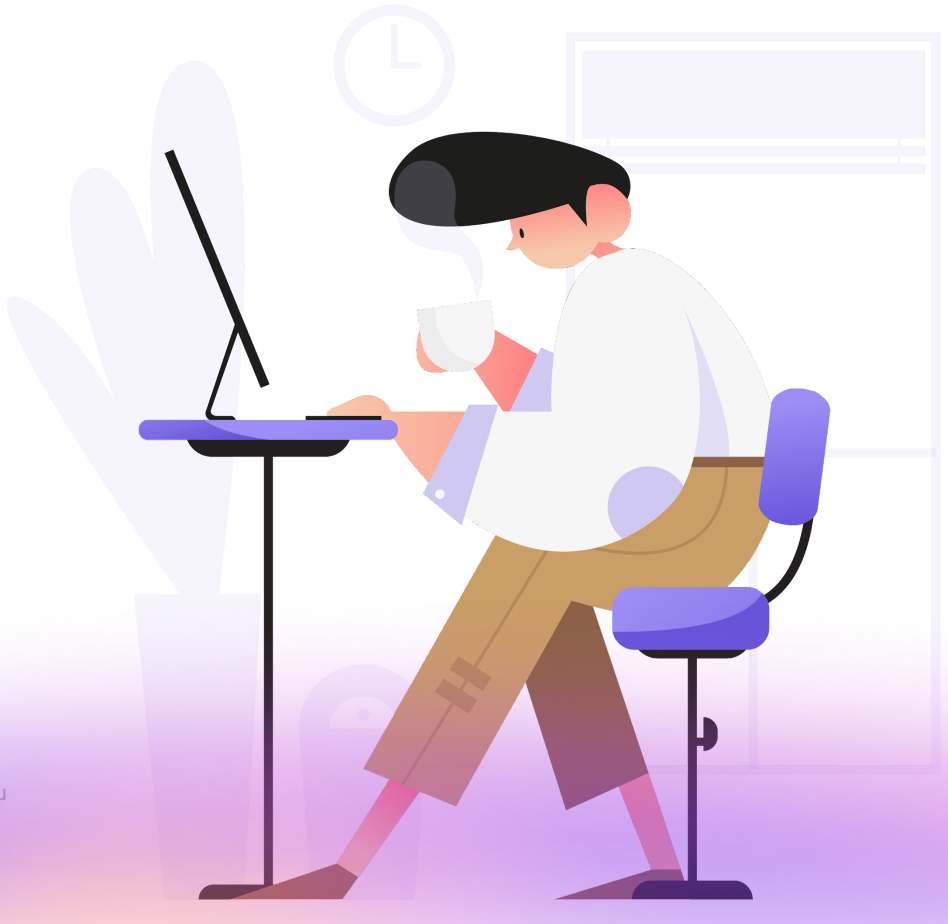
GeekBrains

Flask

Шаблоны Jinja2. Комплексные приложения на Flask. Blueprints

- Flask v1.1.x
- Python >= 3.8



На этом уроке

1. Узнаем, что такое Blueprints.
2. Познакомимся с Jinja2 и применим на практике.
3. Научимся подключать статику к проекту.
4. Используем новые функции-помощники.

Оглавление

Теория

Jinja2

Jinja2 — это 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. Он подобен шаблонизатору Django, но предоставляет Python-подобные выражения, обеспечивая исполнение шаблонов в песочнице. Это текстовый шаблонизатор, поэтому он может быть использован для создания любого вида разметки, а также исходного кода. Jinja позволяет настраивать теги, фильтры, тесты и глобальные переменные. Также, в отличие от шаблонизатора Django, Jinja позволяет конструктору шаблонов вызывать функции с аргументами на объектах.

Возможности:

- песочница для исполнения;
- мощная автоматическая HTML-экранирующая система для предотвращения XSS;
- наследование шаблонов;
- компилируется до оптимального кода python just in time;
- опциональная досрочная компиляция шаблонов;
- легко отлаживается. Номера строк исключений непосредственно указывают на нужную строку в шаблоне;
- настраиваемый синтаксис.

Управляющие конструкции позволяют добавлять в шаблоны элементы управления потоком и циклы. По умолчанию управляющие конструкции используют разделитель `{% ... %}` вместо двойных фигурных скобок `{{ ... }}`.

Цикл for позволяет перебирать последовательность. В этом примере мы выводим всех пользователей на страницу при помощи цикла:

```
{% set user_list = ['Bob', 'Alice', 'John'] %}

<ul>
{% for user in user_list %}
  <li>{{ user }}</li>
{% endfor %}
</ul>
```

Blueprints

Flask использует концепцию *blueprint'ов* (blueprint — «эскиз») для создания компонентов приложений и поддержки общих шаблонов внутри приложения или между приложениями. Блупринты могут как значительно упростить большие приложения, так и предоставить общий механизм регистрации в приложении операций из расширений Flask. Объект **Blueprint** работает аналогично объекту приложения **Flask**, но в действительности он не является приложением. Обычно это лишь *эскиз* для сборки или расширения приложения.

Блупринты во Flask могут пригодиться, если нужно:

- разделить приложения на набор блупринтов. Они идеальны для больших приложений; проект должен создать объект приложения, инициализировав несколько расширений и зарегистрировав набор blueprints;
- зарегистрировать blueprint в приложении по определённом префиксу URL и/или в поддомене. Параметры в префиксе URL или поддомене становятся обычными аргументами представлений (со значениями по умолчанию) для всех функций представлений в блупринте;
- зарегистрировать blueprint несколько раз в приложении с разными правилами URL;
- предоставить фильтры шаблонов, статические файлы, шаблоны и другие вспомогательные средства с помощью блупринтов. Blueprint не является реализацией приложения или функций представлений;
- зарегистрировать blueprint в приложении в любом из этих случаев при инициализации расширения Flask.

Blueprint во Flask не является подключаемым приложением, потому что это на самом деле не приложение — это набор операций, которые могут быть зарегистрированы в приложении, возможно даже не один раз. Почему бы не воспользоваться несколькими объектами приложений? Вы можете это сделать, но ваши приложения будут иметь раздельные файлы конфигурации и будут управляться слоем WSGI.

Вместо этого блупринты предоставляют разделение на уровне Flask, позволяя использовать общий файл конфигурации приложения, и могут менять объект приложения необходимым образом при регистрации. Побочным эффектом будет невозможность отменить регистрацию blueprint, если приложение уже было создано — только уничтожить целиком весь объект приложения.

Основная концепция блупринтов заключается в том, что они записывают операции для выполнения при регистрации в приложении. Flask связывает функции представлений с блупринтами при обработке запросов и генерировании URL'ов от одной конечной точки к другой.

Bootstrap

Bootstrap — свободный набор инструментов для создания сайтов и веб-приложений. Включает в себя HTML- и CSS-шаблоны оформления для типографики, веб-форм, кнопок, меток, блоков навигации и прочих компонентов веб-интерфейса, включая JavaScript-расширения.

Функции

render_template

Функция используется для визуализации шаблонов. Генерация HTML из Python — невесёлое и довольно сложное занятие, так как вам необходимо самостоятельно заботиться о безопасности приложения, производя для HTML обработку специальных последовательностей (escaping). Для визуализации шаблона вы можете использовать метод **render_template()**. Всё, что вам необходимо, — это указать имя шаблона, а также переменные в виде именованных аргументов, которые вы хотите передать движку обработки шаблонов:

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask будет искать шаблоны в папке templates. Структура такая:

```
/app.py
/templates
    /hello.html
```

Пример шаблона (файл `hello.html`):

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello, World!</h1>
{% endif %}
```

redirect

Чтобы перенаправить пользователя в иную конечную точку, используйте функцию **redirect().** Она возвращает объект ответа (приложение WSGI), который при вызове перенаправляет клиента в целевое расположение. Например, можно вернуть ответ `redirect('/')`, `redirect('/home/')`, `redirect('/auth/login/')` и так далее. Поддерживаемые коды: 301, 302, 303, 305, 307 и 308. Не поддерживаются 300, потому что это не настоящее перенаправление, и 304, потому что это ответ на запрос с определенными заголовками If-Modified-Since.

url_for

Используется для построения (генерации) URL. Раз Flask может искать соответствия в URL, может ли он их генерировать? Конечно, да. Для построения URL для специфической функции вы можете использовать функцию **url_for()**. В качестве первого аргумента она принимает имя функции, кроме того, она принимает ряд именованных аргументов, каждый из которых соответствует переменной части правила для URL.

Например: `url_for('index')`, `url_for('user_details', user_id=42)`.

Неизвестные переменные части добавляются к URL в качестве параметров запроса.

Практическое задание

Создаём шаблоны: базовый шаблон и шаблон индекса

Внутри модуля blog создаём папку для шаблонов — templates (это имя будет искать Flask). Создаём в ней шаблоны: base.html и index.html.

В base.html добавляем стандартную базовую разметку страницы и указываем блоки:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>
    {% block title %}
      Base
    {% endblock %}
  </title>
</head>
<body>
  <div>
    {% block body %}
      Hello base!
    {% endblock %}
  </div>
</body>
</html>
```

Теги добавляются при помощи {% %}. Добавляем теги `block` и именуем их `title` и `body`. Обязательно закрываем теги `block` при помощи `endblock`.

Создаём index.html. В нём мы расширяем шаблон base.html при помощи тега `extends` и переопределяем блоки `title` и `body`:

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

{% block title %}
  Index Page
{% endblock %}

{% block body %}
  <h1>Hello index page!</h1>
{% endblock %}
```

В модуле блога обновляем файл app.py — добавляем код для отрисовки шаблона:

```
from flask import Flask, render_template

app = Flask(__name__)

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

Создаём blueprint для отрисовки страниц пользователя, view для списка пользователей и шаблон

Теперь внутри шаблонов мы создаём подпапку users и там создаём шаблон страницы со списком пользователей: blog/templates/users/list.html. Нам важно создавать шаблоны в отдельных папках по двум причинам:

- при работе с несколькими блупринтами таким образом мы защищаем себя от переопределения list.html файлом с таким же именем, но для другой секции. Нам не нужно придумывать для каждого приложения users-list.html и т.д.;
- мы отделяем папку с шаблонами, как и отделяем кусок логики при помощи блупринтов.

Снова расширяем базовый шаблон. Таким образом мы сможем соблюдать единый стиль на всех страницах нашего сайта и использовать одни и те же строчки кода для всех страниц (например, навигационная панель, которую мы добавим позже).

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

{% block title %}
  Users list
{% endblock %}

{% block body %}
  <h1>Users</h1>
  <div>
    <ul>
      {% for user_id in users %}
        <li>{{ users[user_id] }}</li>
      {% endfor %}
    </ul>
  </div>
{% endblock %}
```

Здесь мы используем цикл, чтобы вывести каждого пользователя. Сейчас передача пользователей будет реализована в формате простого словаря. Выводим в виде списка. Обратите внимание, что цикл нужно завершать соответствующим тегом.

В модуле blog создаём модуль views (папка views и файл __init__.py). Здесь у нас будут все блупринты.

Создаём блупринт юзеров в файле blog/views/users.py:

```
from flask import Blueprint, render_template

users_app = Blueprint("users_app", __name__)
USERS = {
    1: "James",
    2: "Brian",
    3: "Peter",
}

@users_app.route("/")
def users_list():
    return render_template("users/list.html", users=USERS)
```

В Blueprint необходимо передать название «приложения» и имя текущего файла. Создаём словарь, который будет представлять базу данных с пользователями. Регистрируем view для обработки обращения к списку пользователей. Отрисовываем шаблон, не забываем передать туда пользователей.

Создаём view и шаблон страницы отдельного пользователя. Добавляем обработку «несуществующего пользователя»

Создаём шаблон blog/templates/users/details.html:

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

{% block title %}
  User #{{ user_id }}
{% endblock %}

{% block body %}
  <h1>{{ user_name }}</h1>
{% endblock %}
```

Добавляем отрисовку шаблона с передачей выбранного пользователя. Не забываем про обработку ошибки, если пользователь не найден — blog/views/users.py:

```
from werkzeug.exceptions import NotFound

@users_app.route("/<int:user_id>/")
def user_details(user_id: int):
    try:
        user_name = USERS[user_id]
    except KeyError:
        raise NotFound(f"User #{user_id} doesn't exist!")
    return render_template('users/details.html', user_id=user_id,
user_name=user_name)
```

Теперь подключаем блупринт и указываем приложению, какой url_prefix использовать для всех вложенных views (файл app.py):

```
from blog.views.users import users_app

app.register_blueprint(users_app, url_prefix="/users")
```

Создаём эндпоинты для приложения users, используем url_for для ссылок

В blog/views/users.py добавляем имена этих view: endpoint. Теперь мы можем ссылаться на этот view по данному имени endpoint.

```
@users_app.route("/", endpoint="list")
...

@users_app.route("/<int:user_id>/", endpoint="details")
...
```

Теперь можно отредактировать шаблоны:

1. В шаблоне blog/templates/users/details.html добавляем ссылку «обратно» — к списку пользователей:

```
<div>
  <a href="{{ url_for('users_app.list') }}">Back to users list</a>
</div>
```

2. В списке пользователей blog/templates/users/list.html вместо простого списка делаем список со ссылками на просмотр выбранного пользователя:

```
<li>
  <a href="{{ url_for('users_app.details', user_id=user_id) }}">
      {{ users[user_id] }}
  </a>
</li>
```

Добавляем и подключаем Bootstrap

Для начала необходимо скачать статику и скопировать в папку blog/static. Рекомендуется разные типы статики складывать в разные папки, то есть в итоге будет подобная структура:

```
blog
├─ static
│    ├── css
│    │    ├── bootstrap.min.css
│    │    └── bootstrap.min.css.map
│    └── js
│         ├── bootstrap.bundle.min.js
│         └── bootstrap.bundle.min.js.map
```

Наполняем blog/templates/base.html на основе примера Bootstrap (starter template). И подключаем статику при помощи url_for('static'...

```
<!-- Required meta tags -->
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">

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

-----

<div class="container">
  {% block body %}
    Hello base!
  {% endblock %}
</div>
```

Добавляем навигацию с ссылками

Продолжаем менять blog/templates/base.html в body:

```
{# nav bar #}
<nav class="navbar navbar-expand-md navbar-light bg-light">
 <div class="container-fluid">
   <a class="navbar-brand" href="{{ url_for('index') }}">Blog</a>
   <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup"
aria-expanded="false" aria-label="Toggle navigation">
     <span class="navbar-toggler-icon"></span>
   </button>
   <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
     <div class="navbar-nav">
       {% for (endpoint, label) in [('users_app.list', 'Users')] %}
         <a href="{{ url_for(endpoint) }}"
            class="nav-link {% if request.endpoint == endpoint -%}active{%- endif %}"
         >
           {{ label }}
         </a>
       {% endfor %}
     </div>
   </div>
 </div>
</nav>
```

Для работы динамических компонентов (любых действий на странице, таких как выпадающие меню, активные кнопки и т.д.) нужно обязательно добавить скрипт в head:

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

Создаём blueprint для статей, страницу-список и шаблон, добавляем в навигационную панель

Действуем похожим образом, как создавали шаблоны и вьюшки для пользователей.

Создаём шаблон blog/templates/articles/list.html:

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

{% block title %}
  Articles list
{% endblock %}

{% block body %}
  <h1>Articles</h1>
  <div>
    <ul>
      {% for title in articles %}
        <li>
          {{ title }}
        </li>
      {% endfor %}
    </ul>
  </div>
{% endblock %}
```

И пишем код самого blog/views/articles.py:

```
from flask import Blueprint, render_template

articles_app = Blueprint("articles_app", __name__)


ARTICLES = ["Flask", "Django", "JSON:API"]


@articles_app.route("/", endpoint="list")
def articles_list():
    return render_template("articles/list.html", articles=ARTICLES)
```

В базовом шаблоне добавляем ссылку на статьи в навигационной панели:

```
{% for (endpoint, label) in [('users_app.list', 'Users'), ('articles_app.list', 'Articles')] %}
```

Снова возвращаемся в blog/app.py, добавляем импорт и регистрируем новый блупринт:

```
from blog.views.articles import articles_app
...

app.register_blueprint(articles_app, url_prefix="/articles")
```

Итоги

На занятии мы рассмотрели Jinja2 и предоставляемые инструменты: наследование шаблонов, условные выражения, циклы, блоки. В шаблонах мы применили Bootstrap для стилей и действий. Также уделили внимание дополнительным функциям-помощникам render_template, redirect, url_for, которые часто используются во view и шаблонах.

Практическое задание

1. Добавить в свой проект на Flask использование шаблонов и стилей Bootstrap.
2. В базовый шаблон добавить навигационную панель, которая будет отображаться на всех страницах ресурса.
3. Реализовать страницы со списком доступных статей и пользователей, а также возможность перехода к их деталям.

Глоссарий

Bootstrap — это открытый и бесплатный HTML, CSS и JS фреймворк, который используется веб-разработчиками для быстрой вёрстки адаптивных дизайнов сайтов и веб-приложений.

Дополнительные материалы

1. https://getbootstrap.com.
2. https://jinja.palletsprojects.com/en/2.11.x/.
3. https://flask.palletsprojects.com/en/1.1.x/.

Используемые источники

1. Официальная документация DRF.
2. REST простым языком.
3. REST и HTTP.
4. Методы HTTP и их назначение.