Flask

Flask - це легкий фреймворк для створення веб-додатків на мові Python. Він надає базові інструменти та бібліотеки для розробки веб-додатків та API. Щоб опанувати Flask, вам слід знати наступні ключові аспекти:

1. **Маршрутизація (Routing)**: У Flask використовується декоратори для визначення URL-адрес та функції, яка повинна бути викликана при обробці запиту. Наприклад:

   ```
   from flask import Flask

   app = Flask(__name__)

   @app.route('/')
   def index():
       return 'Hello, World!'
   ```

   У цьому прикладі, коли користувач переходить на головну сторінку (`'/'`), викликається функція `index()` , яка повертає рядок "Hello, World!".

2. **Шаблонізатори (Templates)**: Flask підтримує використання шаблонізаторів, таких як Jinja2, для відображення сторінок з динамічним контентом. Шаблонізатори дозволяють вбудовувати змінні та структури даних у статичний HTML-код.

3. **Робота з формами**: Flask надає інструменти для обробки введення користувача через HTML-форми. Ви можете використовувати бібліотеки, такі як Flask-WTF, для спрощення роботи з формами.

4. **Робота з базами даних**: Хоча Flask сам по собі не накладає обмежень на тип бази даних, з яким можна працювати, багато розробників використовують SQLAlchemy для роботи з реляційними базами даних, або Flask-MongoEngine для NoSQL баз даних.

5. **Обробка запитів та відповідей (Requests and Responses)**: Flask надає зручний інтерфейс для отримання даних з HTTP-запитів (GET, POST, PUT, DELETE) та формування відповідей, що включають статус-коди, заголовки та вміст.

6. **Керування сесіями та аутентифікація**: Flask дозволяє працювати з сесіями, що дозволяє зберігати дані про користувача на протязі декількох запитів. Також існують розширення, як Flask-Login, для реалізації аутентифікації користувачів.
7. **Розширення (Extensions)**: Flask має велику кількість розширень, які дозволяють легко додавати додатковий функціонал до вашого додатку, такий як робота з формами, авторизація, робота з базами даних тощо.
8. **Управління конфігурацією**: Flask надає можливість налаштувати різні параметри за допомогою конфігураційних файлів або змінних середовища.
9. **Розгортання (Deployment)**: Важливо знати, як розгортати ваш Flask додаток, щоб зробити його доступним для користувачів. Для цього можна використовувати різні платформи, такі як Heroku, AWS, або власний сервер.

Це загальний огляд основних аспектів Flask. З кожним з цих аспектів варто ознайомитися більш докладно, щоб глибше розуміти, як працює цей фреймворк та як використовувати його для розробки ваших власних веб-додатків.

Установка Flask

Наступним кроком є встановлення Flask, але перед тим, як я розповімо про це, я хочу розповісти вам про найкращі методи, пов'язані з встановленням пакетів Python.

У Python пакети, такі як Flask, доступні в загальному репозиторії, звідки їх можна завантажити та встановити. Офіційний репозиторій пакетів Python називається PyPI, що означає Python Package Index (також відомий як "cheese shop"). Встановлення пакету з PyPI дуже просте, оскільки у Python є інструмент під назвою pip, який виконує цю роботу (у Python 2.7 pip не входить до складу Python і його потрібно встановлювати окремо).

Щоб встановити пакет на свій комп'ютер, ви використовуєте pip наступним чином:

```
$ pip install <ім'я-пакету>
```

Цікаво, що цей метод встановлення пакетів може не працювати в більшості випадків. Якщо ваш інтерпретатор Python був встановлений глобально для всіх користувачів вашого комп'ютера, ймовірно, що ваш запис користувача не матиме прав на внесення змін у нього, тому єдиний спосіб виконати вищезазначену команду -

запустити її від імені адміністратора. Проте навіть без цього ускладнення розумійте, що відбувається, коли ви встановлюєте пакет, таким чином, як описано вище. Інструмент pip завантажить пакет з PyPI, а потім додасть його до вашого каталогу Python. З цього моменту кожний сценарій Python, який у вас є на системі, матиме доступ до цього пакету. Уявіть ситуацію, коли ви закінчили веб-додаток, використовуючи версію 0.11 Flask, яка була останньою версією Flask на момент запуску, але тепер її замінила версія 0.12. Тепер ви хочете запустити другий додаток, для якого вам би хотілося використовувати версію 0.12, але якщо ви замініте встановлену версію 0.11, ви ризикуєте зламати свій старий додаток. Бачите проблему? Було б ідеально, якби можна було встановити Flask 0.11, який використовуватиметься вашим старим додатком, а також встановити Flask 0.12 для вашого нового.

Щоб вирішити проблему підтримки різних версій пакетів для різних додатків, Python використовує концепцію віртуальних середовищ. Віртуальне середовище - це повна копія інтерпретатора Python. Коли ви встановлюєте пакети в віртуальному середовищі, загальносистемний інтерпретатор Python не торкається, тільки копія. Таким чином, рішення мати повну свободу для встановлення будь-яких версій ваших пакетів для кожного додатка - використовувати інше віртуальне середовище для кожного додатка. Віртуальні середовища мають додаткову перевагу - вони належать користувачеві, який їх створює, тому не потрібна облікова запис адміністратора.

Подальша частина може бути виконана просто у pycharm, тому її можна пропустити

Почнемо з створення каталогу, в якому буде розташований проект. Я збираюся назвати цей каталог microblog, так як це ім'я додатка:

```
$ mkdir microblog
$ cd microblog
```

Якщо ви використовуєте Python 3, в нього включено підтримку віртуальних середовищ, тому все, що вам потрібно зробити для їх створення, це:

```
$ python3 -m venv venv
```

За допомогою цієї команди Python запустить пакет venv, який створить віртуальне середовище з іменем venv. Перше venv у команді - це ім'

я пакета віртуального середовища Python, а друге - ім'я віртуального середовища, яке я буду використовувати для цієї конкретної середовища. Якщо ви вважаєте, що це заплутує вас, ви можете замінити другий venv іншим ім'ям, яке ви хочете призначити своєму віртуальному середовищу. Загалом я створюю свої віртуальні середовища з іменем venv в каталозі проекту, тому кожного разу, коли я підключаюся до проекту, я знаходжу його відповідне віртуальне середовище.

Зверніть увагу, що в деяких операційних системах вам може знадобитися використовувати python замість python3 в приведеній вище команді. Деякі установки використовують python для випусків Python 2.x і python3 для випусків 3.x, тоді як інші відображають python у випусках 3.x.

По завершенні команди у вас буде каталог з іменем venv, де зберігаються файли віртуального середовища.

Якщо ви використовуєте будь-яку версію Python старшу за 3.4 (включаючи випуск 2.7), віртуальні середовища не підтримуються відразу. Для цих версій Python вам потрібно завантажити і встановити сторонній інструмент virtualenv, перед тим як створювати віртуальні середовища. Після того, як virtualenv встановлено, ви можете створити віртуальне середовище за допомогою наступної команди:

```
$ virtualenv venv
```

або так

```
$ python virtualenv.py venv
```

Примітка перекладача: У мене встановлено кілька версій Python. Я використовую Python3.3. У моєму випадку довелося вводити рядок так:

```
C:\\microblog>c:\\Python33\\python.exe c:\\Python33\\Lib\\site-packages\\virtualenv.py venv
```

У отриманому повідомленні видно, що встановлено pip та кілька пакетів:

```
Using base prefix 'c:\\\\Python33'
New python executable in C:\\microblog\\venv\\Scripts\\python.exe
Installing setuptools, pip, wheel...done.
```

Незалежно від методу, який ви використовували для його створення, ви створили своє власне віртуальне середовище. Тепер вам потрібно повідомити системі, що ви хочете його використовувати, активуючи його. Щоб активувати нове віртуальне середовище, використовуйте наступну команду:

```
$ source venv/bin/activate
(venv) $ _
```

Якщо ви використовуєте cmd (командний рядок Microsoft Windows), команда активації трохи відрізняється:

```
$ venv\\Scripts\\activate
(venv) $ _
```

Коли ви активуєте віртуальне середовище, конфігурація терміналу змінюється так, що інтерпретатор Python, який знаходиться всередині нього, стає активним, і його можна запустити, набираючи "python". При цьому у запиті терміналу вказується ім'я активованого віртуального середовища. Зміни, внесені у терміналі, є тимчасовими і обмежені лише цим сеансом, і вони не будуть збережені при закритті вікна терміналу. Якщо ви одночасно працюєте з декількома вікнами терміналу, кожне з них матиме своє власне активоване віртуальне середовище.

Тепер, коли ви створили і активували віртуальне середовище, ви можете нарешті встановити Flask:

```
(venv) C:\\microblog>pip install flask
Collecting flask
  Using cached Flask-0.12.2-py2.py3-none-any.whl
Requirement already satisfied: click>=2.0 in c:\\python33\\lib\\site-packages (from flask)
Requirement already satisfied: Werkzeug>=0.7 in c:\\python33\\lib\\site-packages (from flas
k)
Requirement already satisfied: Jinja2>=2.4 in c:\\python33\\lib\\site-packages (from flask)
Requirement already satisfied: itsdangerous>=0.21 in c:\\python33\\lib\\site-packages (from
flask)
Requirement already satisfied: markupsafe in c:\\python33\\lib\\site-packages (from Jinja2>
```

```
=2.4->flask)
Installing collected packages: flask
Successfully installed flask-0.12.2
```

Якщо ви хочете перевірити, що Flask встановлено в вашому віртуальному середовищі, ви можете запустити інтерпретатор Python і імпортувати Flask:

```
(venv) C:\\microblog>python
Python 3.3.5 (v3.3.5:62cf4e77f785, Mar  9 2014, 10:37:12) [MSC v.1600 32 bit (Intel)] on wi
n32
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>>
```

Якщо імпорт не видає жодних помилок, це означає, що Flask успішно встановлено і готове до використання.

Фласк додаток "Привіт, світе"

Якщо ви відвідаєте веб-сайт Flask, вас привітає дуже простий приклад "Привіт, світе" з п'ятьма рядками коду. Замість повторення цього тривіального прикладу, я покажу вам трохи складніший, який надасть вам хорошу основу для створення більших додатків.

Додаток існуватиме у формі пакета.

Примітка перекладача: Пакет - це колекція модулів.

В Python підкаталог, який містить файл **__init__**.py, вважається пакетом і може бути імпортований. При імпортуванні пакета **__init__**.py виконується і визначає, які символи пакет надає для зовнішнього світу.

Давайте створимо пакет з назвою app, в якому буде розміщено наше додаток. Переконайтеся, що ви знаходитесь в каталозі microblog, а потім виконайте таку команду:

```
(venv) $ mkdir app
```

__init__.py для пакета додатку буде містити такий код:

```
from flask import Flask

app = Flask(__name__)

from app import routes
```

Цей сценарій просто створює об'єкт додатку як екземпляр класу Flask, імпортованого з пакета flask. Змінна **name**, яку передає класу Flask, є вбудованою змінною Python, яка визначається іменем модуля, в якому вона використовується. Flask використовує розташування модуля, переданого сюди, як точку виходу, коли йому потрібно завантажити пов'язані ресурси, такі як файли шаблонів, про які я розкажу в розділі 2. Для всіх практичних цілей передача **name** майже завжди буде налаштовувати Flask в правильному напрямку. Потім додаток імпортує модуль routes, який ще не існує.

Одним

з аспектів, який може здатися заплутаним спочатку, є те, що існують два об'єкти з іменем app. Пакет додатку визначається каталогом додатку та сценарієм **__init__**.py і входить в пакет app. Локальна змінна app визначається як екземпляр класу Flask в сценарії **__init__**.py, що робить його частиною пакета додатку.

Ще одна особливість полягає в тому, що модуль routes імпортується внизу, а не вгорі сценарія, як це завжди робиться. Нижній імпорт є обхідним шляхом для уникнення циклічного імпорту, який є загальною проблемою при використанні додатків Flask. Ви побачите, що модуль routes повинен буде імпортувати змінну додатку, визначену в цьому сценарії, тому розміщення одного з обох взаємних імпортів внизу дозволить уникнути помилки, яка виникає через взаємні посилання між цими двома файлами.

Тепер що включити до модуля routes? routes - це різні URL-адреси, які реалізує додаток. У Flask обробники маршрутів додатків записуються як функції Python, які називаються функціями перегляду. Функції перегляду відповідають одному або кільком URL-адресам маршрутів, тому Flask знає, яку логіку слід виконувати, коли клієнт запитує цей URL-адрес.

Ось ваша перша функція перегляду, яку вам потрібно написати в новому модулі з іменем app/routes.py:

```
from app import app
```

```
@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

Ця функція перегляду насправді досить проста, вона просто повертає вітання у вигляді рядка. Дві дивні строки @app.route над функцією - це декоратори, унікальна особливість мови Python. Декоратор змінює функцію, яка йде за ним. Загальний шаблон з декораторами - використовувати їх для реєстрації функцій як зворотних викликів для певних подій. В даному випадку декоратор @app.route створює зв'язок між URL-адресою, вказаною як аргумент, і функцією. У цьому прикладі є два декоратори, які зв'язують URL-адреси / і /index з цією функцією. Це означає, що коли веб-браузер запитує одну з цих двох URL-адрес, Flask викликає цю функцію і передає значення, яке повертається, назад у браузер як відповідь. Якщо вам здається, що це ще не має сенсу, це не триватиме довго, поки ви запустите цей додаток.

Щоб завершити додаток, вам потрібно створити сценарій Python на верхньому рівні, який визначає екземпляр додатку Flask. Давайте назвемо цей сценарій microblog.py і визначимо його як один рядок, який імпортує екземпляр додатку:

```
from app import app
```

Пам'ятаєте два об'єкти app? Тут ви можете побачити обидва разом в одному реченні. Екземпляр додатку Flask називається app і входить до пакета app. from app import app імпортує змінну app, яка входить до пакета app. Якщо ви вважаєте це заплутаним, ви можете перейменувати або пакет, або змінну на щось інше.

Щоб переконатися, що ви все робите правильно, нижче наведено схему структури проекту:

```
microblog/
  venv/
  app/
    __init__.py
    routes.py
  microblog.py
```

Вірте чи ні, але перша версія додатку завершена! Перш ніж запускати його, Flask потрібно повідомити, як його імпортувати, встановивши змінну середовища FLASK_APP:

```
(venv) $ export FLASK_APP=microblog.py
```

Якщо ви використовуєте Microsoft Windows, скористайтеся командою 'set' замість 'export' у вищезазначеній команді.

Готові бути враженими? Ви можете запустити свій перший веб-додаток за допомогою наступної команди:

```
(venv) $ flask run
```

```
Службовий Flask додаток "microblog"
```

```
Запущено на http://127.0.0.1:5000/ (Натисніть CTRL+C для виходу)
```

Примітка перекладача: Я був вражений, оскільки отримав помилку кодування. Якщо у вас версія Python старша за 3.6, ймовірно, вас чекає сюрприз. Типу:

Syntax Error: (unicode error) 'utf-8' codec can't decode byte 0xcf in position 0: invalid continuation byte:

У моєму випадку винні були кириличні символи у назві комп'ютера. Я змінив їх на PK і все працювало. Винний модуль socket

```
(venv) C:\microblog>python
Python 3.3.5 (v3.3.5:62cf4e77f785, Mar  9 2014, 10:37:12) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

```
from socket import gethostbyaddr
gethostbyaddr('127.0.0.1')
('Acer-PK', [], ['127.0.0.1'])
```

Дійсно все зафункціонувало:

```
(venv) C:\microblog>set FLASK_APP=microblog.py
```

```
(venv) C:\microblog>flask run
```

```
Службовий Flask додаток "microblog"
```

```
Запущено на http://127.0.0.1:5000/ (Натисніть CTRL+C для виходу)
```

Щоб написати "Привіт, світ!" українською, потрібно внести зміни у файл routes.py:

```
@app.route('/')
@app.route('/index')
def index():
  return "Привіт, світ!"
```

Коли ви закінчите грати з сервером, просто натисніть Ctrl-C, щоб зупинити його.

Шаблони

Інтро

Шаблони - це файли, які дозволяють відокремити вміст та дизайн веб-сторінки. У Flask шаблони записуються як окремі файли, що зберігаються в папці `templates`, яка розташована всередині пакета додатка.

Шаблони використовують синтаксис подібний до Python, але з власними особливостями. Наприклад, блоки `{{...}}` використовуються для вставки динамічного контенту, який буде визначений під час виконання.

Після створення шаблону, його можна використовувати в функції прогляду за допомогою `render_template()`. Ця функція приймає ім'я файлу шаблону та список

аргументів для шаблону. Вона повертає готовий HTML-код, в якому всі заповнювачі (`{{...}}`) замінені реальними значеннями.

Використання шаблонів дозволяє зберігати бізнес-логіку та презентацію окремо, що полегшує роботу над додатком та його масштабування.

Ось ваш перший шаблон, який має схожий за функціональністю вигляд з HTML-сторінкою, що повертається функцією перегляду `index()` вище. Запишіть цей файл у `app/templates/index.html`:

```
<html>
    <head>
        <title>{{ title }} - Мікроблог</title>
    </head>
    <body>
        <h1>Привіт, {{ user.username }}!</h1>
    </body>
</html>
```

Це стандартна та дуже проста HTML-сторінка. Єдине цікаве на цій сторінці - це те, що для динамічного контенту є декілька заповнювачів, що знаходяться в розділі `{{...}}`. Ці заповнювачі представляють частини сторінки, які є змінними і будуть визначені лише під час виконання.

Я сподіваюся, ви погодитеся зі мною, що рішення, використовуване вище, не є достатньо ефективним. Замисліться, наскільки складним буде код в цьому представленні, коли у мене з'являться повідомлення в блозі від користувачів, які постійно будуть змінюватися. Додаток також буде мати більше функцій перегляду, які будуть пов'язані з іншими URL-адресами, отже уявіть, якщо одного чудового дня я вирішу змінити макет цього додатка і доведеться оновлювати HTML у кожному представленні. Це, очевидно, не є варіантом, який масштабується з ростом додатка.

Якби ви могли відокремити логіку вашого додатка від його дизайну або представлення веб-сторінок, то все було б гораздо краще організовано, чи не так? Навіть можна найняти веб-дизайнера для створення вражаючого сайту, поки ви займаєтесь написанням логіки додатка на Python.

Шаблони допомагають досягти цього розділення між представленням та бізнес-логікою. У Flask, шаблони записуються як окремі файли, які зберігаються у папці templates, що знаходиться всередині пакету додатка. Так що, переконавшись, що ви

знаходитесь в каталозі microblog, створіть каталог, в якому будуть зберігатися шаблони:

(venv) $ mkdir app/templates

або так:

(venv) C:\microblog>mkdir app\templates

Нижче ви можете побачити свій перший шаблон, який схожий за функціональністю на HTML-сторінку, яку повертає функція перегляду index() вище. Запишіть цей файл у app/templates/index.html:

```
<html>
    <head>
        <title>{{ title }} - Мікроблог</title>
    </head>
    <body>
        <h1>Привіт, {{ user.username }}!</h1>
    </body>
</html>
```

Це стандартна та дуже проста HTML-сторінка. Єдине цікаве на цій сторінці - це те, що для динамічного контенту є декілька заповнювачів, що знаходяться в розділі `{{...}}`. Ці заповнювачі представляють частини сторінки, які є змінними і будуть визначені лише під час виконання.

Тепер, коли представлення сторінки було вивантажено в HTML-шаблон, функція перегляду може бути спрощена (файл \app\routes.py):

```
# -*- coding: utf-8 -*-
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return render_template('index.html', title='Головна', user=user)
```

Це виглядає набагато краще, чи не так? Спробуйте цю нову версію додатка, щоб побачити, як працює шаблон. Як тільки ви завантажите сторінку у свій браузер, ви можете переглянути вихідний HTML-код і порівняти його з вихідним шаблоном.

Операція, яка перетворює шаблон в повноцінну HTML-сторінку, називається рендерінгом. Щоб відобразити шаблон, мені довелося імпортувати функцію, яка постачається з флаговою інфраструктурою під назвою `render_template()`. Ця функція приймає ім'я файлу шаблона та список аргументів шаблона і повертає той самий шаблон, але при цьому всі заповнювачі в ньому замінюються фактичними значеннями.

Функція `render_template()` викликає механізм шаблонів Jinja2, який постачається разом із Flask. Jinja2 замінює блоки `{{...}}` відповідними значеннями, вказаними аргументами, вказаними в виклику `render_template()`.

Умовні оператори

Ви бачили, як Jinja2 замінює заповнювачі фактичними значеннями під час рендерингу, але це лише одна з багатьох потужних операцій, які підтримує Jinja2 в файлах шаблонів. Наприклад, шаблони також підтримують керуючі оператори, задані всередині блоків {% ...%}. Наступна версія шаблону `index.html` додає умовний вираз:

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Мікроблог</title>
        {% else %}
        <title>Ласкаво просимо до Мікроблогу!</title>
        {% endif %}
    </head>
    <body>
        <h1>Привіт, {{ user.username }}!</h1>
    </body>
</html>
```

Тепер шаблон трохи розумніший. Якщо функція перегляду забуває передати значення змінної заповнювач заголовка, замість того, щоб показувати порожній заголовок, шаблон надасть значення за замовчуванням. Ви можете спробувати, як працює ця умова, видаливши аргумент `title` в виклику функції `render_template()` у функції перегляду (файл `app\\routes.py`).

Цикли

Користувач, який увійшов до системи, ймовірно, захоче побачити останні повідомлення від підключених користувачів на домашній сторінці, тому зараз я збираюся розширити додаток для підтримки цього.

Знову використовую хитрість із уявним об'єктом, щоб створити деяких користувачів та деякі повідомлення для демонстрації:

```
# -*- coding: utf-8 -*-
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Ельдар Рязанов'}
    posts = [
        {
            'author': {'username': 'Джон'},
            'body': 'Чудовий день в Портленді!'
        },
        {
            'author': {'username': 'Сюзан'},
            'body': 'Фільм "Месники" був таким крутим!'
        },
        {
            'author': {'username': 'Ипполит'},
            'body': 'Яка огидна ця ваша заливна риба!!'
        }
    ]
    return render_template('index.html', title='Головна', user=user, posts=posts)
```

Для представлення користувацьких повідомлень я використовую список, де кожен елемент є словником і має поля Author і Body. Коли я дійду до реалізації користувачів та повідомлень в блозі справжньо, я намагатимуся зберегти ці імена полів, щоб вся робота, яку я виконав для проектування та тестування шаблону домашньої сторінки, використовуючи ці тимчасові об'єкти, продовжувала працювати і тоді, коли я представлю справжніх користувачів і повідомлення з бази даних.

У шаблоні мені потрібно вирішити нову проблему. Список повідомлень може мати будь-яку кількість елементів. Скільки повідомлень буде представлено на сторінці? Шаблон не може робити жодних припущень щодо того, скільки записів існує, тому він повинен бути готовий до відображення стільки повідомлень, скільки передає представлення у загальному способі.

Для цього типу проблем, Jinja2 пропонує for структури керування (файл `app/templates/index.html`):

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Мікроблог</title>
        {% else %}
        <title>Ласкаво просимо до Мікроблогу</title>
        {% endif %}
    </head>
    <body>
        <h1>Привіт, {{ user.username }}!</h1>
        {% for post in posts %}
        <div><p>{{ post.author.username }} говорить: <b>{{ post.body }}</b></p></div>
        {% endfor %}
    </body>
</html>
```

Просто, чи не так? Спробуйте цю нову версію додатка і не забудьте поекспериментувати з додаванням більше контенту до списку повідомлень, щоб побачити, як шаблон адаптується і завжди відображає всі повідомлення, які надсилає функція представлення.

Шаблони форм

Наступним кроком є додавання форми до HTML-шаблону, щоб її можна було візуалізувати на веб-сторінці. Поля, визначені в класі `LoginForm`, вміють відображати себе як HTML, тому це завдання досить просте. Нижче ви можете побачити шаблон для входу в систему, який я планую зберігати в файлі `app/templates/login.html`:

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

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}
```

```
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

У цьому шаблоні я використовую ще один раз `base.html`, як показано в главі 2, через розширення шаблону. Фактично я буду робити це з усіма шаблонами, щоб забезпечити однорідну компоновку, яка включає верхню навігаційну панель для всіх сторінок додатку.

У цьому шаблоні передбачається, що об'єкт форми, створений з класу `LoginForm`, буде наданий як аргумент, який можна побачити як посилання `form`. Цей аргумент буде переданий у функцію перегляду входу, яку я до цього моменту ще не написав.

HTML-елемент `<form>` використовується як контейнер для веб-форми. Атрибут `action` форми використовується для того, щоб повідомити веб-браузеру URL-адресу, який має використовуватися при надсиланні інформації, введеної користувачем у форму. Якщо для дії вказано порожню рядок, форма надсилається на URL-адресу, яка в даний момент знаходиться в адресному рядку, тобто URL-адресу, який відображає форму на сторінці. Атрибут `method` вказує метод HTTP-запиту, який має використовуватися при надсиланні форми на сервер. За замовчуванням він відправляється з запитом GET, але майже у всіх випадках використання POST-запиту поліпшує взаємодію з користувачем, оскільки запити цього типу можуть надсилати дані форми в тіло запиту, тоді як запити GET додають поля форми до URL-адреси, заповнюючи адресний рядок переглядача.

Атрибут `novalidate` використовується для того, щоб повідомити веб-браузеру не застосовувати перевірку до полів у цій формі, що фактично залишає це завдання додатку Flask, запущеному

на сервері. Використання `novalidate` є повністю необов'язковим, але для цієї першої форми важливо, щоб ви встановили його, оскільки це дозволить вам протестувати перевірку на стороні сервера пізніше в цьому розділі.

Аргумент шаблону `form.hidden_tag()` створює сховане поле, яке містить токен, який використовується для захисту форми від атак CSRF. Все, що вам потрібно зробити, щоб форма була захищеною, - це включити це сховане поле і визначити змінну `SECRET_KEY` в конфігурації Flask. Якщо ви піклуєтеся про ці дві речі, Flask-WTF зробить все решту за вас.

Якщо ви вже писали HTML-веб-форми раніше, можливо, ви знайшли дивним, що в цьому шаблоні немає HTML-полів. Це тому, що поля з об'єкта `Form` вміють відображати себе як HTML. Все, що мені потрібно було зробити, це включити `{{ form.<field_name>.label }}` там, де потрібен підпис поля, і `{{ form.<field_name>() }}` там, де потрібно саме поле. Для полів, які потребують додаткових атрибутів HTML, їх можна передати як аргументи. Поля `username` і `password` у цьому шаблоні приймають `size` як аргумент, який буде доданий до HTML-елементу `<input>` як атрибут. Також можна додавати класи CSS або ідентифікатори до полів форми.

Представлення форми

Останнім кроком перед тим, як ви побачите цю форму у браузері, є код нової функції представлення в додатку, яка відображає шаблон з попереднього розділу.

Так от, давайте напишемо нову функцію представлення, яка відповідає URL-адресі `/login`, створить форму і передасть її в шаблон для відображення. Ця функція представлення може бути розміщена в модулі `app/routes.py`, доповнюючи вміст:

```
from flask import render_template
from app import app
from app.forms import LoginForm

# ...

@app.route('/login')
def login():
    form = LoginForm()
    return render_template('login.html', title='Sign In', form=form)
```

Тут я імпортував клас `LoginForm` з `forms.py`, створив екземпляр об'єкта з нього і передав його в шаблон. Синтаксис `form = form` може виглядати дивно, але просто передає об'єкт форми, створений на попередньому рядку (і показаний праворуч), в шаблон з ім'ям `form` (показане ліворуч). Цього достатньо для відображення полів форми.

Щоб спростити доступ до форми входу, базовий шаблон повинен включати посилання на навігаційну панель:

```
<div>
    Microblog:
    <a href="/index">Home</a>
```

```
    <a href="/login">Login</a>
</div>
```

На цьому етапі можна запустити додаток і переглянути його в браузері. При запуску додатку введіть `http://localhost:5000/` у адресному рядку браузера, а потім натисніть посилання "Sign In" у верхній навігаційній панелі

Отримання даних з форми

Якщо ви намагаєтеся натискати кнопку "Увійти", браузер відображає помилку "Method Not Allowed" (Метод не дозволений). Це пов'язано з тим, що функція входу в систему з попереднього розділу виконує лише частину завдання. Вона може відображати форму на веб-сторінці, але у неї немає жодної логіки для обробки даних, введених користувачем. Це ще одна область, де Flask-WTF полегшує роботу. Нижче наведена оновлена версія функції представлення, яка приймає та перевіряє дані, надані користувачем:

```
from flask import render_template, flash, redirect

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for user {}, remember_me={}'.format(
            form.username.data, form.remember_me.data))
        return redirect('/index')
    return render_template('login.html', title='Sign In', form=form)
```

Першою новинкою у цій версії є аргумент `methods` у декораторі маршруту. Це повідомляє Flask, що ця функція представлення приймає запити GET і POST, перевизначаючи значення за замовчуванням, яке приймає тільки запити GET. Протокол HTTP вказує, що GET-запити - це ті, що повертають інформацію клієнту (у цьому випадку веб-переглядачу). Усі запити в додатку до цього часу належать до цього типу. POST-запити зазвичай використовуються, коли браузер надсилає дані форми на сервер (хоча, фактично, запити GET також можуть використовуватися для цієї мети, але це не рекомендується). Помилка "Method Not Allowed", яку браузер показав вам раніше, виникає тому, що браузер намагався надіслати запит POST, а додаток не був налаштований на його прийняття. Подаючи аргумент `methods`, ви повідомляєте Flask про необхідність приймати методи запиту.

Метод `form.validate_on_submit()` виконує всю обробку форми. Коли браузер надсилає запит GET для отримання веб-сторінки з формою, цей метод повертає `False`, тому в цьому випадку функція пропускає оператор `if` і переходить до відображення шаблону в останньому рядку функції.

Коли браузер надсилає запит POST в результаті натискання користувачем кнопки submit, `form.validate_on_submit()` збирає всі дані, запускає всі валідатори, прикріплені до полів, і якщо все в порядку, повертає `True`, сигналізуючи, що дані дійсні і можуть бути оброблені додатком. Але якщо хоча б одне поле не пройде перевірку, функція поверне `False`, і це призведе до того, що форма буде повернена користувачеві, наприклад, у випадку запиту GET. Пізніше я планую додати повідомлення про помилку, коли перевірка не вдасться.

Коли `form.validate_on_submit()` повертає значення `True`, функція входу в систему викликає дві нові функції, імпортовані з Flask. Функція `flash()` - це корисний спосіб показати повідомлення користувачеві. Багато додатків використовують цю техніку, щоб повідомити користувачеві, чи було яке-небудь дійство успішним чи ні. У цьому випадку я буду використовувати цей механізм як тимчасовий спосіб показати повідомлення, яке підтверджує, що додаток отримав дані для входу.

Другою новою функцією, яку використовує функція входу в систему, є `redirect()`. Ця функція вказує веб-переглядачеві автоматично перейти на іншу сторінку, вказану в якості аргумента. Ця функц

ія представлення використовує її для перенаправлення користувача на сторінку `/index` додатку.

Коли ви викликаєте функцію `flash()`, Flask зберігає повідомлення, але на веб-сторінках не з'являються магічні повідомлення. Шаблони додатка повинні відображати ці розгорнуті повідомлення таким чином, який працює для дизайну веб-сайту. Я планую додати ці повідомлення в базовий шаблон, щоб всі шаблони успадковували цю функціональність. Ось оновлений базовий шаблон:

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - microblog</title>
        {% else %}
        <title>microblog</title>
        {% endif %}
    </head>
    <body>
```

```
        <div>
            Microblog:
            <a href="/index">Home</a>
            <a href="/login">Login</a>
        </div>
        <hr>
        {% with messages = get_flashed_messages() %}
        {% if messages %}
        <ul>
            {% for message in messages %}
            <li>{{ message }}</li>
            {% endfor %}
        </ul>
        {% endif %}
        {% endwith %}
        {% block content %}{% endblock %}
    </body>
</html>
```

Тут я використовую конструкцію `with`, щоб призначити результат виклику `get_flashed_messages()` змінній `messages`, все в контексті шаблону. Функція `get_flashed_messages()` поступає з Flask і повертає список всіх повідомлень, які були зареєстровані раніше в `flash()`. Умова, яка слідує, перевіряє, чи має повідомлення якийсь вміст, і у цьому випадку елемент `<ul>` відображається з кожним повідомленням у вигляді пункту списку `<li>`. Цей стиль рендерингу виглядає не дуже добре, але тема стилізації веб-додатка з'явиться пізніше.

Цікавою властивістю цих flash-повідомлень є те, що після їх запиту один раз через функцію `get_flashed_messages()` вони видаляються зі списку повідомлень, тому вони з'являються тільки один раз після виклику функції `flash()`.

Тепер час спробувати додаток знову і перевірити, як працює форма. Переконайтеся, що ви намагаєтеся надіслати форму з пустими полями для імені користувача або пароля, щоб побачити, як валідатор `DataRequired` зупиняє процес надсилання.

Підвищення ефективності перевірки полів

Валідатори, прикріплені до полів форми, запобігають передачу недопустимих даних у додаток. Спосіб, яким додаток реагує на недопустимий ввід в поля форми, полягає в повторному відображенні форми, щоб дозволити користувачеві внести необхідні виправлення.

Якщо ви намагалися ввести недопустимі дані, я впевнений, що ви помітили, що, хоча механізми перевірки працюють добре, немає повідомлень користувачеві про те, що щось не так з формою, користувач просто отримує форму назад. Наступна задача - покращити взаємодію з користувачем, додавши значуще повідомлення про помилку поруч з кожним полем, яке не вдалося перевірити.

Фактично, валідатори форм вже створюють ці описові повідомлення про помилки, тому все, що потрібно, - це додати деяку додаткову логіку в шаблон для їх відображення.

Ось шаблон входу з доданими повідомленнями (Це поле обов'язкове.) для перевірки імені користувача та пароля:

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

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post"  novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
 {% endblock %}
```

Єдине змінення, яке я зробив, - це додав цикли відразу після поля username та password, які відображають повідомлення про помилки, додані валідаторами, у червоному кольорі. Зазвичай всі поля, які мають прикріплені валідатори, матимуть повідомлення про помилки, які додаються в `form.<field_name>.errors`. Це буде список, оскільки поля можуть мати кілька валідаторів, і може бути більше одного повідомлення про помилку для відображення користувачеві.

Якщо ви намагаєтеся надіслати форму з порожнім ім'ям користувача або паролем, ви отримаєте червоне повідомлення про помилку.

Створення зв'язків

Форма входу в систему тепер майже завершена, але перед завершенням цього розділу я хотів би обговорити правильний спосіб включення посилань в шаблони та перенаправлень. До цього моменту ви бачили декілька прикладів, де були визначені посилання. Наприклад, це поточна панель навігації в базовому шаблоні:

```
<div>
    Microblog:
    <a href="/index">Home</a>
    <a href="/login">Login</a>
</div>
```

Функція представлення login також визначає посилання, яке передається функції redirect():

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # ...
        return redirect('/index')
        # ...
```

Одна з проблем з написанням посилань безпосередньо в шаблонах і вихідних файлах полягає в тому, що якщо в один прекрасний день ви вирішите перегрупувати свої посилання, вам доведеться шукати й замінювати ці посилання у всьому додатку.

Щоб краще контролювати ці посилання, Flask надає функцію `url_for()`, яка генерує URL-адреси, використовуючи внутрішнє відображення URL-адрес для функцій представлення. Наприклад, `url_for('login')` повертає `/login`, а `url_for('index')` повертає `/index`. Аргументом для `url

_for()` є ім'я кінцевої точки, яке є ім'ям функції представлення.

Ви можете запитати, чому краще використовувати імена функцій замість URL-адрес. Справа в тому, що URL-адреси набагато частіше змінюються, ніж імена функцій, які

є внутрішніми. Друга причина полягає в тому, що, як ви згодом дізнаєтеся, деякі URL-адреси мають динамічні компоненти, тому для генерації цих URL вручну потрібно об'єднати кілька елементів, що є втомлюючим і схильним до помилок. `url_for()` також може генерувати ці складні URL-адреси.

Отже, з цього моменту я буду використовувати `url_for()` кожного разу, коли мені потрібно створити URL додатка. У результаті панель навігації в базовому шаблоні виглядає так:

```
<div>
    Microblog:
    <a href="{{ url_for('index') }}">Home</a>
    <a href="{{ url_for('login') }}">Login</a>
</div>
```

І ось оновлена функція `login()`:

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

# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('index'))
# ...
```

URL Building (Побудова URL)

URL Building - це процес створення URL-адреси на основі деяких параметрів чи даних. В Flask цей процес вирішується за допомогою функції `url_for()`. Вона дозволяє генерувати URL-адреси на основі імені функції виду (view function) та аргументів, що передаються цій функції.

Наприклад, якщо у вас є вид з ім'ям `user_profile`, який очікує аргумент `username`, ви можете побудувати його URL-адресу так:

```
from flask import Flask, url_for

app = Flask(__name__)
```

```
@app.route('/user/<username>')
def user_profile(username):
    # Опрацьовання профілю користувача
    pass

with app.test_request_context():
    url = url_for('user_profile', username='john_doe')
    print(url)
```

У цьому прикладі `url_for('user_profile', username='john_doe')` генерує URL-адресу для виду `user_profile` з аргументом `username` встановленим на `'john_doe'`. Результат буде `/user/john_doe`.

Цей підхід до побудови URL-адрес важливий, оскільки він дозволяє уникнути прямого вказування URL-адрес у коді. Замість цього ви використовуєте імена видів та аргументи, що робить ваш код більш зрозумілим та обчислюваним. Також, якщо ви змініте URL-адресу в майбутньому, вам не потрібно буде змінювати її вручну в усіх місцях, де вона використовується. Вам просто доведеться змінити її в одному місці - в функції маршруту (route function).

Отже, URL Building в Flask є потужним інструментом, який полегшує роботу з URL-адресами у вашому додатку.