

Проект WebServer. Обработка форм

План урока

- 1 Обработка форм на Flask
- 2 Шаблоны
- 3 Условия в шаблонах
- 4 Циклы в шаблонах
- 5 Наследование шаблонов
- 6 Знакомство с Flask-WTF

Аннотация

На этом занятии мы поработаем с шаблонами на Flask, а также рассмотрим несколько вариантов создания и обработки форм.

1. Обработка форм на Flask

На прошлом занятии мы немного коснулись того, как пользователь может взаимодействовать с нашим приложением, отправляя ему данные через параметры в декораторе **app.route**. Но, как вы могли заметить, такое взаимодействие не очень удобно, так как в общем случае параметров может быть много, а предлагать пользователю заносить их в адресную строку руками — не лучшая идея. К тому же, HTML-разметка позволяет создавать формы для ввода данных, с которыми Flask, конечно, умеет работать. Давайте вспомним, какие типы элементов ввода поддерживает HTML.

Вообще говоря, разных типов полей ввода довольно много, и периодически в новые версии языка разметки добавляются дополнительные. Вот некоторые из таких типов:

- `button` — кнопка;
- `checkbox` — множественный выбор;
- `color` — поле выбора цвета;
- `date`, `datetime`, `datetime-local`, `month`, `time`, `week` — ввод даты и времени;
- `email` — поле для ввода адреса электронной почты;
- `file` — поле для выбора файла;
- `number` — поле для ввода числовой информации;
- `password` — поле для ввода пароля;
- `radio` — выбор одного из нескольких вариантов;
- `range` — ползунок (как в музыкальном или видео-плеере);
- `submit` — кнопка для отправки формы;
- `tel` — поле для ввода телефона;
- `text` — поле для ввода текста;
- `url` — поле для ввода адреса в интернете.

Давайте сделаем форму с несколькими самыми распространёнными типами полей ввода, для их стилизации используем Bootstrap. Чтобы увидеть, в каком виде информация из этих полей придёт на сервер нашего веб-приложения, напомним такой код (предварительно импортировав **request** из Flask):

```
@app.route('/form_sample', methods=['POST', 'GET'])
def form_sample():
    if request.method == 'GET':
        return '''<!doctype html>
                <html lang="en">
                <head>
```

```
<meta charset="utf-8">
<meta name="viewport"
content="width=device-width, initial-scale=1, shrink-
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0
integrity="sha384-Gn5384xqQ1aowXA+058RXPxPg6fy4IWvTNh
crossorigin="anonymous">
<title>Пример формы</title>
</head>
<body>
<h1>Форма для регистрации в суперсекретной системе</h1>
<form method="post">
  <input type="email" class="form-control" id="email">
  <input type="password" class="form-control" id="password">
  <div class="form-group">
    <label for="classSelect">В каком вы классе</label>
    <select class="form-control" id="classSelect">
      <option>7</option>
      <option>8</option>
      <option>9</option>
      <option>10</option>
      <option>11</option>
    </select>
  </div>
  <div class="form-group">
    <label for="about">Немного о себе</label>
    <textarea class="form-control" id="about" rows="3">
  </div>
  <div class="form-group">
    <label for="photo">Приложите фотографию</label>
    <input type="file" class="form-control-file">
  </div>
  <div class="form-group">
    <label for="form-check">Укажите пол</label>
    <div class="form-check">
      <input class="form-check-input" type="radio" checked="">
      <label class="form-check-label" for="male">
        Мужской
      </label>
    </div>
    <div class="form-check">
      <input class="form-check-input" type="radio">
      <label class="form-check-label" for="female">
        Женский
      </label>
    </div>
  </div>
</div>
```

```

        <div class="form-group form-check">
            <input type="checkbox" class="form-check-input" />
            <label class="form-check-label" for="acceptRu">
        </div>
        <button type="submit" class="btn btn-primary">Зарегистрироваться
    </form>
</body>
</html>'''

elif request.method == 'POST':
    print(request.form['email'])
    print(request.form['password'])
    print(request.form['class'])
    print(request.form['file'])
    print(request.form['about'])
    print(request.form['accept'])
    print(request.form['sex'])
    return "Форма отправлена"

```

Форма для регистрации в суперсекретной системе

user1

Введите пароль

В каком вы классе

7

Немного о себе

Приложите фотографию

Выберите файл

Файл не выбран

Укажите пол

☒ Мужской

☐ Женский

☐ Готов быть добровольцем

Записаться

Мы дополнили наш декоратор `app.route` новым параметром — списком методов протокола HTTP, с которыми он работает. Всего методов довольно много, но мы будем говорить только о пяти из них:

1. GET — запрашивает данные, не меняя состояния сервера («прочитать»).
2. POST — отправляет данные на сервер («отправить»).
3. PUT — заменяет все текущие данные сервера данными запроса («заменить»).
4. DELETE — удаляет указанные данные («удалить»).
5. PATCH — используется для частичного изменения данных («изменить»).

Таким образом, наша функция **form_sample** работает с двумя методами. Если мы хотим получить данные с сервера, то тогда отработывает ветка условия, в которой мы отправляем пользователю форму для заполнения. Когда пользователь заполнил форму и нажал на кнопку «Записаться», данные формы заворачиваются в специальный аналог словаря в сущности **request**, которая хранит всю информацию о пользовательском запросе.

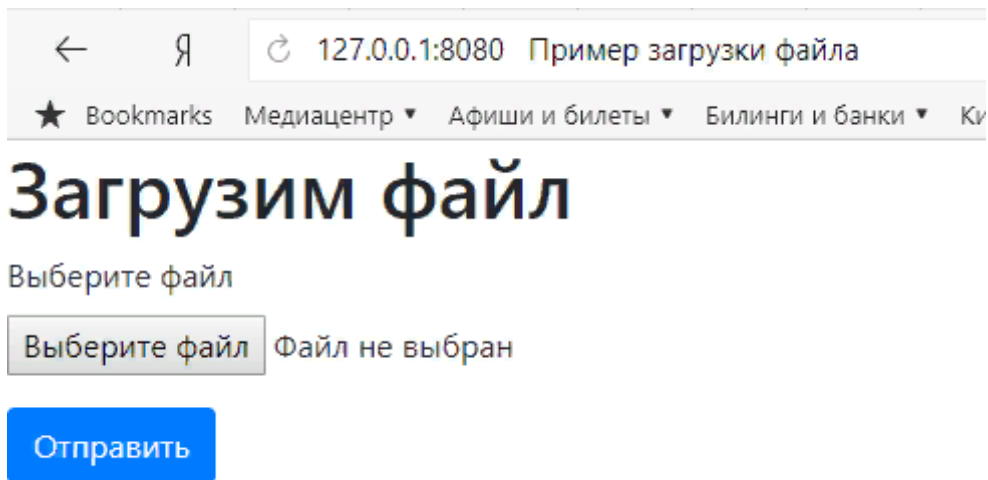
Обратите внимание: поскольку данные формы заворачиваются в аналог словаря, может случиться ситуация, когда по ключу мы не сможем найти значение (например, если пользователь не отметит чекбокс), и тогда будет выброшено исключение. Чтобы этого избежать, лучше обращаться к данным не напрямую по ключу, а с помощью метода **get**. Таким образом, более корректная обработка значения чекбокса будет выглядеть так:

```
request.form.get('accept')
```

Как вы могли заметить, у нас произошла небольшая странность с приложенным файлом: мы достали только его название, а не содержимое. Это произошло потому, что содержимое файла хранится в другом месте. Давайте напишем ещё один пример:

```
@app.route('/sample_file_upload', methods=['POST', 'GET'])
def sample_file_upload():
    if request.method == 'GET':
        return '''<!doctype html>
            <html lang="en">
            <head>
                <meta charset="utf-8">
                <meta name="viewport"
                    content="width=device-width, initial-scale=1, shrink-
                <link rel="stylesheet"
                    href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0
                    integrity="sha384-Gn5384xqQ1aowXA+058RXPxPg6fy4IWvTNh
                    crossorigin="anonymous">
                <title>Пример загрузки файла</title>
            </head>
            <body>
                <h1>Загрузим файл</h1>
                <form method="post" enctype="multipart/form-data">
                    <div class="form-group">
                        <label for="photo">Выберите файл</label>
                        <input type="file" class="form-control-file"
                    </div>
                        <button type="submit" class="btn btn-primary">Отпр
                    </form>
                </body>
            </html>'''
    elif request.method == 'POST':
        f = request.files['file']
```

```
print(f.read())
return "Форма отправлена"
```



← Я ↻ 127.0.0.1:8080 Пример загрузки файла

★ Bookmarks Медиацентр ▾ Афиши и билеты ▾ Билинги и банки ▾ Ки

Загрузим файл

Выберите файл

Выберите файл Файл не выбран

Отправить

Обратите внимание: кроме доступа к файлу в **request.files** мы немного изменили саму разметку формы, добавив в неё параметр

```
enctype="multipart/form-data"
```

Иначе форма так и продолжит отправлять только имена файлов, а при попытке доступа к самому файлу мы получим ошибку.

2. Шаблоны

Делать разметку непосредственно в коде Python плохо в 99,99% случаев. Это сложно поддерживать, неудобно писать, и наверняка вы почувствовали дискомфорт, пока делали предыдущие примеры. А у нас были достаточно простые страницы и небольшое количество информации, которая менялась динамически.

Для того, чтобы сделать жизнь программистов лучше, во Flask есть прекрасный механизм создания HTML-шаблонов, который мы сейчас и рассмотрим.

Практически всегда отделение логики приложения от макетов веб-страниц — отличная идея. Таким образом достигается нормальная организация внутри команды и становится возможным разделение работ. Каждый занимается своим делом: веб-дизайнер делает красиво, а разработчик — чтобы работало. Шаблоны как раз помогают достичь этого разделения. Во Flask шаблоны записываются как отдельные файлы, хранящиеся в папке **templates**, которая

находится (по-умолчанию) в корневой папке приложения. Давайте её создадим. И добавим в эту папку файл с HTML-разметкой — index.html со следующим содержимым:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>{{ title }}</title>
</head>
<body>
  <h1>Привет, {{ username }}!</h1>
</body>
</html>
```

Для нас такая разметка не представляет ничего сложного. Интерес вызывают разве что непонятные параметры в двух фигурных скобках. Давайте посмотрим, как шаблон будет работать. Для этого импортируем **render_template** из flask и напишем для нашего приложения новый обработчик главной страницы:

```
@app.route('/')
@app.route('/index')
def index():
    user = "Ученик Яндекс.Лицея"
    return render_template('index.html', title='Домашняя страница',
                           username=user)
```

И никакой разметки в нашем python-файле — прекрасно, не правда ли? Операция, которая преобразует шаблон в HTML-страницу, называется **рендерингом**. Чтобы отобразить шаблон, нам пришлось импортировать функцию **render_template()**. Эта функция принимает имя файла шаблона и перечень аргументов шаблона и возвращает один и тот же шаблон, но при этом все заполнители

```
{{...}}
```

в нём заменяются фактическими значениями переданных аргументов. Этот процесс работает схоже с прекрасно знакомым нам методом **format** для строк в Python. Механизм шаблонов, встроенный во Flask, называется **Jinja2**.

Важно не забывать, что кроме непосредственно переданных параметров, внутри шаблона мы имеем доступ и к служебным объектам. Например, уже знакомый нам **request**, или **session**, о котором мы поговорим позже. Кроме простой подстановки параметров Jinja2 умеет делать ещё несколько полезных вещей. Речь о них пойдёт далее.

3. Условия в шаблонах

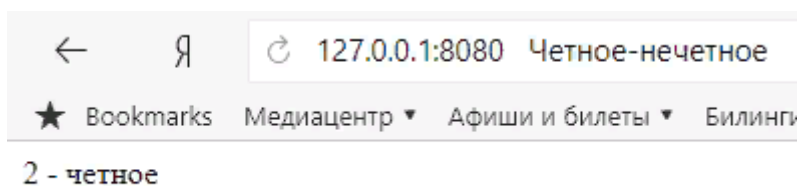
Шаблонизатор Flask поддерживает условные операторы, заданные внутри блоков `{% ...%}`. Давайте создадим шаблон `odd_even.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Четное-нечетное</title>
</head>
<body>
  {% if number % 2 == 0 %}
    <div>{{ number }} - четное</div>
  {% else %}
    <div>{{ number }} - нечетное</div>
  {% endif %}
</body>
</html>
```

И обработчик:

```
@app.route('/odd_even')
def odd_even():
    return render_template('odd_even.html', number=2)
```

Теперь в зависимости от того, что мы передали в шаблон, будет обрабатывать тот или иной блок.



Общий синтаксис условного оператора в шаблонах:

```
{% условие %}
    ветка 1
{% else %} (не обязательно)
    ветка 2
{% endif %}
```


Поддерживаются вложенные условия.

4. Циклы в шаблонах

Jinja2 поддерживает ещё и циклы `for`. Давайте создадим тестовый json-файл со списком новостей примерно следующего содержания:

```
{
  "news": [
    {
      "title": "Сегодня хорошая погода",
      "content": "Невероятно, сегодня хорошая погода"
    },
    {
      "title": "Завтра хорошая погода",
      "content": "С ума сойти, и завтра хорошая погода"
    },
    {
      "title": "Послезавтра дождь",
      "content": "Все вошло в норму"
    }
  ]
}
```

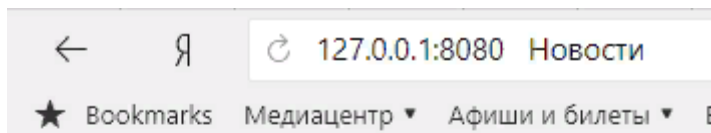
Напишем обработчик:

```
@app.route('/news')
def news():
    with open("news.json", "rt", encoding="utf8") as f:
        news_list = json.loads(f.read())
    print(news_list)
    return render_template('news.html', news=news_list)
```

И шаблон:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
<title>Новости</title>
</head>
<body>
    {% for item in news['news'] %}
        <h2>{{item["title"]}}</h2>
        <div>{{item["content"]}}</div>
    {% endfor %}
</body>
</html>
```



Сегодня хорошая погода

Невероятно, сегодня хорошая погода

Завтра хорошая погода

С ума сойти, и завтра хорошая погода

Послезавтра дождь

Все вошло в норму

Как вы можете заметить, общий синтаксис цикла выглядит так:

```
{% for переменная цикла in набор значений %}
    код
{% endfor %}
```

В качестве набора значений может выступать всё то, что и в обычном цикле на Python. Можно использовать и `range` (тогда можно смоделировать ситуации, когда значение в шаблон передавать не нужно, а цикл всё равно работает). Поддерживается вложенность.

5. Наследование шаблонов

В большинстве веб-приложений вверху или сбоку страницы есть главное меню, панели навигации с несколькими часто используемыми ссылками, внизу страницы зачастую располагается подвал (футер) с контактной информацией и т.д. Если веб-приложение содержит

несколько страниц, не составит большого труда добавить такую информацию во все шаблоны. Но по мере роста количества страниц это будет становиться всё труднее и труднее. Может возникнуть ситуация, когда при изменении номера телефона или добавлении нового пункта меню придётся изменить несколько сотен шаблонов. Кроме того, вы помните, что надо переиспользовать код, где это возможно, а писать одно и тоже несколько раз — плохая практика.

Jinja2 имеет функцию наследования шаблона, которая решает эту проблему. Мы можем разместить части макета страницы, которые являются общими для всех шаблонов, в базовом шаблоне, из которого выводятся все остальные шаблоны.

Давайте создадим базовый шаблон, который будет содержать небольшое верхнее меню, в файле `base.html`:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to=
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
  <title>{{title}}</title>
</head>
<body>
  <header>
    <nav class="navbar navbar-light bg-light">
      <a class="navbar-brand" href="#">Наше приложение</a>
    </nav>
  </header>
  <!-- Begin page content -->
  <main role="main" class="container">
    {% block content %}{% endblock %}
  </main>
</body>
</html>
```

В базовом шаблоне используется оператор управления блоком `{% block %}{% endblock %}`, чтобы определить место, куда будет вставляться содержимое дочерних шаблонов. Блокам присваивается уникальное имя (в нашем случае — `content`), на которое производные шаблоны могут ссылаться. Давайте изменим и наш `index.html`, который теперь будет выглядеть следующим образом:

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

{% block content %}
```

```
<h1>Привет, {{ username }}!</h1>
{% endblock %}
```

В **extends** мы указываем, какой шаблон мы хотим расширить, а в **block** — какой именно блок (их может быть несколько).

Обычно базовые шаблоны делают таким образом, чтобы они отвечали за общую структуру страницы. Новые страницы веб-приложения создают как производные шаблоны из одного и того же базового шаблона для избежания дублирования кода. Дочерний шаблон может расширять несколько блоков родительского шаблона, и при этом сам быть родительским для другого шаблона. Про все тонкости можно почитать в [официальной документации jinja2](#).

6. Знакомство с Flask-WTF

Микрофреймворк Flask силен в том числе и своей расширяемостью, которая позволяет значительно наращивать функциональность веб-приложения за счёт дополнительных модулей и небольших усилий. Мы рассмотрим модуль **flask-wtf** для обработки форм. У вас может возникнуть закономерный вопрос — зачем, ведь мы уже научились работать с формами? На самом деле, работа со сложными формами через разметку всё равно достаточно непростая задача, а **flask-wtf** помогает не только скрыть эту сложность, но и использует при этом объектно-ориентированный подход.

Чтобы установить **flask-wtf**, достаточно выполнить команду:

```
pip install flask-wtf
```

Прежде чем приступить к дальнейшей работе, давайте сделаем небольшую настройку нашего приложения и добавим следующую строку после создания переменной **app**:

```
app.config['SECRET_KEY'] = 'yandexlyceum_secret_key'
```

Эта настройка защитит наше приложение от [межсайтовой подделки запросов](#). Конечно, наш придуманный ключ довольно простой, но этот параметр необходим для корректной работы модуля. В принципе, защиту от CSRF-атаки можно отключить, но это не рекомендуется даже в учебных приложениях, как наше, чтобы при разработке своих больших проектов вы про это ненароком не забыли. Хорошая идея — хранить настройки приложения в отдельном файле конфигурации и считывать их при старте приложения.

Давайте создадим форму авторизации для входа в наше абстрактное приложение, которая будет содержать текстовое поле для ввода логина, поле для ввода пароля, чекбокс «Запомнить

меня» и кнопку отправки формы на сервер. Для начала создадим класс нашей будущей формы. Создадим файл loginform.py, в котором напишем следующий код:

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

class LoginForm(FlaskForm):
    username = StringField('Логин', validators=[DataRequired()])
    password = PasswordField('Пароль', validators=[DataRequired()])
    remember_me = BooleanField('Запомнить меня')
    submit = SubmitField('Войти')
```

Как видно из кода, мы импортируем класс **FlaskForm** из модуля **flask_wtf** — основной класс, от которого мы будем наследоваться при создании своей формы. Из модуля **wtforms** (**flask_wtf** — обёртка для этого модуля) мы импортируем типы полей, которые нам пригодятся для создания нашей формы: текстовое поле, поле ввода пароля, булево поле (из него получается чекбокс), и кнопку отправки данных.

Кроме этого, из модуля **wtforms.validators** импортируем проверку, которая скажет нам о том, введены ли данные в поле или нет. Создаём необходимые поля, на поля ввода логина и пароля вешаем проверку наличия там введённой информации.

Теперь перейдём к шаблону. Давайте создадим новый шаблон login.html, который будет расширять уже существующий шаблон:

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

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

```

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

```

Как видите, здесь для создания формы мы оперируем уже не разметкой, а атрибутами объекта `form`. **`form.hidden_tag`** — атрибут, который добавляет в форму токен для защиты от атаки, о которой мы говорили раньше. Из интересного в шаблоне есть ещё циклы, которые добавляют вывод ошибок заполнения полей. В нашем случае валидатор только один, но в общем случае их может быть несколько, поэтому ошибки лучше выводить именно таким образом. Теперь добавим обработчик:

```

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        return redirect('/success')
    return render_template('login.html', title='Авторизация', form=form)

```

Тут мы создаём нашу форму, и, если все поля прошли валидацию, после нажатия на кнопку отправки данных отправляем нашего пользователя на страницу удачного логина (не забудьте её сначала создать, а также импортировать **`redirect`** из модуля flask).

Наше приложение

Авторизация

Логин

Пароль

☐ Запомнить меня

Проверьте, как всё работает.

Обратите внимание, что проверка правильности значений полей в нашем случае ведётся на сервере. Поэтому важно поставить у нашей формы в шаблоне параметр **novalidate**, иначе Bootstrap будет проверять поля прямо в браузере и до сервера информация не дойдёт. Вообще значения полей можно проверять и там и там, иногда стоит даже дублировать проверки, чтобы злоумышленники не смогли изменить информацию, которую вы проверили на клиенте. Может возникнуть ситуация, когда данные формы были проверены на клиенте, а затем они были изменены уже после отправки формы (например, добавлен вредоносный код в текст поля). Если на сервере не проверить данные ещё раз, то ваше веб-приложение может утратить работоспособность и потерять все данные.

С загрузкой файлов в **flask-wtf** тоже есть свои небольшие особенности. Для загрузки файла достаточно создать поле типа **FileField**, а в обработке отправленной информации для получения содержимого файла добавить:

```
f = form.<название поля с файлом>.data
```

Мы с вами узнали, как отображать данные пользователю и как получать от него информацию. Осталось рассмотреть вопрос, как и где полученную информацию хранить и накапливать. Эти вопросы мы рассмотрим на следующем уроке.