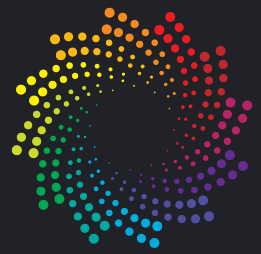


Flask: Templates

Andrea Pescetti
andrea@datamira.com

Luglio 2024

Scopo del modulo

Presentare
l'uso di template di pagina
e la struttura dei progetti

Argomenti

- Struttura di un progetto
- Template di pagina
- Usare template per i form

Struttura
di un progetto Flask

Applicazioni complesse

- Quando le applicazioni raggiungono una complessità superiore rispetto alle applicazioni pensate solo per test, tenere tutto in un unico file non è più possibile.
- Flask non impone particolari regole su come suddividere un'applicazione in file e cartelle.
- Tuttavia ha dei default che rendono immediato strutturare un'applicazione secondo certi schemi, mentre altri schemi richiedono adattamenti manuali.

Schema tipico di una applicazione

```
progetto/
│
├── app/
│   ├── __init__.py
│   ├── routes.py
│   ├── models.py
│   └── templates/
│       └── base.html
│
├── config.py
├── run.py
└── requirements.txt
```

La cartella principale

Nella cartella principale trovano posto questi tre file:

- `requirements.txt` : Serve per ricreare l'ambiente quando distribuiamo l'applicazione

```
pip freeze > requirements.txt      # Per generarlo
pip install -r requirements.txt    # Per ricreare l'ambiente altrove
```

- `run.py` : Il punto di ingresso dell'applicazione, che crea il server.
- `config.py` : Usato per differenziare tra configurazioni (di sviluppo, staging, produzione...) che ad esempio usano credenziali diverse per i servizi web. Non lo useremo perché lavoriamo solo in sviluppo.

Il file `run.py`

- Versione minimale senza supporto per configurazioni multiple:

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

- Con supporto per configurazioni multiple (possibile esempio):

```
from app import create_app
import os
config_name = os.getenv('FLASK_CONFIG') or 'default'
app = create_app(config_name)
if __name__ == '__main__':
  app.run()
```

La cartella `app/`

La cartella `app/` contiene:

- Un file `__init.py__` che, come si può capire dalla convenzione *dunder*, è un nome speciale di Python. La presenza di `__init__.py` segnala a Python che la directory dovrebbe essere trattata come un modulo; il file contiene codice di inizializzazione per il modulo.
- Un file `routes.py` che contiene le definizioni delle routes, con decorator e funzione che genera il contenuto.
- Un file `models.py` che useremo nel seguito (quando introdurremo Flask-SQLAlchemy per la gestione database) e che definisce i modelli di dati: le strutture delle tabelle del database e le relazioni tra di esse.

Il file `app/__init.py__`

- Una prima versione minimale, senza supporto per configurazioni multiple, e poi una versione più strutturata:

```
from flask import Flask
def create_app():
  app = Flask(__name__)
  return app
```

```
from flask import Flask
from config import config
def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    # Inizializzazione delle estensioni
    # from .extensions import ...
    return app
```

La cartella `app/templates`

- La cartella `app/templates` contiene i template di pagina.
- Anche se i nomi di file sono per convenzione `.html`, quindi ad esempio `base.html`, i file non sono al 100% file HTML. Infatti i template sono file HTML che vengono processati da Python e che usano una sintassi estesa, nota come Jinja2.
- Notiamo tuttavia che Flask non impone la scelta di Jinja2; è perfettamente possibile (ma richiede più lavoro) usare un motore di template a scelta.

Template di pagina

Jinja2

- Jinja2 è un motore di template per Python, utilizzato principalmente per generare markup HTML o altri formati di testo.
- È stato sviluppato da Armin Ronacher, il creatore di Flask, ed è il motore di template predefinito di Flask, ma è usato anche da altri programmi Python.
- Si contraddistingue per semplicità (facilita la generazione di contenuti dinamici), sicurezza (protegge contro le vulnerabilità comuni come l'injection), flessibilità (supporta l'uso di variabili, cicli, condizioni, inclusioni).

Cos'è un template

- Un template è una pagina (o parte di pagina) già preimpostata e pensata per essere riutilizzabile.
- Oltre al codice HTML classico, sono presenti dei segnaposti (placeholder, da non confondere però con l'attributo HTML `placeholder`) che vengono sostituiti a runtime.
- Inoltre i template possono contenere logica (cicli, condizionali, funzioni...) e dei blocchi che hanno un nome identificativo.

Chiamare un template

- Collocare il file HTML (all'inizio può essere un file HTML senza sintassi estesa) nella cartella `templates/` .
- Modificare la route come segue:

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

- Questo già permette di usare template statici. Tipicamente si passano come argomenti le variabili, o `vars()` che passa al template tutte le variabili locali per ulteriore elaborazione:

```
  return render_template('index.html', nome='Andrea', cognome='Pescetti')
  return render_template('index.html', vars_dict=vars())
```

Esempio di template

```
<!DOCTYPE html>
<html lang="it">
<head>
  <meta charset="UTF-8">
  <title>{{ title }}</title>
</head>
<body>
  <h3>{{ content }}</h3>
  {% block maincontent %}
  {% endblock %}
</body>
</html>
```

Delimitatori speciali

I template oltre al codice HTML standard ammettono, tra l'altro:

- `{% ... %}` : Statements (istruzioni come cicli e condizionali)
- `{{ ... }}` : Espressioni da stampare, come le variabili
- `{# ... #}` : Commenti che vengono ignorati e non inseriti nell'output

Variabili ed espressioni

- Servono a stampare un valore. Le parentesi indicano l'istruzione di stampa, non l'accesso ai valori.
- Per sicurezza, le variabili non vengono stampate esattamente come sono state passate, ma viene applicato un filtro di escape automaticamente, per evitare injection di codice HTML.

```
{{ nome }}
{{ 'Dott. ' + nome }}
{{ persona.nome }}          {# Due modi equivalenti  #}
{{ persona['nome'] }}       {# tra cui scegliere     #}
```

Filtri

- Trasformano un'espressione prima di stamparla. Si concatenano:

```
{{ nome|striptags|title }}
```

- Filtri per modificare maiuscole/minuscole:

```
nome|capitalize    nome|title    nome|lower    nome|upper
```

- Filtri per ripulire o non ripulire (attenzione!) HTML

```
nome|escape    nome|e    nome|safe
```

- Filtri per togliere tag HTML o spazi:

```
nome|striptags    nome|trim
```

Condizionali

- Testano se una variabile esiste o non esiste (`if variabile`)
- Oppure hanno a disposizione una sintassi di condizioni simili a quella di Python (con `==` e gli altri operatori)

```
{% if prodotti %}
  <p>Ho dei prodotti</p>
{% else %}
  <p>Non ho prodotti</p>
{% endif %}
```

Cicli for

- Iterano su una variabile iterabile (lista, dizionario e simili) con una sintassi molto simile a `for` di Python:

```
{% if utenti %}
<ul>
{% for utente in utenti %}
    <li>{{ utente.mail|e }}</li>
{% endfor %}
</ul>
{% endif %}
```

Estensione

- I template possono ereditare l'uno dall'altro.
- I blocchi sono segnaposti inseriti nel template padre che non vengono mostrati se non valorizzati.
- Per valorizzarli, nel template figlio forniamo solo il blocco che è cambiato rispetto al template padre, che è indicato come argomento di `extends` .

```
{% extends "home.html" %}
{% block maincontent %}
<img src="logo.png" alt="logo">
<p>Questo è il blocco con il
contenuto principale della pagina.</p>
{% endblock %}
```

Inclusione

- L'inclusione si usa per importare porzioni di pagina che sono comuni a molte pagine, ad esempio header o footer.

```
{% extends "home.html" %}
{% block maincontent %}
{% include "header.html" %}
<p>Questo è il blocco con il
contenuto principale della pagina.</p>
{% endblock %}
```

Usare i form in Flask

Struttura dei form

- Per form semplici in Flask si può usare il sistema di template standard.
- Quindi un form sarà semplicemente un template specifico, con i vari campi e con un'azione POST verso un'altra (o spesso la stessa) route, che in caso di GET mostrerà il form vuoto e in caso di POST lo elaborerà.
- Esistono estensioni apposite per Flask, ed in particolare Flask-WTF, che offrono funzioni avanzate utili in contesti di produzione, come protezione con CSRF (che impedisce di inviare un form se non si è effettivamente su una pagina del sito), supporto per CAPTCHA e upload di file semplificato.

Gestire richieste POST

- I parametri passati in POST sono visibili all'interno dell'oggetto `request` ed accessibili attraverso `request`:

```
@app.route('/test', methods=['POST'])
def parametri():
  print(request)
  print("GET", request.args)
  print("POST", request.form)
```

Gestire GET e POST insieme

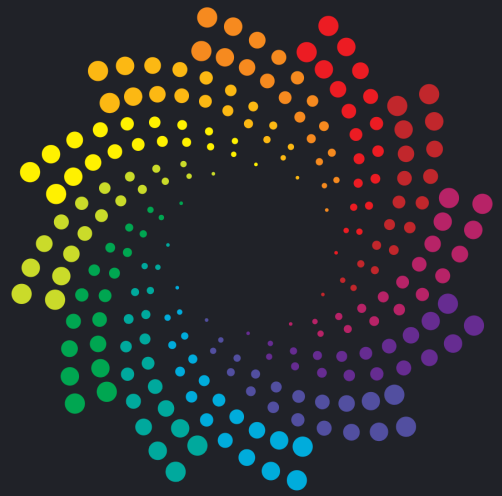
```
@app.route('/login', methods=['GET', 'POST'])
def login():
  if request.method == 'POST':
    username = request.form['username']
    password = request.form['password']
  if username == 'admin' and password == 'abc123':
    # Login riuscito
    return redirect('/dashboard')
  else:
    # Login non riuscito
    abort(401)
  return render_template('login.html')
```

🥷 Una macro utile per i form

```
{% macro input(name, value='', type='text', size=20) -%}
    <input type="{{ type }}" name="{{ name }}" value="{{
        value|e }}" size="{{ size }}">
{%- endmacro %}
```

La macro può essere chiamata come un'ordinaria funzione:

```
<p>{{ input('username') }}</p>
<p>{{ input('password', type='password') }}</p>
```



DATAMIRA

Andrea Pescetti

andrea@datamira.com

(o via LinkedIn)