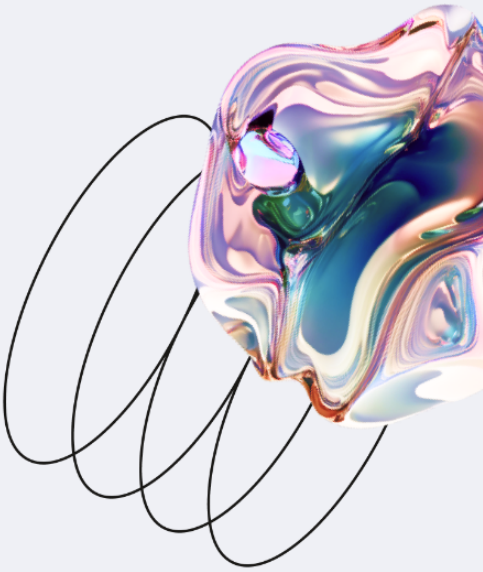
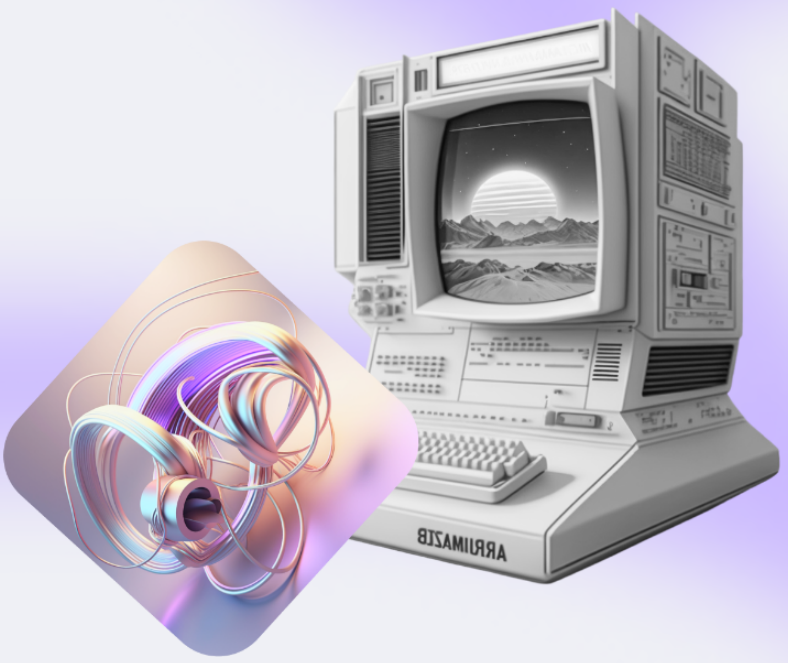
GeekBrains



Лекция 1.

Знакомство с Flask



Оглавление

На этой лекции мы

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

Что такое Flask

Flask — это микрофреймворк для Python, созданный в 2010 году разработчиком по имени Армин Ронахер. Пристава «микро» говорит о том, что Flask действительно маленький. У него в комплекте нет ни набора инструментов, ни библиотек, которыми славятся другие популярные фреймворки. Но он создан с потенциалом для расширения. Во фреймворке есть набор базовых возможностей, а расширения отвечают за все остальное. «Чистый» Flask не умеет подключаться к базе данных, проверять данные формы, загружать файлы и так далее. Для добавления этих функций нужно использовать расширения. Это помогает использовать только те из них, которые на самом деле нужны.
При этом Flask не такой жесткий в отношении того, как разработчик должен структурировать свою программу. Он отлично подходит для освоения веб-разработки.

Hello world! на Flask

Python помогают решить проблему управления зависимостями проектов. Если у вас есть несколько проектов на Python, то вероятность того, что вам придется работать с разными версиями библиотек или самого Python, очень высока. Но использование виртуальных сред позволяет создавать независимые группы библиотек для каждого проекта, что предотвращает конфликты между версиями и не дает одному проекту повлиять на другой. В Python уже есть модуль venv для создания виртуальных сред, который можно использовать как в разработке, так и в производстве.

Виртуальное окружение

Создаём виртуальное окружение в Linux или MacOS

```
mkdir project
cd project
```

```
python3 -m venv venv
```

Создаём виртуальное окружение в Windows

```
mkdir project
cd project
python -m venv .venv
```

Далее активируем виртуальное окружение, чтобы все дальнейшие действия выполнялись внутри него.

```
venv/bin/activate  # Linux/MacOS
venv\Scripts\activate  # Windows
venv\Scripts\activate.ps1  # Windows PowerShell
```

Установка Flask и обязательный компонентов

Для установки Flask выполняем следующую команду

```
pip install Flask
```

Помимо самого фреймворка будет установлено несколько обязательных зависимостей.

- Werkzeug — набор инструментов WSGI, стандартного интерфейса Python для развёртывания веб-приложений и взаимодействия между ними и различными серверами разработки. Отвечает за роутинг, обработку запросов и ответов, а также предоставляет такие возможности, как debugger и reloader.
- Jinja2 — движок шаблонов и одновременно современный язык шаблонов для Python, созданный по образцу шаблонов Django. Он быстр, широко используется и безопасен, благодаря дополнительной среде отрисовки шаблона в песочнице.
- Click — фреймворк для написания приложений командной строки. Он предоставляет консольную команду flask и позволяет добавлять пользовательские команды управления.
- MarkupSafe поставляется с Jinja. MarkupSafe исключает ненадежный ввод при рендеринге шаблонов, чтобы избежать атак путем внедрения нежелательного кода.
- itsDangerous - дополнение, которое подписывает данные для обеспечения их целостности. Он используется для защиты cookie файлов в сеансе Flask.

Создаём первое веб-приложение

Простейшее приложение на Flask займёт несколько строк кода в одном файле.

```
from flask import Flask

app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello World!'


if __name__ == '__main__':
    app.run()
```

В первой строке указываем импортировать из модуля flask класс Flask. Экземпляр этого класса будет нашим WSGI-приложением.
Далее создаем экземпляр этого класса и сохраняем его в переменную app. Мы передаем классу имя модуля или пакета. Flask не знает, где искать шаблоны, статические файлы и так далее, поэтому мы должны использовать переменную __name__.
Затем используем декоратор route() из переменной app.
Декоратор в Python — это функция, которая принимает другую функцию в качестве аргумента.
Декоратор route() принимает строку с URL-адресом. При переходе пользователя сайта по указанному адресу запускается декорированная функция.
Внутри функции-представления (hello world() в нашем примере) прописывается логика обработки пользовательского запроса. Функция обязательно возвращает (return) сообщение, которое мы хотим отобразить в браузере пользователя. При этом сообщение может быть строкового типа (str).
Наконец, мы используем run()-функцию для запуска локального сервера с нашим приложением. Условие __name__ == '__main__' означает, что сервер работает только в том случае, если скрипт выполняется непосредственно из Python-интерпретатора и не используется в качестве импортированного модуля.

Первый запуск

Запустить веб сервер можно несколькими способами. Например можно запустить файл проекта на выполнение из вашей IDE.
Другой способ — команда flask в терминале ОС. Например

```
flask --app lesson_1/project run
```

Благодаря установке click у нас работает команда flask в которую по ключу --app передаём название основного файла проекта и команду run на запуск.

```
 * Serving Flask app 'lesson_1/project'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Обратите внимание на предупреждение. Встроенный сервер подходит для разработки проекта, но его не рекомендуют использовать в продакшене.
Чтобы убедиться в работе, переходим по адресу http://1_27.0.0.1:5000 и видим в браузере Hello World!

Первая оптимизация

Внесём изменения в проект. В корневом каталоге создадим файл wsgi.py со следущим кодом.

```
from lesson_1.project import app

if __name__ == "__main__":
    app.run(debug=True)
```

Импортируем из файла проекта переменную приложения. Параметр debug=True включает режим отладки.
Теперь для запуска сервера из командной строки достаточно выполнить команду

```
flask run --debug
```

Файл с именем wsgi.py будет найден автоматически.

🔥 **Важно!** В рамках урока будут использоваться различные, зачастую не связанные между собой примеры кода. Поэтому в каждом файле будет использоваться своя конструкция app.run() для запуска именно этого файла в качестве Flask сервера.

Устройство view функций

Разберём подробнее как работаю функции представления.
Маршрут (или путь) используется во фреймворке Flask для привязки URL-адреса к функции представления. Эта функция отвечает на запрос. Во Flask декоратор route() используется, чтобы связать URL с функцией.

```
@app.route('/')
def index():
    return 'Привет, незнакомец!'
```

Код назначает функцию index() обработчиком корневого URL в приложении. Когда приложение будет получать запрос, где путь — /, вызывается функция index(), и на этом запрос завершается.
В следующем примере создано три маршрута в виде трёх отдельных view функций

```
@app.route('/')
def index():
    return 'Привет, незнакомец!'


@app.route('/Николай/')
def nike():
    return 'Привет, Николай!'


@app.route('/Иван/')
def ivan():
    return 'Привет, Ванечка!'
```

Теперь при переходе по адресу http://127.0.0.1:5000/Иван/ в браузере будет открываться новая страница с приветствием.

> 🔥 **Внимание!** Если в строке браузера указать адрес без слеша на конце, сервер автоматически обработает адрес со слешем. Важно не забывать ставить слеш в конце адреса, который передаётся в route().

Множественное декорирование

Одна функция-представление может быть декорирована несколькими декораторами.

```
@app.route('/Фёдор/')
@app.route('/Fedor/')
@app.route('/Федя/')
def fedor():
    return 'Привет, Феодор!'
```

Функция представления имеет три декоратора. При переходе по любому из этих адресов в браузере отобразится одна и та же строка «Привет, Феодор!».

Прописываем логику обработки URL

Создадим функцию представление, которая будет получать в URL-адресе имя и здороваться с указанием переданного имени.

```
@app.route('/')
@app.route('/<name>/')
def hello(name='незнакомец'):
    return 'Привет, ' + name + '!'
```

Функция будет отрабатывать корневой адрес и адреса, где передаётся любой текст между корневым слешем и замыкающим. При этом текст из браузера сохраняется в переменной <name>.
Далее функция hello() принимает на вход содержимое переменной name. Если в браузере ничего не ввести, будет подставлено значение по умолчанию — «незнакомец».

🔥 **Внимание!** Обратите внимание, что в route() переменная заключается в треугольные скобки, а в hello() — без скобок.

Функция возвращает динамически сгенерированную строку: «Привет» плюс переданное имя или слово «незнакомец» плюс восклицательный знак.

Типы переменных при передаче в функцию

В примере выше мы передали через name строковое значение. Это действие по умолчанию. Помимо этого можно передавать следующие данные:

- string — (по умолчанию) принимает текст без слеша
- int — принимает позитивные целые числа
- float — принимает позитивные числа с плавающей точкой
- path — как string, но принимает слеши
- uuid — принимает строки UUID

В примере ниже содержимое строки после file воспринимается как путь и попадает в переменную path независимо от количества слешей

```
@app.route('/file/<path:file>/')
def set_path(file):
    print(type(file))
    return f'Путь до файла "{file}"'
```

🔥 **Внимание!** Переменная file содержит строку типа str. Разница в типах именно в восприятии слешей как части содержимого строки

А в этом примере num ожидает число с плавающей запятой.

```
@app.route('/number/<float:num>/')
def set_number(num):
    print(type(num))
    return f'Передано число {num}'
```

Если вы попытаетесь передать данные другого типа, получим ошибку 404, страница не будет отработана.

Выводим HTML

Рассмотрим два варианта вывода HTML.

Многостраничный текст с тегами

Python легко может сохранить многостраничный документ в переменной, если заключить его в три двойные кавычки.

```
html = """
<h1>Привет, меня зовут Алексей</h1>
<p>Уже много лет я создаю сайты на Flask.<br/>Посмотрите на мой сайт.</p>
"""
```

Содержимое переменной можно вернуть, используя функцию представления. При этом браузер выведет текст с учётом тегов.

```
@app.route('/text/')
def text():
    return html
```

Как вы видите, html теги не выводятся в браузере как текст, а преобразуются в теги.

При желании можно сделать страницу динамической. В примере ниже каждая строчка стихотворения хранится как элемент списка list. Для примера в первой лекции этого достаточно. Но вы должны понимать, что аналогичным образом можно использовать данные из БД, внешних источников и т.п.

```
@app.route('/poems/')
def poems():
    poem = ['Вот не думал, не гадал,',
            'Программистом взял и стал.',
            'Хитрый знает он язык,',
            'Он к другому не привык.',
            ]
      txt = '<h1>Стихотворение</h1>\n<p>' + '<br/>'.join(poem) +
'</p>'
    return txt
```

При желании можно прописать любую логику внутри функции, в зависимости от задач программиста и того, какую информацию необходимо вывести на странице сайта.

Рендеринг HTML файла

Попробуем вывести файл index.html, используя локальный сервер Flask.

```
<!doctype html>
<html lang="ru">
<head>
   <meta charset="utf-8">
   <link rel="stylesheet" href="/static/css/bootstrap.min.css">
   <title>Главная</title>
</head>
<body>
   <h1 class="text-monospace">Привет, меня зовут Алексей</h1>
   <img src="/static/image/foto.jpg" alt="Моё фото" width="300">
    <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>
</body>
</html>
```

Начнём с того, что импортируем функцию отрисовки шаблонов. render_template() принимает в качестве первого аргумента название html-файла, который необходимо вывести в браузер.

```
from flask import render_template
```

Добавим функцию рендеринга в функцию представления и укажем ей на файл index.html. Общий код будет выглядеть так:

```
from flask import Flask
from flask import render_template

app = Flask(__name__)


@app.route('/index/')
def html_index():
   return render_template('index.html')
```

После перехода по локальному адресу получим сообщение об ошибке:

```
TemplateNotFound
jinja2.exceptions.TemplateNotFound: index.html
```

Функция render_template() ищет файл index.html в папке templates. Необходимо перенести его в нужную папку. Другие html-файлы также необходимо складывать в указанную папку.
После перезагрузки сервер выводит страницу в браузер.

> 🔥 **Внимание!** Если изображения или стили отсутствуют, необходимо переместить их в соответствующие каталоги: стили в static/css, а изображения — в static/image. В самом html проверить путь к файлам:

```
<link rel="stylesheet" href="/static/css/style.css">
<img src="/static/image/foto.png" alt="Моё фото" width="300">
```

После очередной перезагрузки сервера мы получим полноценную html-страницу.

Шаблонизатор Jinja

В примере выше мы смогли отрисовать HTML страницу силами Flask. Но если быть более точным, нам помог шаблонизатор Jinja. При этом сам html файл представляет статичную страницу сайта. Это удобно для быстрого получения информации клиентом от сервера. Но крайне неудобно для создания чего-то большего, чем одна страница. Благодаря Jinja статические html-страницы превращаются в шаблоны для формирования динамических сайтов.

Пробрасываем контекст из представления в шаблон

Функция render_template после имени шаблона может принимать неограниченное число именованных аргументов и пробрасывать их в шаблон. Шаблон позволяет вывести значение по имени, заключив его в двойные фигурные скобки {{ }}. Такие скобки — аналог функции print() в Python.
Изменим строку вывода в функции, добавив аргумент name со значением «Харитон»:

```
return render_template('index.html', name='Харитон')
```

В шаблоне заменим имя владельца на вывод переменной из шаблона.

```
<h1 class="text-monospace">Привет, меня зовут {{ name }}</h1>
```

Jinja не ограничивает пользователя в количестве переменных, которые необходимо передать в шаблон. Но для сохранения читаемости кода рекомендуется сохранять все переменные в словарь и пробрасывать в шаблон его распакованный вид.
Распаковка словаря — передача его содержимого как отдельных значений. В Python для распаковки словаря необходимо добавить две звёздочки ** перед именем словаря.

Модифицируем нашу функцию представления для передачи не только имени, но и заголовка страницы.

```
@app.route('/index/')
def index():
    context = {
        'title': 'Личный блог',
        'name': 'Харитон',
    }
    return render_template('index.html', **context)
```

Теперь в шаблон проброшены переменные name и title и можно заменить содержимое шаблона внутри тега <title> на переменную.

```
<title>{{ title }}</title>
```

🔥 **Внимание!** До и после двойных фигурных скобок рекомендуется оставлять пробел. Это не только облегчает чтение, но и в некоторых случаях заставляет код работать верно.

Условный оператор в шаблоне

Ветвления в Jinja имеют схожую с Python логику, но немного отличаются по синтаксису. Оператор if и логическое условие заключаются в скобки вида {% %}. В отличие от Python обязательным является закрывающий условие код вида {% endif %}.

```
{% if user %}
    <p>Вы вошли под именем {{ user }}</p>
```

```
{% endif %}
```

Если в шаблон передали переменную user, будет выведен абзац текста. В противном случае код между открывающим и закрывающим операторами будет проигнорирован, не появится на стороне клиента.
Как и в Python, шаблоны поддерживают сложные условия благодаря конструкциям `{% elif %}` и `{% else %}`
Например мы можем выбирать окончание предложения в зависимости от переданного числа.

```
<p>К прочтению предлагается {{ number }}
    {% if number == 1 %}
        пост
    {% elif 2 <= number <= 4 %}
        поста
    {% else %}
        постов
    {% endif %}
</p>
```

> 🔥 **Важно!** Не забудьте добавить ключ number в словарь context для пробрасывания переменной из функции в шаблон.

Вывод в цикле

Аналогично Python, можно использовать цикл for внутри шаблона для вывода элементов последовательности. Из примера ниже понятно, что цикл заключается в специальные скобки `{% %}`, а конец цикла обязательно заканчивается блоком `{% endfor %}`

```
{% for item in item_list %}
    {{ item }}
{% endfor %}
```

Изменим представление poems(), которое создали ранее на лекции. Сформируем аналогичный вывод стихотворения силами шаблонизатора Jinja.
Помещаем список со строками стихотворения в словарь и пробросим его в шаблон.

```
@app.route('/poems/')
def poems():
    context = {'poem': ['Вот не думал, не гадал,',
                        'Программистом взял и стал.',
                        'Хитрый знает он язык,',
                        'Он к другому не привык.',
                       ]}
            #    txt    =    """<h1>Стихотворение</h1>\n<p>"""    +
'<br/>'.join(poem) + '</p>'
    return render_template('poems.html', **context)
```

В шаблоне poems.html создадим цикл для форматированного вывода

```
...
<body>
<h1 class="text-monospace text-center">Стихотворение</h1>
<p class="row">
   {% for line in poem %}
            <span  class="text-black-50  col-12  col-md-6">{{  line
}}</span><br/>
   {% endfor %}
</p>
</body>
...
```

Как и в Python, условия и циклы можно использовать совместно, помещая одно в другое в зависимости от задач программиста.

Вывод сложных структур в цикле

Иногда необходимо вывести информацию о нескольких однотипных объектах с набором свойств. Например, информацию о пользователях из базы данных. Или если упростить задачу, список словарей с одинаковыми ключами. Для опытных программистов очевидно, что оба вывода идентичны. Рассмотрим список словарей.

```
@app.route('/users/')
def users():
    _users = [{'name': 'Никанор',
               'mail': 'nik@mail.ru',
               'phone': '+7-987-654-32-10',
               },
```

```
                {'name': 'Феофан',
                 'mail': 'feo@mail.ru',
                 'phone': '+7-987-444-33-22',
                 },
                {'name': 'Оверран',
                 'mail': 'forest@mail.ru',
                 'phone': '+7-903-333-33-33',
                 }, ]
    context = {'users': _users}
    return render_template('users.html', **context)
```

При выводе в шаблоне используем точечную нотацию для доступа к элементам списка словарей.

```
...
<body>
<div class="row">
        <h1   class="col-12   text-monospace   text-center">Список
пользователей из БД</h1>
   {% for user in users %}
       <div class="col-12 col-md-6 col-lg-4">
           <h2>{{ user.name }}</h2>
           <p>{{ user.mail }}</p>
           <p>{{ user.phone }}</p>
       </div>
   {% endfor %}
</div>
</body>
...
```

Наследование шаблонов

Начнём с классической ситуации дублирования кода, который нарушает принцип DRY. Рассмотрим две html-страницы с большим объёмом одинакового кода.
Шаблон main.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>{{ title }}</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>

  <div class="row">
        <h1 class="col-12 col-md-6 display-2">Привет, меня зовут
Алексей</h1>
        <img src="/static/image/foto.jpg" class="col-12 col-md-6
img-fluid rounded-circle" alt="Моё фото">
  </div>

  <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>
```

Шаблон data.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>{{ title }}</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>

  <div class="row">
      <div class="col-12 col-md-6 col-lg-4">
          <p>Lorem ipsum dolor sit amet, consectetur adipisicing
elit. Culpa, fugiat obcaecati? Dignissimos earum facilis incidunt
modi, molestias mollitia nam quis recusandae voluptatum?</p>
      </div>
      <div class="col-12 col-md-6 col-lg-4">
              <p> Dicta id officia quibusdam vel voluptates. Ad
adipisci aliquid animi architecto commodi deleniti dolor
doloremque facilis fugiat hic illo nam odit officia placeat
provident quam quisquam quo reiciendis repudiandae sint suscipit
unde, velit voluptatem! </p>
      </div>
      <div class="col-12 col-md-6 col-lg-4">
                <p>Ab accusamus delectus et expedita id iste,
laboriosam optio quam, recusandae sed veritatis voluptate!
Accusamus blanditiis debitis et tempora. Ab architecto asperiores
aut consequuntur distinctio earum iusto nihil, non odit quidem
soluta veniam.</p>
      </div>
  </div>

  <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>
```

Для того, чтобы выводить эту пару страниц достаточно несколько строк кода на Flask

```
@app.route('/main/')
def main():
    context = {'title': 'Главная'}
    return render_template('main.html', **context)
```

```python
@app.route('/data/')
def data():
    context = {'title': 'База статей'}
    return render_template('data.html', **context)
```

На каждой странице всего несколько различных строк в середине. Остальной код дублируется, Представьте, что у вас большой проект на десятки аналогичных страниц. Сколько же времени вы затратите, чтобы изменить шапку или футер во всём проекте?

Базовый и дочерние шаблоны

Создадим базовый шаблон base.html, который будет включать весь одинаковый код.

```
<!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>
```

Исключённый текст для заголовка сайта был заменён на:

```
{% block title %} Мой сайт {% endblock %}
```

Для содержимого страницы код заменён на:

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

Количество блоков в базовом шаблоне и их названия зависят от задачи, которую решает разработчик. Содержимое внутри block впоследствии будет заполнено дочерними шаблонами. Инструкция block принимает один аргумент — название блока. Внутри шаблона это название должно быть уникальным, иначе возникнет ошибка.

Если в дочернем шаблоне блок отсутствует, выводится информация из базового шаблона. В нашем примере, если в дочернем шаблоне не прописать блок title, будет выведено значение «Мой сайт» из базового шаблона, а вместо содержимого увидим что “Страница не заполнена”

Теперь из main.html и data.html можно удалить дублирующиеся строки и указать, что эти шаблоны расширяют базовый.

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
Алексей</h1>
          <img src="/static/image/foto.jpg" class="col-12 col-md-6
```

```
img-fluid rounded-circle" alt="Моё фото">
   </div>
{% endblock %}
```

Шаблон data.html

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

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

{% block content %}
  <div class="row">
      <div class="col-12 col-md-6 col-lg-4">
           <p>Lorem ipsum dolor sit amet, consectetur adipisicing
elit. Culpa, fugiat obcaecati? Dignissimos earum facilis incidunt
modi, molestias mollitia nam quis recusandae voluptatum?</p>
      </div>
      <div class="col-12 col-md-6 col-lg-4">
              <p> Dicta id officia quibusdam vel voluptates. Ad
adipisci aliquid animi architecto commodi deleniti dolor
doloremque facilis fugiat hic illo nam odit officia placeat
provident quam quisquam quo reiciendis repudiandae sint suscipit
unde, velit voluptatem! </p>
      </div>
      <div class="col-12 col-md-6 col-lg-4">
                <p>Ab accusamus delectus et expedita id iste,
laboriosam optio quam, recusandae sed veritatis voluptate!
Accusamus blanditiis debitis et tempora. Ab architecto asperiores
aut consequuntur distinctio earum iusto nihil, non odit quidem
soluta veniam.</p>
      </div>
  </div>
{% endblock %}
```

Содержимое одноимённых блоков в дочерних шаблонах будет подставлено в соответствующее место базового.

🔥 **Внимание!** Использование переменной {{ super() }} в дочерних шаблонах позволяет выводить содержимое родительского блока, а не заменять его!

После такой оптимизации достаточно внести изменение в базовом шаблоне, чтобы обновить одинаковую информацию на всех страницах сайта.
Дочерние шаблоны компакты и содержат только специфичную для страницы информацию. А при отрисовке через Jinja в них легко передавать динамически изменяемую информацию.

🔥 **Важно!** Сохранять текстовую информацию внутри html файла как в data.html нелогично. Она должна храниться в базе данных. А шаблон в этом случае может получать её через контекст и выводить в цикле.

Вывод

На этой 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

Домашнее задание

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