

# Лекция 2.

## Погружение во Flask



# Оглавление

<b>Погружение во Flask</b>	<b>3</b>
Экранирование пользовательских данных	3
<b>Генерация url адресов</b>	<b>4</b>
Генерация пути к статике	5
<b>Обработка запросов</b>	<b>7</b>
Обработка GET запросов	7
Обработка POST запросов	8
Замена route на get и post	9
Загрузка файлов через POST запрос	10
<b>Несколько полезных функций</b>	<b>11</b>
Обработка ошибок	11
Декоратор errorhandler	12
Функция abort	14
Некоторые коды ошибок	15
Перенаправления	17
Flash сообщения	18
Секретный ключ	19
Шаблон для flash сообщений	19
Категории flash сообщений	20
<b>Хранение данных</b>	<b>21</b>
Работа с cookie файлами в Flask	21
Создание ответа	22
Сессии	23
<b>Вывод</b>	<b>25</b>

## На этой лекции мы

1. Узнаем про экранирование пользовательских данных
2. Разберёмся с генерацией url адресов
3. Изучим обработку GET и POST запросов
4. Узнаем несколько полезных функций Flask
5. Разберёмся с cookie файлами и сессиями

# Краткая выжимка, о чём говорилось в предыдущей лекции

## На прошлой лекции мы:

1. Узнали о фреймворке Flask
2. Разобрались в установке и настройке Flask для первого запуска
3. Изучили работу функций представлений - view
4. Узнали о способах передачи html кода от сервера клиенту
5. Изучили работу с шаблонизатором Jinja
6. Разобрались с наследованием шаблонов

## Подробный текст лекции

### Погружение во Flask

На этой лекции мы продолжим знакомиться с фреймворком Flask и разберём ряд тонкостей и особенностей в работе с ним. Начнём с основ безопасности.

### Экранирование пользовательских данных

Начнём занятие с того, что не каждый пользователь будет делать то, что вы от него хотите. Например попросим пользователя передать путь до файла в адресной строке.

```
@app.route('/')
def index():
    return 'Введи путь к файлу в адресной строке'

@app.route('/<path:file>/')
def get_file(file):
    print(file)
    return f'Ваш файл находится в: {file}!'
```

А теперь вместо пути, передадим следующую строку:

[http://127.0.0.1:5000/<script>alert\("I am hacker"\)</script>/](http://127.0.0.1:5000/<script>alert()

На страницу будет выведен текст без пути. И одновременно сработает js скрипт с всплывающим сообщением о хакере. А ведь код может быть не таким безобидным как в примере.

Для повышения безопасности необходимо экранировать пользовательский ввод. Для этого используйте функцию `escape` из модуля `markupsafe`.

```
...
from markupsafe import escape

...
@app.route('/<path:file>/')
def get_file(file):
    return f'Ваш файл находится в: {escape(file)}!'
```

## Генерация url адресов

На прошлом занятии мы использовали относительные имена во всех примерах кода. Такой подход удобен пока приложения небольшое. Но лучше сразу привыкать к хорошему тону и при формировании адреса использовать функцию `url_for()`.

Рассмотрим поведение функции на следующем примере.


```
@app.route('/test_url_for/<int:num>/')
def test_url(num):
    text = f'В num лежит {num}<br>'
    text += f'Функция {url_for("test_url", num=42)} = <br>'
    text += f'Функция {url_for("test_url", num=42, data="new_data")} = <br>'
    text += f'Функция {url_for("test_url", num=42, data="new_data", pi=3.14515)} = <br>'
    return text
```

При переходе по адресу `test_url_for/7/` увидим следующий вывод:

```
В num лежит 7
Функция url_for("test_url", num=42) = '/test_url_for/42/'
Функция url_for("test_url", num=42, data="new_data") =
'/test_url_for/42/?data=new_data'
Функция url_for("test_url", num=42, data="new_data", pi=3.14515)
```

```
= '/test_url_for/42/?data=new_data&pi=3.14515'
```

Как видно из примера функция `url_for` принимает имя `view` функции в качестве первого аргумента и любое количество ключевых аргументов. Каждый ключ соответствует переменной в URL адресе. Отсутствующие в адресе переменные добавляются к адресу в качестве параметров запроса, т.е. после знака вопрос “?” как пары ключ-значение, разделённые символом `&`.

 **Внимание!** Обратите внимание, что первый параметр совпадает с названием функции-представления, а не с адресом внутри `route`. Таким образом изменение маршрутов автоматически изменит генерируемые `url` без лишних правок. Ведь имена `view` функций останутся прежними.

## Генерация пути к статике

Один из распространённых способов использования `url_for` является указание пути к файлам статике внутри шаблонов.

Рассмотрим следующее представление

```
@app.route('/about/')
def about():
    context = {
        'title': 'Обо мне',
        'name': 'Харитон',
    }
    return render_template('about.html', **context)
```

На прошлом занятии мы выводили примерно такой шаблон

```
<!doctype html>
<html lang="ru">
<head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="/static/css/bootstrap.min.css">
    <title>{{ title }}</title>
</head>
<body>
    <h1 class="text-monospace">Привет, меня зовут {{ name }}</h1>
```

```

<p class="text-body text-justify">Lorem ipsum dolor sit amet, consectetur adipisicing
elit. Ad cupiditate doloribus ducimus nam provident quo similique! Accusantium
aperiam fugit magnam quas reprehenderit sapiente temporibus voluptatum!</p>
<p class="alert-dark">Все права защищены &copy;</p>


<script src="/static/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```


В качестве статики тут прописаны стили и скрипты bootstrap, а также изображение из каталога image. Исправим эти три строки шаблона используя url\_for

```
...
<link rel="stylesheet" href="{{ url_for('static', filename='css/bootstrap.min.css') }}">
...

...
<script src="{{ url_for('static', filename='js/bootstrap.bundle.min.js') }}"></script>
...
```

Чтобы сгенерировать URL-адреса для статических файлов, необходимо использовать специальное имя “static” в качестве первого параметра, а по ключу filename передать путь до файла внутри каталога static.

 **Внимание!** Не стоит создавать view функцию с именем static.


 **Важно!** Во время разработки приложения за раздачу статики отвечает Flask. При запуске рабочего проекта статику раздаёт веб-сервер, а не Flask. Для этого надо настроит сервер. Изменять шаблоны Flask не нужно, url\_for сгенерировала необходимые пути.

# Обработка запросов

Современные приложения должны уметь обрабатывать отправляемые от клиентов данные. Это может быть информация из адресной строки, данные формы или даже наборы байт — файлы разных типов.

## Обработка GET запросов

До этого момента мы работали только с GET запросами. Представления реагировали на url адреса и получали из них данные в виде переменных. Через адресную строку можно передавать только текстовые данные. У самой строки есть ограничение на длину. А данные передаются либо как часть адреса, либо как пара ключ-значение после знака вопрос.

 **Важно!** Обработка GET запросов является поведением по умолчанию для представлений.

```
@app.route('/get/')
def get():
    if level := request.args.get('level'):
        text = f'Похоже ты опытный игрок, раз имеешь уровень {level}<br>'
    else:
        text = 'Привет, новичок.<br>'
    return text + f'{request.args}'
```


В первую очередь мы импортировали request — глобальный объект Flask, который даёт доступ к локальной информации для каждого контекста запроса. Звучит сложно. Если проще, то request содержит данные, которую клиент передаёт на сторону сервера.

Дополнительные параметры собираются в словаре args объекта request. И раз перед нами словарь, можно получить значение обратившись к ключу через метод get().

Перейдём по адресу <http://127.0.0.1:5000/get/?name=alex&age=13&level=80> и увидим следующий вывод:

```
Похоже ты опытный игрок, раз имеешь уровень 80
```

```
ImmutableMultiDict([('name', 'alex'), ('age', '13'), ('level', '80')])
```

 **Важно!** Используйте метод `get` для поиска значения внутри `request.args`. Так вы избежите ошибок обращения к несуществующему ключу. Строку формирует пользователь, а он может не знать о обязательных ключах. Альтернатива — блок `try` с обработкой `KeyError`.

## Обработка POST запросов

POST запросы используются для отправки данных на сервер. Они отличаются от GET запросов тем, что данные, передаваемые в POST запросах, не видны в URL. Также POST запросы могут содержать большее количество данных, чем GET.

Для того, чтобы передать данные в POST запросе, обычно используют HTML форму. У формы нужно указать атрибут `method="post"` для правильной обработки сервером.

Ниже приведен пример HTML формы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <title>Форма для POST запроса</title>
</head>
<body>
<form action="/submit" method="post">
  <input type="text" name="name" placeholder="Имя">
  <input type="submit" value="Отправить">
</form>
</body>
</html>
```

В данном примере мы создаем HTML-форму с полем "name" и кнопкой "submit". При нажатии на кнопку страница отправляет POST-запрос на указанный URL, в данном случае `/submit`.

В Python-коде функция, ассоциированная с URL `/submit`, использует функцию `request.form.get()` для получения данных, переданных через форму.


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



```
app = Flask(__name__)

@app.route('/submit', methods=['GET', 'POST'])
def submit():
    if request.method == 'POST':
        name = request.form.get('name')
        return f'Hello {name}!'
    return render_template('form.html')
```

В декораторе передаём список ['GET', 'POST'] по ключу methods. Теперь view готова обрабатывать как get, так и post запросы. Внутри делаем проверку метода. Если была отправлена форма, функция request.form.get() извлекает данные, переданные через форму, и сохраняет их в переменную name. Последняя строка сработает в случае get запроса и выводит шаблон с формой для заполнения.

 **Важно!** Используйте метод get для поиска значения внутри request.form. Так вы избежите ошибок обращения к несуществующему ключу. Нет гарантий, что клиент отправит все ключи, которые разработчик передаёт в HTML форме. Альтернатива - блок try с обработкой KeyError.

GET и POST запросы нужны, чтобы отправлять данные на сервер. GET запросы используются, чтобы получать данные, а POST — чтобы отправлять.

## Замена route на get и post

Рассмотренный выше пример функции submit можно записать иначе.

```
@app.get('/submit')
def submit_get():
    return render_template('form.html')

@app.post('/submit')
def submit_post():
    name = request.form.get('name')
    return f'Hello {name}!'
```

Вместо одной функции, которая обрабатывает и get и post запросы были созданы две. В первой использован декоратор get и она отвечает за отрисовки формы. Вторая функция имеет декоратор post с тем же самым аргументом, что и у get. Внутри читаем данные формы без лишних проверок метода запроса.

## Загрузка файлов через POST запрос

Загрузка файлов на сервер является неотъемлемой частью многих веб-приложений. В Flask, загрузка файлов может быть выполнена с помощью модуля Flask и объекта request. Рассмотрим простейший пример такой загрузки. Шаблон формы для загрузки файлов

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Форма для загрузки файла</title>
</head>
<body>
    <h1>Загружаем новый файл на сервер</h1>
    <form method=post enctype=multipart/form-data>
        <input type=file name=file>
        <input type=submit value=Загрузить>
    </form>
</body>
</html>
```

Параметр `enctype=multipart/form-data` создаёт форму для загрузки данных. Первая строк формы создаёт кнопку для прикрепления файла с доступом по имени `file`. Вторая — отправляет файл на сервер.

Простейший код Flask для приёма файла будет следующим.

```
from pathlib import PurePath, Path

from flask import Flask, request, render_template
from werkzeug.utils import secure_filename

app = Flask(__name__)

@app.route('/upload', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        file = request.files.get('file')
        file_name = secure_filename(file.filename)
        file.save(PurePath.joinpath(Path.cwd(), 'uploads',
file_name))
        return f"Файл {file_name} загружен на сервер"
    return render_template('upload.html')
```

Представление `upload` в первую очередь делает проверку на метод запроса. Первоначальный `Get` запрос приведёт к отрисовки шаблона `upload.html`. Его мы рассмотрели выше.

Получив `post` с файлом, сохраняем его в переменной `file`. В это время присланный набор байт будет храниться в оперативной памяти или во временном каталоге, если файл очень большой. Чтобы избежать проблем с плохими именами используем функцию `secure_filename` из модуля `werkzeug.utils`.



**Внимание!** Функция может вернуть пустую строку, если имя исходного файла не подходит для данной ОС.

У полученного файла (переменная `file`) есть метод `save`. Передав в него путь, происходит действительное сохранение присланного файла.



**Важно!** По умолчанию Flask не ограничивает размер файла для загрузки и не контролирует что именно загружается. Это открывает потенциальную опасность для межсетевого скриптинга.

## Несколько полезных функций

В этой главе рассмотрим несколько полезных функций, которые сделают ваше приложение лучше.

### Обработка ошибок

Что будет, если пользователь перешёл на несуществующую страницу? Если ничего не предпринимать, получим следующий вывод:

#### Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

## Декоратор errorhandler

Flask предоставляет возможности для обработки ошибок и способен заменить стандартный текст на симпатичную страницу в стиле вашего сайта.

Обработка ошибок в Flask происходит с помощью декоратора `errorhandler()`. Этот декоратор позволяет определить функцию-обработчик ошибок, которая будет вызываться в случае возникновения ошибки в приложении.

Например, чтобы обработать ошибку 404 (страница не найдена), необходимо определить функцию, которая будет вызываться при возникновении этой ошибки:

```
import logging

from flask import Flask, render_template, request

app = Flask(__name__)
logger = logging.getLogger(__name__)

@app.route('/')
def index():
    return '<h1>Hello world!</h1>'

@app.errorhandler(404)
def page_not_found(e):
    logger.warning(e)
    context = {
        'title': 'Страница не найдена',
        'url': request.base_url,
    }
    return render_template('404.html', **context), 404
```

В этом примере мы определяем функцию `page_not_found()`, которая будет вызываться при ошибке 404. Функция возвращает шаблон HTML страницы 404 и код ошибки 404. Обратите внимание, что в переменную `e` попадает текст той самой ошибки о “Not Found...”. Её мы выводим в логи как предупреждение.

В качестве контекста пробрасываем в шаблон заголовок страницы и адрес, по которому пытался перейти пользователь. Свойство `base_url` у объекта `request` возвращает тот адрес, который видит пользователь в адресной строке браузера.

Что касается шаблона, возьмём базовый из прошлой лекции.

Шаблон `base.html`

```

<!doctype html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1, shrink-to-fit=no">
  <link rel="stylesheet" href="/static/css/bootstrap.min.css">
  <title>
    {% block title %}
    Мой сайт
    {% endblock %}
  </title>
</head>
<body>
<div class="container-fluid">
  <ul class="nav nav-pills justify-content-end
align-items-end">
    <li class="nav-item"><a href="/main/"
class="nav-link">Основная</a></li>
    <li class="nav-item"><a href="/data/"
class="nav-link">Данные</a></li>
  </ul>

  {% block content %}
  Страница не заполнена
  {% endblock %}

  <div class="row fixed-bottom modal-footer">
    <hr>
    <p>Все права защищены &copy; </p>
  </div>
</div>

<script src="/static/js/bootstrap.bundle.min.js"></script>
</body>
</html>

```

В этом случае шаблон для ошибки 404 может выглядеть например так:

```

{% extends 'base.html' %}

{% block title %}
{{ title }}
{% endblock %}

```

```
{% block content %}
<div class="row">
    <p class="col-12">Мы не нашли страницу: &laquo;{{ url
    }}&raquo;.<br>
    <a href="{{ url_for('index') }}">Попробуйте вернуться на
    главную</a>
    </p>
</div>
{% endblock %}
```

Обратите внимание, что адрес главной страницы указан не явно, а генерируется через `url_for`. Подобная практика должна использоваться во всех шаблонах проекта для удобства масштабирования.

## Функция `abort`

Функция `abort()` также используется для обработки ошибок в Flask. Она позволяет вызвать ошибку и передать ей код ошибки и сообщение для отображения пользователю.

Например, чтобы вызвать ошибку 404 с сообщением "Страница не найдена", необходимо использовать функцию `abort()`:

```
import logging

from flask import Flask, render_template, request, abort

from flask_lesson.db import get_blog

app = Flask(__name__)
logger = logging.getLogger(__name__)

@app.route('/')
def index():
    return '<h1>Hello world!</h1>'

@app.route('/blog/<int:id>')
def get_blog_by_id(id):
    ...
    # делаем запрос в БД для поиска статьи по id
```

```

result = get_blog(id)
if result is None:
    abort(404)

...
# возвращаем найденную в БД статью

@app.errorhandler(404)
def page_not_found(e):
    logger.warning(e)
    context = {
        'title': 'Страница не найдена',
        'url': request.base_url,
    }
    return render_template('404.html', **context), 404

```

В этом примере мы используем функцию `abort()` внутри `get_blog_by_id` для вызова ошибки 404 в случае отсутствия статьи в базе данных.



**Внимание!** Чтобы код внутри представления отработал без ошибок, написана следующая функция заглушка:

```

def get_blog(id):
    return None

```

## Некоторые коды ошибок


- 400: Неверный запрос
- 401: Не авторизован
- 403: Доступ запрещен
- 404: Страница не найдена
- 500: Внутренняя ошибка сервера

Иногда из-за ошибок в коде сервер может возвращать ошибку 500. В идеальном мире код предусматривает все возможные ситуации и не отдаёт ошибку 500. Но почему бы не подстелить соломки.

Удалим функции `get_blog` из примера выше. Теперь при попытке найти статью по `id` получаем сообщение на странице:

## Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

 **Важно!** Если вы запускаете сервер в режиме отладки, будет выведена трассировка ошибки, а не сообщение. Перезапустите сервер с параметром `debug=False`

В несколько строк напишем обработчик для вывода сообщения в стиле проекта.

```
...
@app.errorhandler(500)
def page_not_found(e):
    logger.error(e)
    context = {
        'title': 'Ошибка сервера',
        'url': request.base_url,
    }
    return render_template('500.html', **context), 500
...
```

По сути взяли за основу обработчик ошибки 404, но лог фиксирует не предупреждение, а ошибку. Плюс новый шаблон, и возврат кода 500 клиенту.

Шаблон 500 может выглядеть так:

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

{% block title %}
{{ title }}
{% endblock %}

{% block content %}
<div class="row alert alert-danger">
    <p class="col-12">На сервере произошла ошибка. Мы уже знаем и
занимаемся исправлениями.<br>
    <a href="{{ url }}">Обновите страницу через несколько
минут</a>
    </p>
</div>
{% endblock %}
```



## Перенаправления

Перенаправления в Framework Flask позволяют перенаправлять пользователя с одной страницы на другую. Это может быть полезно, например, для перенаправления пользователя после успешной отправки формы или для перенаправления пользователя на страницу авторизации при попытке доступа к защищенной странице без авторизации.

Для перенаправления в Flask используется функция `redirect()`. Она принимает URL-адрес, на который нужно перенаправить пользователя, и возвращает объект ответа, который перенаправляет пользователя на указанный адрес.

Например, чтобы перенаправить пользователя на главную страницу сайта, можно использовать следующий код:

```
from flask import Flask, redirect, url_for

app = Flask(__name__)

@app.route('/')
def index():
    return 'Добро пожаловать на главную страницу!'

@app.route('/redirect/')
def redirect_to_index():
    return redirect(url_for('index'))

...
```

В этом примере мы определяем два маршрута: `/` для главной страницы и `/redirect` для перенаправления на главную страницу. Функция `redirect_to_index()` использует функцию `redirect()` для перенаправления пользователя на главную страницу с помощью функции `url_for()`, которая возвращает URL-адрес для указанного маршрута.

Функция `redirect()` также может использоваться для перенаправления пользователя на внешний URL-адрес. Например:

```
...
@app.route('/external')
def external_redirect():
    return redirect('https://google.com')
...
```

В этом примере мы используем функцию `redirect()` для перенаправления пользователя на внешний URL-адрес `https://google.com`.

Кроме того, в Flask есть возможность использовать перенаправления с параметрами. Например, чтобы передать параметры в URL-адрес при перенаправлении, можно использовать следующий код:

```
...
@app.route('/hello/<name>')
def hello(name):
    return f'Привет, {name}!'

@app.route('/redirect/<name>')
def redirect_to_hello(name):
    return redirect(url_for('hello', name=name))
...
```

В этом примере мы определяем маршрут `'/hello/<name>'`, который принимает параметр `'name'`, и маршрут `'/redirect/<name>'`, который использует функцию `redirect()` для перенаправления пользователя на маршрут `'/hello/<name>'` с передачей параметра `'name'`. Функция `url_for()` возвращает URL-адрес для указанного маршрута с передачей параметров.

## Flash сообщения

Flash сообщения в Flask являются способом передачи информации между запросами. Это может быть полезно, например, для вывода сообщений об успешном выполнении операции или об ошибках ввода данных.

Для работы с flash сообщениями используется функция `flash()`. Она принимает сообщение и категорию, к которой это сообщение относится, и сохраняет его во временном хранилище.

Например, чтобы вывести сообщение об успешной отправке формы, можно использовать следующий код:

```
from flask import Flask, flash, redirect, render_template,
request, url_for

app = Flask(__name__)
app.secret_key =
b'5f214cacbd30c2ae4784b520f17912ae0d5d8c16ae98128e3f549546221265e
4'
```

```
@app.route('/form', methods=['GET', 'POST'])
def form():
    if request.method == 'POST':
        # Обработка данных формы
        flash('Форма успешно отправлена!', 'success')
        return redirect(url_for('form'))
    return render_template('form.html')
```

В этом примере мы определяем маршрут '/form' для отображения и обработки формы. Если метод запроса POST, то происходит обработка данных формы и выводится сообщение об успешной отправке с помощью функции flash() и категории 'success'. Затем происходит перенаправление на страницу с формой с помощью функции redirect().

## Секретный ключ

Небольшое отступление. Чтобы не получать ошибки вида при работе с сессией

```
RuntimeError: The session is unavailable because no secret key
was set. Set the secret_key on the application to something
unique and secret.
```

необходимо добавить в Flask приложение секретный ключ.

Простейший способ генерации такого ключа, выполнить следующие пару строк кода

```
>>> import secrets
>>> secrets.token_hex()
```

Сразу после создания приложения прописываем инициализацию ключа сгенерированным набором байт. Теперь данные в безопасности, можно продолжать развивать приложение.

## Шаблон для flash сообщений

Чтобы вывести flash сообщения в HTML шаблоне, можно использовать следующий код шаблона:

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

{% block title %}
{{ title }}
{% endblock %}
```

```

{% block content %}
<form action="/form" method="post">
    {% with messages = get_flashed_messages(with_categories=true)
    %}

        {% if messages %}
            {% for category, message in messages %}
                <div class="alert alert-{{ category }}">
                    {{ message }}
                </div>
            {% endfor %}
        {% endif %}
    {% endwith %}
    <input type="text" name="name" placeholder="Имя">
    <input type="submit" value="Отправить">
</form>
{% endblock %}

```

Этот код использует функцию `get_flashed_messages()` для получения всех flash сообщений с категориями (блок `with`). Далее проверяем передавались ли сообщения через flash. Если да, в цикле происходит получение категорий и сообщений, т.к. указан параметр `with_categories=true`. Далее их вывод в соответствующих блоках с применением стилей bootstrap.

## Категории flash сообщений

Категории сообщений в flash позволяют различать типы сообщений и выводить их по-разному. Категория по умолчанию `message`. Но вторым аргументом можно передавать и другие категории, например `warning`, `success` и другие. Например, чтобы вывести сообщение об ошибке ввода данных, можно использовать следующую модификацию функции:

```

@app.route('/form', methods=['GET', 'POST'])
def form():
    if request.method == 'POST':
        # Проверка данных формы
        if not request.form['name']:
            flash('Введите имя!', 'danger')
            return redirect(url_for('form'))
        # Обработка данных формы
        flash('Форма успешно отправлена!', 'success')
        return redirect(url_for('form'))

```

```
return render_template('form.html')
```

Проверяем данные формы на наличие имени. Если имя не указано, то выводится сообщение об ошибке с категорией `danger` и происходит перенаправление на страницу с формой. Сама форма будет работать без изменений.

Flash сообщения являются удобным способом передачи информации между запросами в Flask. Они позволяют выводить сообщения пользователю и упрощают обработку ошибок и успешных операций.

## Хранение данных

В финале лекции рассмотрим возможность сохранения данных между запросами.

### Работа с cookie файлами в Flask

Cookie файлы — это небольшие текстовые файлы, которые хранятся в браузере пользователя и используются для хранения информации о пользователе и его предпочтениях на сайте. В Flask, работа с cookie файлами очень проста и может быть выполнена с помощью самого фреймворка, без установки дополнительных модулей.

Для работы с cookie файлами, необходимо импортировать модуль Flask и объект `request`, который позволяет получить доступ к cookie файлам. Подобное мы проделывали несколько раз за лекцию. Разберем куки на примере

```
from flask import Flask, request, make_response
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

```
def index():
```

```
    # устанавливаем cookie
```

```
    response = make_response("Cookie установлен")
```

```
    response.set_cookie('username', 'admin')
```

```
    return response
```

```
@app.route('/getcookie/')
```

```
def get_cookies():
    # получаем значение cookie
    name = request.cookies.get('username')
    return f"Значение cookie: {name}"
```

Мы устанавливаем значение cookie файла с ключом "username" и значением "admin" в функции index(). Затем мы получаем значение cookie файла с ключом "username" в функции get\_cookies() и выводим его на экран.

## Создание ответа

Несколько слов о функции make\_response(). Во всех прошлых примерах мы возвращали из view функций обычный текст, текст форматированный как HTML, динамически сгенерированные страницы через render\_template и даже запросы переадресации благодаря функциям redirect и url\_for. Каждый раз Flask неявно формировал объект ответа - response. Если же мы хотим внести изменения в ответ, можно воспользоваться функцией make\_response.

Изменим прошлый пример. Для начала создадим шаблон main.html

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

{% block title %}
    {{ super() }} - {{ title }}
{% endblock %}

{% block content %}
    <div class="row">
        <h1 class="col-12 col-md-6 display-2">Привет, меня зовут
        {{ name }}</h1>
        
    </div>
{% endblock %}
```

Шаблон принимает заголовок и имя пользователя. Для отрисовки он расширяет базовый шаблон. Ничего нового и никаких упоминаний “печенек”.

А теперь модифицируем представление

```
from flask import Flask, request, make_response, render_template
...
@app.route('/')
def index():
```

```

    context = {
        'title': 'Главная',
        'name': 'Харитон'
    }
    response = make_response(render_template('main.html',
**context))
    response.headers['new_head'] = 'New value'
    response.set_cookie('username', context['name'])
    return response
...

```

Используя `render_template` пробрасываем контекст в шаблон, но не возвращаем его, а передаём результат в функцию `make_response`. Ответ сформирован, но мы можем внести в него изменения перед возвратом. В нашем примере добавили в заголовки пару ключ-значение и установили куки для имени пользователя.



**Важно!** Не путайте заголовки ответа и содержимое блок `head` в теле ответа.

## Сессии

Сессии в Flask являются способом сохранения данных между запросами. Это может быть полезно, например, для хранения информации о пользователе после авторизации или для сохранения состояния формы при перезагрузке страницы. Для работы с сессиями в Flask используется объект `session`. Он представляет собой словарь, который можно использовать для записи и чтения данных. По сути сессия — продвинутая версия `cookies` файлов.

Пример использования сессии для хранения имени пользователя. Начнём с шаблона

```

{% extends 'base.html' %}

{% block title %}
{{ title }}
{% endblock %}

{% block content %}
<form method="post">
    <input type="text" name="username" placeholder="Имя">
    <input type="submit" value="Отправить">
</form>

```

```
{% endblock %}
```

Простая форма запрашивает username и отправляет его на сервер. А теперь код Flask:

```
from flask import Flask, request, make_response, render_template, session, redirect, url_for

app = Flask(__name__)
app.secret_key =
'5f214cacbd30c2ae4784b520f17912ae0d5d8c16ae98128e3f549546221265e4
'

@app.route('/')
def index():
    if 'username' in session:
        return f'Привет, {session["username"]}'
    else:
        return redirect(url_for('login'))

@app.route('/login/', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form.get('username') or
'NoName'
        return redirect(url_for('index'))
    return render_template('username_form.html')

@app.route('/logout/')
def logout():
    session.pop('username', None)
    return redirect(url_for('index'))
```

Для работы с сессиями в Flask необходимо установить секретный ключ (secret\_key). Он используется для шифрования данных в сессии и должен быть уникальным и сложным. Подобное мы уже делали на лекции.

Далее мы определяем три маршрута:

- '/' для вывода имени пользователя
- '/login' для авторизации
- '/logout' для выхода.



При отправке формы на странице /login происходит запись имени пользователя в сессию. Если имя пользователя уже есть в сессии, то оно выводится на странице '/.

Сессии в Flask имеют несколько особенностей

- Срок действия сессии по умолчанию составляет 31 день, но его можно изменить с помощью параметра `app.permanent_session_lifetime`. Передаётся новое значение как объект `timedelta`.
- Данные в сессии хранятся на стороне сервера, поэтому они не могут быть изменены или прочитаны клиентом.
- При использовании сессий необходимо обеспечить безопасность приложения, чтобы злоумышленники не могли получить доступ к данным в сессии. Например, для защиты от атак перехвата сессии можно использовать HTTPS и установку параметра `app.session_cookie_secure=True`.

Чтобы удалить данные из сессии используем функцию `logout`. Внутри неё вызываем `session.pop()` с именем удаляемого ключа: `session.pop('username', None)`.

В нашем случае функция удаляет значение по ключу 'username' из сессии. Если ключа нет в сессии, то функция не вызывает ошибку.

Сессии Flask позволяют сохранять данные между запросами и обеспечивают безопасность приложения. Они могут быть использованы для хранения информации о пользователе, настройках приложения и других данных.

## Вывод

На этой лекции мы:

1. Узнали про экранирование пользовательских данных
2. Разобрались с генерацией url адресов
3. Изучили обработку GET и POST запросов
4. Узнали несколько полезных функций Flask
5. Разобрались с cookie файлами и сессиями

## **Домашнее задание**

Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.