = self.__cur.fetchall()
        if res: return res
    except sqlite3.Error as e:
        print("Ошибка получения статьи из БД "+str(e))

    return []
```

В методе `getDevicesAnonce` запрашиваются все записи из таблицы **device** и сортируются по новизне: сначала самые свежие, затем, более поздние. После этого выбираются все записи с помощью метода **fetchall** и при успешности этой операции, возвращается список. Иначе, возвращается пустой список.

3) Изменить шаблон главной страницы **index.html** для отображения этого списка следующим образом:

```
{% extends 'template.html' %}

{% block content %}
{{ super() }}


---



## Список устройств




{% for p in devices %}
- <a href="{ { url_for('showDevice', id_ device=p.id) }}">
{{p.name}} </a> {{ p.IPAddr[:50]}} </p>

{% endfor %}


{% endblock %}
```

В этом шаблоне в блоке **for** перебирается список **devices** и формируются теги **li** с названием устройства и его анонсом.

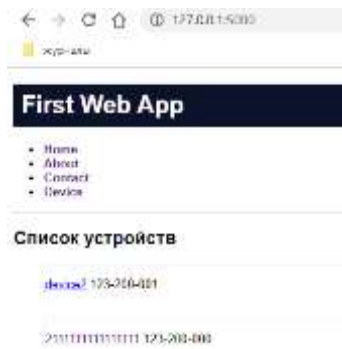
4) Для оформления страницы, добавим в файл **styles.css** следующие стили:

```
ul.list-devices {
    list-style: none;
    margin: 0;
    padding 0;
    max-width: 600px;
}
ul.list-devices li {
    margin: 20px 0 0 0;
    border: 1px solid #eee;
}
ul.list-devices.title {
    margin: 0;
    padding: 5px;
    background: #eee;
}
ul.list-devices.announce {
    margin: 0;
    padding: 10px 5px 10px 5px;
}
```

5) Добавить в *template.html* ссылку на добавление устройства:

```
<li><a href="{ { url_for('addDevice') }}">Device</a></li>
```

Результат:



Таким образом, рассмотрели простые операции с БД: запись данных в таблицы БД и их считывание.