Урок WEB. flask-wtf

Шаблоны. flask-wtf

1 Шаблоны
2 Условия в шаблонах
3 Циклы в шаблонах
4 Создание переменных в шаблоне
5 Наследование шаблонов
6 Знакомство с Flask-WTF

Аннотация

На этом уроке избавимся от необходимости создавать HTML-разметку непосредственно в коде, а также рассмотрим возможность создания форм с использованием объектного подхода с помощью библиотеки flask-wtf.

1. 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 ее 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 тот же шаблон, но при этом все блоки `{{...}}` в нем 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.

Запустите приложение и перейдите по ссылке http://127.0.0.1:8080/, из контекстного меню по правому клику в браузере выбирите «Показать исходный код» или «view page source». Посмотрите, какой HTML-код был сгенерирован на основе шаблона.

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.

Параметры в шаблон не обязательно передавать по отдельности, так как их может быть много и получать мы их можем внутри нашей потенциально длиной функции в разных местах. Можно собирать их в словарь параметров, а потом распаковывать его в вызове функции `render_template`:

```
@app.route('/')
@app.route('/index')
def index():
    param = {}
    param['username'] = "Ученик Яндекс.Лицея"
    param['title'] = 'Домашняя страница'
    return render_template('index.html', **param)
```

Если мы не заполним какой-то из параметров шаблона, то он по умолчанию будет считаться равным пустой строке.

Кроме простой подстановки параметров Jinja2 умеет делать еще несколько полезных вещей. Речь о них пойдет далее.

2. Условия в шаблонах

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

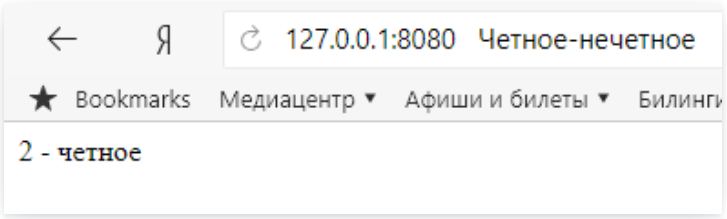
```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <title>Четное-нечетное</title>
</head>
<body>
 {% if number % 2 == 0 %}
        <div>{{ number }} - чётное</div>
 {% else %}
       <div>{{ number }} - нечётное</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 отрабатывать тот или иной блок.



Общий синтаксис условного оператора в шаблонах очень похож на тот, к которому мы привыкли в Python:

```
{% if условие_1 %}
       ветка 1
 {% elif условие_2 %} (не обязательно)
       ветка 1
{% else %} (не обязательно)
      ветка 2
{% endif %}
```

Поддерживаются вложенные условия.

3. Циклы в шаблонах

Jinja2 поддерживает еще 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"
        },
```

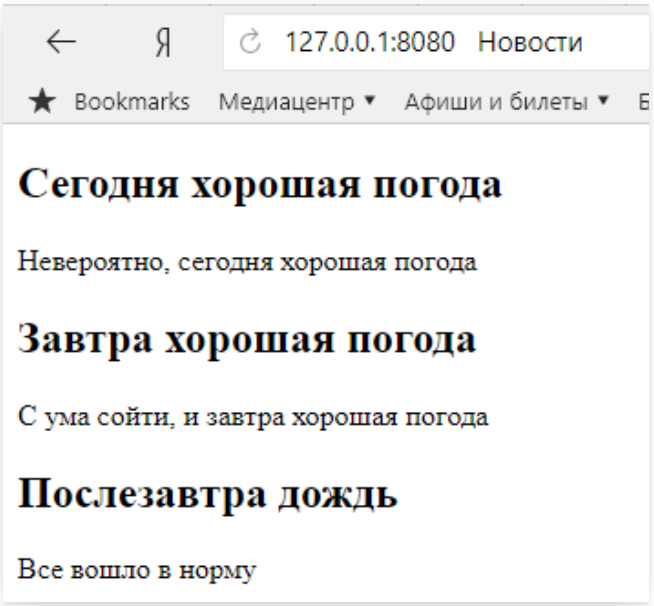
```
        {
            "title": "Послезавтра дождь",
            "content": "Все вошло в норму"
        }
    ]
}
```

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>
```



Как вы можете заметить, общий синтаксис цикла выглядит так:

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

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

Внутри цикла доступна переменная `loop` с рядом полезных атрибутов, например:

- `index` — индекс итерации с 1
- `index0` — индекс итерации с нуля
- `first` — True, если первая итерация, иначе False
- `last` — True, если последняя итерация, иначе False

4. Создание переменных в шаблоне

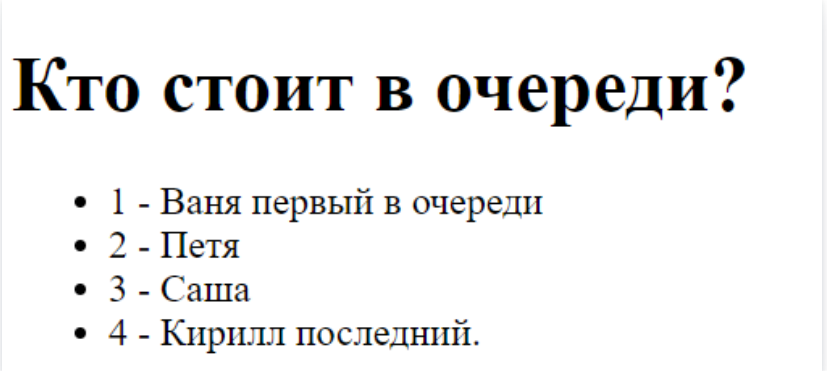
Несмотря на то что практически все значения для шаблона мы передаем из нашего кода на Python, периодически возникают ситуации, когда возникает необходимость создать переменную непосредственно в шаблоне, например, для хранения промежуточного результата вычислений. Это можно сделать с помощью ключевого слова `set`.

```
{% set a = 10 %}
```

Давайте рассмотрим пример для иллюстрации создания переменных в шаблоне, а заодно посмотрим на переменную `loop` внутри цикла:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ title }}</title>
</head>
<body>
<h1>Кто стоит в очереди?</h1>
{% set user_list = ['Ваня', 'Петя', 'Саша', 'Кирилл'] %}
<ul>
    {% for user in user_list %}
    <li>{{ loop.index }} - {{ user }} {% if loop.first %} первый в очереди {% elif loop.last %} по
    </li>
    {% endfor %}
</ul>
</body>
</html>
```

Добавим обработчик, запустим:



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 увеличения масштаба это будет становиться все труднее и труднее. Может возникнуть ситуация, когда при изменении номера телефона или добавлении нового пункта меню придется 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 общие для всех шаблонов части макета страницы 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-fit=no">
    <link rel="stylesheet"
          href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
          integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
          crossorigin="anonymous">
    <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**.

И еще.

Jinja поддерживает еще много интересного:

- Макросы — аналог функций из Python — куски кода шаблона, которые можно переиспользовать, обращаясь по имени
- Фильтры — некоторые функции, которые можно применять к данным при выводе, в большинстве случаев они дублируют функциональность методов строк, и позволяют немного модифицировать данные при отображении в шаблоне

Про это и много другое можно почитать в **документации**.

6. Знакомство с Flask-WTF

Микрофреймворк Flask силен, в том числе, и своей расширяемостью, которая позволяет значительно наращивать функциональность веб-приложения за счет дополнительных модулей с небольшими усилиями. Мы рассмотрим модуль flask-wtf для создания и 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 все 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 примера, мы импортируем класс `FlaskForm` из модуля flask_wtf — основной класс, от которого мы будем наследоваться при создании своей формы. Из модуля wtforms (flask_wtf — обертка для этого модуля) мы импортируем типы полей, которые нам пригодятся для создания нашей формы: текстовое поле, поле ввода пароля, булевое 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. Создаем необходимые поля, на поля ввода логина и пароля вешаем проверку наличия там введенной информации.

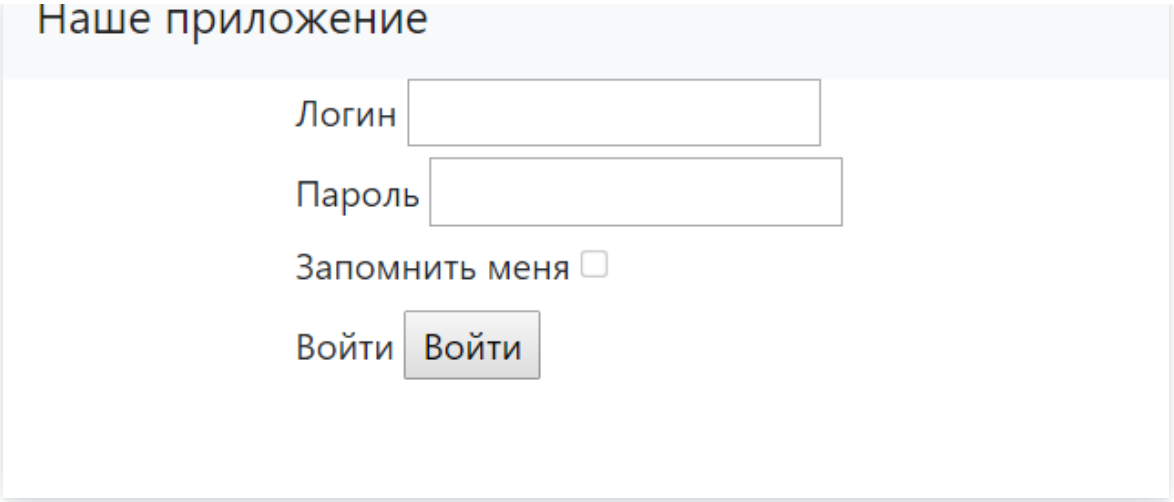
Прежде чем добраться до шаблона, давайте напишем обработчик, который будет оперировать пока не созданным шаблоном login.html (шаблон мы оставим на потом, чтобы посмотреть несколько вариантов):

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

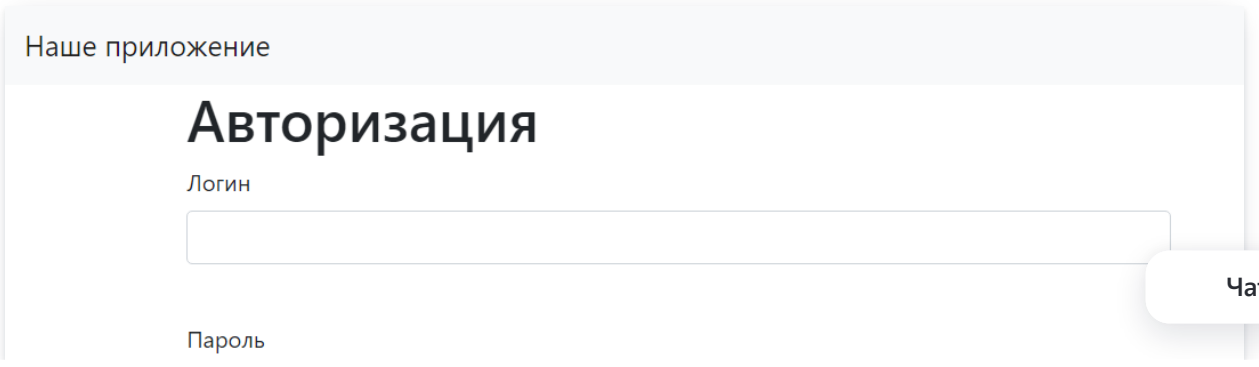
Теперь перейдем к шаблону. Давайте создадим новый шаблон login.html, который будет расширять уже существующий шаблон base.html. Есть несколько вариантов того, как мы можем отобразить поля нашей формы `LoginForm`. Если мы хотим просто создать поля и отобразить их с минимальной настройкой внешнего вида, тогда можно просто обойти их в цикле вот таким достаточно универсальным кодом:

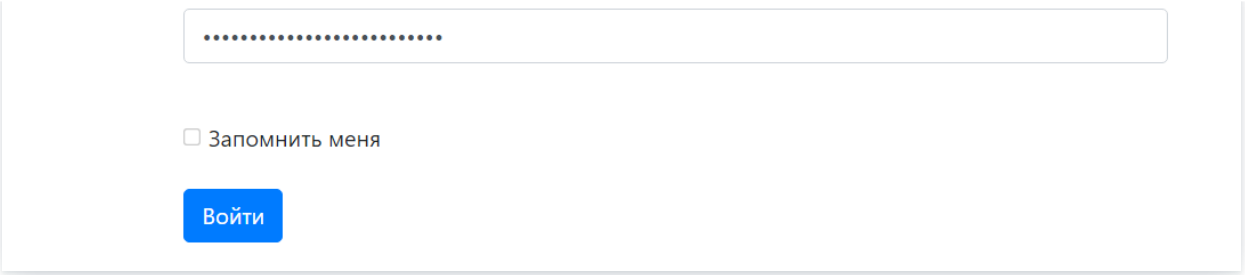
```
{% extends "base.html" %}

{% block content %}
    <form action="" method="post" novalidate>
        <div>
            {{ form.csrf_token }}
        </div>
        {% for field in form if field.name != 'csrf_token' %}
            <div>
                {{ field.label() }}
                {{ field() }}
                {% for error in field.errors %}
                    <div class="error">{{ error }}</div>
                {% endfor %}
            </div>
        {% endfor %}
    </form>
{% endblock %}
```



Если нам нужно больше контроля над отображением полей, можно обращаться в верстке к каждому из них по отдельности:

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

{% block content %}
<h1>Авторизация</h1>
<form action="" method="post" novalidate>
    {{ form.hidden_tag() }}
    <p>
        {{ form.username.label }}<br>
        {{ form.username(class="form-control") }}<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(class="form-control", type="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(type="submit", class="btn btn-primary") }}</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 еще 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.

Если мы хотим добавить к компонентам какой-нибудь стиль, класс или указать какой-то другой атрибут, то их можно просто передать как параметры к вызову нужной части формы.

Проверьте, как все работает.

Обратите внимание: проверка правильности значений полей в нашем случае ведется 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 дойдет. Вообще значения полей лучше проверять и там, и там:

- На клиенте для удобства пользователя (ему не надо ждать обновления страницы, чтобы получить ошибку)
- На сервере для безопасности, чтобы злоумышленники не смогли изменить информацию, которую вы проверили на клиенте. В конце концов, никто не запретит желающему сделать POST значений формы по нужному адресу не с HTML-страницы, а с помощью библиотеки `requests`

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 еще 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
```

В библиотеке flask-wtf есть поля для всех самых распространенных типов полей ввода, которые мы рассматривали на прошлом уроке, различные валидаторы.

Уверены, что для вас не является секретом факт, что большинство веб-приложений используют в качестве источника информации базы данных, с которыми мы познакомились, когда создавали приложение с графическим пользовательским интерфейсом с использованием компонентов PyQt. На следующих нескольких уроках мы продолжим с ними работать. И рассмотрим очень мощный инструмент, который упростит разработку именно в части общения с базами данных.