BD 2 - Flask

Luca Cosmo

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Flask

Flask è un framework minimale per lo sviluppo di applicazioni web:

- basato sul linguaggio di programmazione Python, che diventa il linguaggio di sviluppo del Logic Tier della web application
- sistema di template HTML, popolati dinamicamente da Python usando informazione della richiesta HTTP o salvata lato server
- una valida alternativa ad ASP, JSP, PHP ed analoghi

Installabile tramite il gestore di pacchetti Python: `pip install flask`

Hello, World!

Contenuto del file `hello.py`:

```
from flask import Flask
app = Flask(__name__)

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

Una route consente di associare una funzione Python (view) ad un path specifico sul server, in questo caso la radice.

Hello, World! - Dietro le Quinte

Il paradigma di Flask si basa sull'uso di decoratori Python:

```
def decor(func):
    def wrapper():
        print("Before call")
        func()
        print("After call")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = decor(say_whee)
say_whee()
```

```
def decor(func):
    def wrapper():
        print("Before call")
        func()
        print("After call")
    return wrapper

@decor
def say_whee():
    print("Whee!")

say_whee()
```

Esecuzione dell'Applicazione

Sotto Linux, potete eseguire il comando per avviare l'applicazione:

```
flask --app hello run
```

Il risultato è l'esecuzione di un web server locale sulla porta 5000. Tale server non è production-ready, vi viene solo fornito per convenienza!

Navigate ora il vostro browser all'indirizzo:

```
http://127.0.0.1:5000/
```

Il browser vi saluterà con il risultato dell'esecuzione della view associata alla route precedente.

Debug Mode

E' possibile attivare la modalità di debug di Flask come segue:

```
flask --app hello run --debug
```

Molto utile in fase di sviluppo:

1. se modificate il contenuto di un file, esso verrà subito ricaricato senza che dobbiate chiudere e rilanciare la vostra applicazione
2. potete visualizzare eventuali messaggi di errore direttamente dalla schermata del vostro browser
3. potete eseguire codice nella vostra applicazione a fini di debug direttamente dal browser (richiede l'inserimento di un PIN)

Route con Variabili

Aggiungiamo ora una seconda route alla nostra applicazione. In questo caso facciamo anche uso di una variabile:

```
@app.route('/users/<username>')
def show_profile(username):
    return 'Welcome to the web app, %s!' % username
```

In questo modo abbiamo sostanzialmente definito una serie arbitraria di pagine, una per ciascun utente della nostra applicazione: una prima forma potente di generazione dinamica dei contenuti HTML.

Route con Variabili

Dato che una route può essere associata ad una funzione arbitraria, si possono anche fare ulteriori controlli. Per esempio distinguere fra utenti anonimi ed utenti registrati:

```
@app.route('/users/<username>')
def show_profile(username):
    users = ['alice', 'bob', 'charlie']
    if username in users:
        return 'Welcome to the web app, %s!' % username
    else:
        return 'Welcome to the web app, anonymous!'
```

Variabili tipate

Possiamo filtrare le variabili anche per tipo, ovvero una route può rispondere a valori numerici ed un'altra a valori stringa.

```
@app.route('/users/<string:username>')
def show_profile(username):
    return f'Welcome {username}!'

@app.route('/users/<int:userid>')
def show_profile(userid):
    return f'Welcome user number {userid}!'
```

Altri tipi sono: int, float, string (default), path, uuid

Template

E' chiaro che le soluzioni viste finora scalano male nel momento in cui vogliamo progettare una pagina HTML complessa.

Flask ci viene in aiuto tramite un sistema di template ereditato da Jinja:

- un template non è altro che un file HTML esteso con una sintassi speciale che ci consente di passare parametri da Flask all'HTML
- i parametri permettono di generare HTML dinamicamente, per esempio sulla base dell'input dell'utente o di una computazione
- questo approccio supporta il disaccoppiamento della logica della web application dalla struttura della pagina HTML visualizzata

Template

Contenuto di `templates/index.html`:

```
<html>
<head>
    <title>My fancy web app</title>
</head>
<body>
    Hello world! <br/>
    Click <a href='/users/anonymous'/>here</a> to continue.
</body>
</html>
```

Aggiornamento della route corrispondente:

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

Template: Passaggio di Parametri

Contenuto di `templates/profile.html`:

```
<html>
<head>
    <title>My fancy web app</title>
</head>
<body>
    {% if user %}
    Welcome to the web app, {{ user }}!
    {% else %}
    Welcome to the web app, anonymous!
    {% endif %}
</body>
</html>
```

Template: Passaggio di Parametri

Aggiornamento della route corrispondente:

```
@app.route('/users/<username>')
def show_profile(username):
    users = ['alice', 'bob', 'charlie']
    if username in users:
        return render_template('profile.html',
                               user=username)
    else:
        return render_template('profile.html')
```

Risorse statiche

Non tutte le richieste al server web richiedono un'elaborazione, certe risorse sono semplici file 'statici' (e.g. immagini, file css e javascript). Flask consentirà l'accesso dirtto a tutti i file contenuti nella cartella speciale denominata "static".

```
<html>
...
<body>
    {% if user %}
    <p>Welcome to the web app, {{ user }}!<p>
    <img src="/static/{{user}}.jpg" />
    {% else %}
    Welcome to the web app, anonymous!
    {% endif %}
</body>
</html>
```

Template: Loop

Estendiamo `profile.html` in modo che visualizzi la lista degli utenti registrati quando un accesso è anonimo:

```
<html>
<head>
    <title>My fancy web app</title>
</head>
<body>
    {% if user %}
    Welcome to the web app, {{ user }}!
    {% else %}
    Welcome to the web app, anonymous! <br/>
    The following users are registered:
    <ul>
    {% for u in reg %}
        <li> {{ u }} </li>
    {% endfor %}
    </ul>
    {% endif %}
</body>
</html>
```

Template: Loop

Aggiornamento della route corrispondente:

```
@app.route('/users/<username>')
def show_profile(username):
    users = ['alice', 'bob', 'charlie']
    if username in users:
        return render_template('profile.html',
                               user=username)
    else:
        return render_template('profile.html',
                               reg=users)
```

Ereditare Template

Nel nostro HTML c'è una certa ridondanza, in particolare la sezione di testa è in comune fra le due pagine. Possiamo estrapolare questa parte comune in un template `base.html` con un blocco da riempire:

```
<html>
<head>
    <title>My fancy web app</title>
</head>
<body>
    {% block pagebody %}
    {% endblock %}
</body>
</html>
```

Ereditare Template

Aggiornamento di `index.html`:

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

{% block pagebody %}
    Hello world! <br/>
    Click <a href='/users/anonymous'/>here</a> to continue.
{% endblock %}
```

Possiamo aggiornare `profile.html` in modo analogo.

Form

Ci rimane un problema da affrontare: il link di accesso all'applicazione ci fa entrare come utente anonimo. Possiamo perciò inserire nella pagina un form dove inserire il nome utente con cui vogliamo accedere:

```
Insert your username:
<form action="/login" method="POST">
    <input name="user" type="text">
    <input type="submit">
</form>
```

Per gestire la sottomissione del form ci serve un'altra route, questa volta pronta a gestire il metodo POST.

Processing del Form

```
@app.route('/login', methods=['GET','POST'])
def login():
    user = request.form["user"]
    return redirect(url_for('show_profile',
                            username=user))
```

La nuova route è molto corta, ma offre vari spunti di discussione:

- l'unico metodo accettato di default da Flask è GET, ma abbiamo modo di personalizzare questo comportamento
- l'oggetto `request` contiene varie informazioni sulla richiesta HTTP
- utilizzo combinato delle funzioni `redirect` ed `url_for`

Esempio: `template.py`

Blueprint

Quando una web application cresce di dimensione e complessità, è utile strutturarla in moduli indipendenti detti blueprint

```
app/
|
|--  app.py
```

```
app/
|
|--  app.py
|--  login.py
```

In questa nuova struttura:

- `login.py` definisce un nuovo blueprint `login_bp`
- `app.py` registra `login_bp` come parte dell'applicazione

Esempio: `blueprint`

Blueprint

Un blueprint è simile ad una sotto-applicazione Flask:

- può avere i propri template: il parametro `template_folder` può essere utilizzato per impostare la directory dei template
- può avere un proprio path: se `url_prefix` è settato a `/auth`, allora una richiesta per `/auth/` risolverà alla route `/` del blueprint
- può avere i propri file statici (immagini, stylesheets, etc.)

Approfondimento: `https://realpython.com/flask-blueprint/`

Materiale Didattico

Queste slide forniscono un punto di partenza ragionevole per approfondire Flask, ma ci sono varie risorse utili sull'argomento:

- Tutorial ufficiale di Flask (link)
- The Flask Mega-Tutorial (link)
- Miguel Grinberg - Flask Web Development: Developing Web Applications with Python (disponibile in biblioteca)

La documentazione è utile anche perchè menziona molte estensioni per Flask, ne vedremo alcune nella prossima lezione.

Checkpoint!

Provate ora a ricostruire l'applicazione che abbiamo discusso in queste slides sulla vostra macchina! In particolare, dovreste avere:

- un'applicazione Flask con 3 route: una per la pagina di benvenuto, una per il login ed una per la pagina del profilo
- una directory con 3 template: uno per la pagina di benvenuto, uno per la pagina del profilo ed una base con la struttura condivisa
- opzionale: strutturate l'applicazione in blueprint, separando l'area riservata (login + profilo) dalla pagina di benvenuto

Avviate la vostra applicazione in modalità debug e giocateci per prendere familiarità con Flask!