EBook Gratuito

APPRENDIMENTO

Flask

Free unaffiliated eBook created from
Stack Overflow contributors.

#flask

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: flask

It is an unofficial and free Flask ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Flask.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con Flask

Osservazioni

Flask è un micro-framework per applicazioni web Python costruito sopra la libreria WSGI di Werkzeug . Flask può essere "micro", ma è pronto per l'uso in produzione su una varietà di esigenze.

Il "micro" in micro-framework significa che Flask mira a mantenere il core semplice ma estendibile. Flask non prenderà molte decisioni per te, ad esempio quale database usare e le decisioni che prenderà sono facili da cambiare. Tutto dipende da te, in modo che Flask possa essere tutto ciò di cui hai bisogno e niente che non sia.

La community supporta un ricco ecosistema di estensioni per rendere la tua applicazione più potente e ancora più facile da sviluppare. Man mano che il progetto cresce, sei libero di prendere le decisioni di progettazione appropriate alle tue esigenze.

Versioni

Versione	Nome in codice	Data di rilascio
0,12	Punsch	2016/12/21
0,11	Assenzio	2016/05/29
0.10	Limoncello	2013/06/13

Examples

Installazione - Stabile

Usa pip per installare Flask in un virtualenv.

```
pip install flask
```

Istruzioni passo passo per la creazione di un virtualenv per il tuo progetto:

```
mkdir project && cd project
python3 -m venv env
# or `virtualenv env` for Python 2
source env/bin/activate
pip install flask
```

Non usare **mai** `sudo pip install` meno che tu non capisca esattamente cosa stai facendo.

Mantenere il progetto in una virtualenv locale, non installare sul sistema Python a meno che non si stia utilizzando il gestore di pacchetti di sistema.

Ciao mondo

Crea `hello.py` :

```
from flask import Flask

app = Flask(__name__)


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

Quindi eseguilo con:

```
export FLASK_APP=hello.py
flask run
 * Running on http://localhost:5000/
```

Aggiungere il codice qui sotto permetterà di eseguirlo direttamente con `python hello.py` .

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

Installazione - Più recenti

Se si desidera utilizzare il codice più recente, è possibile installarlo dal repository. Sebbene tu abbia potenzialmente nuove funzionalità e correzioni, solo le versioni numerate sono ufficialmente supportate.

```
pip install https://github.com/pallets/flask/tarball/master
```

Installazione - Sviluppo

Se si desidera sviluppare e contribuire al progetto Flask, clonare il repository e installare il codice in modalità di sviluppo.

```
git clone ssh://github.com/pallets/flask
cd flask
python3 -m venv env
source env/bin/activate
pip install -e .
```

Ci sono alcune dipendenze e strumenti extra da prendere in considerazione.

sfinge

Utilizzato per costruire la documentazione.

```
pip install sphinx
cd docs
make html
firefox _build/html/index.html
```

py.test

Utilizzato per eseguire la suite di test.

```
pip install pytest
py.test tests
```

tox

Utilizzato per eseguire la suite di test contro più versioni di Python.

```
pip install tox
tox
```

Nota che tox utilizza solo interpreti che sono già installati, quindi se Python 3.3 non è installato sul tuo percorso, non verrà testato.

Leggi Iniziare con Flask online: https://riptutorial.com/it/flask/topic/790/iniziare-con-flask

Capitolo 2: Accesso ai dati della richiesta

introduzione

Quando si lavora con un'applicazione Web, talvolta è importante accedere ai dati inclusi nella richiesta, oltre l'URL.

In Flask questo è memorizzato sotto l'oggetto di **richiesta** globale, a cui è possibile accedere nel codice tramite la `from flask import request` .

Examples

Accesso alla stringa di query

La stringa di query è la parte di una richiesta che segue l'URL, preceduta da un `?` marchio.

Esempio: `https://encrypted.google.com/search` **`?hl=en&q=stack%20overflow`**

Per questo esempio, stiamo creando un semplice web server echo che rispetta tutto ciò che è stato inviato tramite il campo `echo` nelle richieste `GET` .

Esempio: `localhost:5000/echo` **`?echo=echo+this+back+to+me`**

Esempio di bottiglia :

```
from flask import Flask, request

app = Flask(import_name=__name__)

@app.route("/echo")
def echo():

    to_echo = request.args.get("echo", "")
    response = "{}".format(to_echo)

    return response

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

Forma combinata e stringa di query

Flask consente anche l'accesso a un CombinedMultiDict che dà accesso ad entrambi gli attributi `request.form` e `request.args` sotto una variabile.

Questo esempio recupera i dati dal `name` un campo modulo inviato insieme al campo `echo` nella stringa di query.

Esempio di bottiglia :

```
from flask import Flask, request

app = Flask(import_name=__name__)


@app.route("/echo", methods=["POST"])
def echo():

    name = request.values.get("name", "")
    to_echo = request.values.get("echo", "")

    response = "Hey there {}! You said {}".format(name, to_echo)

    return response

app.run()
```

Accesso ai campi del modulo

È possibile accedere ai dati del modulo inviati tramite una richiesta `POST` o `PUT` in Flask tramite l'attributo `request.form` .

```
from flask import Flask, request

app = Flask(import_name=__name__)


@app.route("/echo", methods=["POST"])
def echo():

    name = request.form.get("name", "")
    age = request.form.get("age", "")

    response = "Hey there {}! You said you are {} years old.".format(name, age)

    return response

app.run()
```

Leggi Accesso ai dati della richiesta online: https://riptutorial.com/it/flask/topic/8622/accesso-ai-dati-della-richiesta

Capitolo 3: analisi

Examples

Test della nostra app Hello World

introduzione

In questo esempio minimalista, usando `pytest` , `pytest` che la nostra app Hello World restituisce "Hello, World!" con un codice di stato HTTP OK di 200, quando viene colpito con una richiesta GET sull'URL `/`

Per prima cosa installiamo `pytest` nel nostro virtualenv

```
pip install pytest
```

E solo per riferimento, questa nostra app ciao mondo:

```
# hello.py
from flask import Flask

app = Flask(__name__)

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

Definire il test

Accanto al nostro `hello.py` , definiamo un modulo di test chiamato `test_hello.py` che verrà scoperto da `py.test`

```
# test_hello.py
from hello import app

def test_hello():
    response = app.test_client().get('/')

    assert response.status_code == 200
    assert response.data == b'Hello, World!'
```

Giusto per valutare, a questo punto la nostra struttura di progetto ottenuta con il comando ad `tree` è:

```
.
├── hello.py
└── test_hello.py
```

Esecuzione del test

Ora possiamo eseguire questo test con il comando `py.test` che scoprirà automaticamente il nostro `test_hello.py` e la funzione di test al suo interno

```
$ py.test
```

Dovresti vedere alcuni risultati e un'indicazione che è passato 1 test, ad es

```
=== test session starts ===
collected 1 items
test_hello.py .
=== 1 passed in 0.13 seconds ===
```

Test di un'API JSON implementata in Flask

Questo esempio presume che tu sappia come testare un'app Flask usando pytest

Di seguito è riportata un'API che accetta un input JSON con valori interi `a` e `b` eg `{"a": 1, "b": 2}`, li somma e restituisce sum `a + b` in una risposta JSON, ad esempio `{"sum": 3}`.

```
# hello_add.py
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add', methods=['POST'])
def add():
    data = request.get_json()
    return jsonify({'sum': data['a'] + data['b']})
```

Testare questa API con `pytest`

Possiamo testarlo con `pytest`

```
# test_hello_add.py
from hello_add import app
from flask import json

def test_add():
    response = app.test_client().post(
        '/add',
        data=json.dumps({'a': 1, 'b': 2}),
        content_type='application/json',
    )

    data = json.loads(response.get_data(as_text=True))

    assert response.status_code == 200
    assert data['sum'] == 3
```

Ora esegui il test con il comando `py.test` .

Accesso e manipolazione delle variabili di sessione nei test mediante Flask-Testing

La maggior parte delle applicazioni Web utilizza l'oggetto sessione per memorizzare alcune informazioni importanti. Questo esempio mostra come testare tale applicazione utilizzando Flask-Testing. L'esempio operativo completo è disponibile anche su github .

Quindi, prima installa Flask-Testing nel tuo virtualenv

```
pip install flask_testing
```

Per poter utilizzare l'oggetto sessione è necessario impostare la chiave segreta

```
app.secret_key = 'my-seCret_KEy'
```

Immaginiamo di avere nella funzione dell'applicazione che è necessario memorizzare alcuni dati in variabili di sessione come questa

```
@app.route('/getSessionVar', methods=['GET', 'POST'])
def getSessionVariable():
  if 'GET' == request.method:
    session['sessionVar'] = 'hello'
  elif 'POST' == request.method:
    session['sessionVar'] = 'hi'
  else:
    session['sessionVar'] = 'error'

  return 'ok'
```

Per testare questa funzione è possibile importare flask_testing e lasciare che la classe di test erediti flask_testing.TestCase. Importa anche tutte le librerie necessarie

```
import flask
import unittest
import flask_testing
from myapp.run import app

class TestMyApp(flask_testing.TestCase):
```

Molto importante prima di iniziare il test è implementare la funzione **create_app** altrimenti ci sarà un'eccezione.

```
  def create_app(self):
    return app
```

Per testare la tua applicazione funziona come volevi hai un paio di possibilità. Se vuoi solo assicurarti che la tua funzione stia impostando valori particolari su una variabile di sessione puoi semplicemente mantenere il contesto intorno e accedere a **flask.session**

```
def testSession1(self):
    with app.test_client() as lTestClient:
      lResp= lTestClient.get('/getSessionVar')
      self.assertEqual(lResp.status_code, 200)
      self.assertEqual(flask.session['sessionVar'], 'hello')
```

Un altro trucco utile è quello di differenziare tra i metodi *GET* e *POST* come nella prossima funzione di test

```
def testSession2(self):
    with app.test_client() as lTestClient:
      lResp= lTestClient.post('/getSessionVar')
      self.assertEqual(lResp.status_code, 200)
      self.assertEqual(flask.session['sessionVar'], 'hi')
```

Ora immagina che la tua funzione si aspetti che una variabile di sessione sia impostata e reagisca in modo diverso su valori particolari come questo

```
@app.route('/changeSessionVar')
def changeSessionVariable():
  if session['existingSessionVar'] != 'hello':
    raise Exception('unexpected session value of existingSessionVar!')

  session['existingSessionVar'] = 'hello world'
  return 'ok'
```

Per testare questa funzione è necessario utilizzare la cosiddetta *transazione di sessione* e aprire la sessione nel contesto del client di test. Questa funzione è disponibile dal **Flask 0.8**

```
def testSession3(self):
    with app.test_client() as lTestClient:
      #keep the session
      with lTestClient.session_transaction() as lSess:
        lSess['existingSessionVar'] = 'hello'

      #here the session is stored
      lResp = lTestClient.get('/changeSessionVar')
      self.assertEqual(lResp.status_code, 200)
      self.assertEqual(flask.session['existingSessionVar'], 'hello world')
```

Eseguire i test è come al solito per unittest

```
if __name__ == "__main__":
    unittest.main()
```

E nella riga di comando

```
python tests/test_myapp.py
```

Un altro bel modo per eseguire i test consiste nell'utilizzare l'Unittest Discovery in questo modo:

```
python -m unittest discover -s tests
```

Leggi analisi online: https://riptutorial.com/it/flask/topic/1260/analisi

Capitolo 4: Autorizzazione e autenticazione

Examples

Usando l'estensione login-flask

Uno dei modi più semplici per implementare un sistema di autorizzazione consiste nell'utilizzare l'estensione login-flask . Il sito web del progetto contiene un quickstart dettagliato e ben scritto, di cui una versione più breve è disponibile in questo esempio.

Idea generale

L'estensione espone un insieme di funzioni utilizzate per:

- registrazione utenti in
- disconnettendo gli utenti
- verificare se un utente è connesso o meno e scoprire quale utente è quello

Cosa non fa e cosa devi fare da solo:

- non fornisce un modo di memorizzare gli utenti, ad esempio nel database
- non fornisce un modo per controllare le credenziali dell'utente, ad esempio nome utente e password

Di seguito c'è una serie minima di passaggi necessari per far funzionare tutto.

Raccomanderei di inserire tutto il codice relativo `auth.py` in un modulo o pacchetto separato, ad esempio `auth.py` In questo modo è possibile creare separatamente classi, oggetti o funzioni personalizzate.

Creare un `LoginManager`

L'estensione utilizza una classe `LoginManager` che deve essere registrata sull'oggetto dell'applicazione `Flask` .

```
from flask_login import LoginManager
login_manager = LoginManager()
login_manager.init_app(app) # app is a Flask object
```

Come accennato in precedenza, `LoginManager` può essere ad esempio una variabile globale in un file o pacchetto separato. Quindi può essere importato nel file in cui è stato creato l'oggetto `Flask` o nella funzione di fabbrica dell'applicazione e inizializzato.

Specifica una richiamata utilizzata per il caricamento degli utenti

Un utente verrà normalmente caricato da un database. Il callback deve restituire un oggetto che rappresenta un utente corrispondente all'ID fornito. Dovrebbe restituire `None` se l'ID non è valido.

```
@login_manager.user_loader
def load_user(user_id):
    return User.get(user_id) # Fetch the user from the database
```

Questo può essere fatto direttamente sotto la creazione del tuo `LoginManager` .

Una classe che rappresenta il tuo utente

Come accennato, il callback `user_loader` deve restituire un oggetto che rappresenta un utente. Cosa significa esattamente? Ad esempio, tale oggetto può essere un wrapper attorno agli oggetti utente memorizzati nel database o semplicemente direttamente un modello dal database. Questo oggetto deve implementare i seguenti metodi e proprietà. Ciò significa che se il callback restituisce il modello del database, è necessario assicurarsi che le proprietà e i metodi citati vengano aggiunti al modello.

- `is_authenticated`

  Questa proprietà dovrebbe restituire `True` se l'utente è autenticato, cioè hanno fornito credenziali valide. Dovrai assicurarti che gli oggetti che rappresentano i tuoi utenti restituiti dal callback `user_loader` restituiscano `True` per quel metodo.

- `is_active`

  Questa proprietà dovrebbe restituire True se questo è un utente attivo - oltre ad essere autenticato, hanno anche attivato il loro account, non sono stati sospesi, o qualsiasi condizione che l'applicazione ha per il rifiuto di un account. Gli account inattivi potrebbero non accedere. Se non si dispone di un meccanismo presente, restituire `True` da questo metodo.

- `is_anonymous`

  Questa proprietà dovrebbe restituire True se si tratta di un utente anonimo. Ciò significa che l'oggetto utente restituito dalla callback `user_loader` deve restituire `True` .

- `get_id()`

  Questo metodo deve restituire un Unicode che identifica in modo univoco questo utente e può essere utilizzato per caricare l'utente dal callback `user_loader` . Nota che questo deve essere un unicode: se l'ID è nativamente un int o qualche altro tipo, dovrai convertirlo in unicode. Se il callback `user_loader` restituisce oggetti dal database, questo metodo restituirà molto probabilmente l'ID del database di questo particolare utente. Lo stesso ID dovrebbe

ovviamente causare il callback `user_loader` per restituire lo stesso utente in un secondo momento.

Se si desidera semplificare le cose da soli (** è in realtà consigliato) è possibile ereditare da `UserMixin` l'oggetto restituito dal callback `user_loader` (presumibilmente un modello di database). Puoi vedere come questi metodi e proprietà sono implementati di default in questo mixin qui .

Registrazione degli utenti in

L'estensione lascia la convalida del nome utente e della password immessi dall'utente all'utente. In effetti l'estensione non interessa se si utilizza una combinazione di nome utente e password o altro meccanismo. Questo è un esempio per la registrazione degli utenti nell'uso di username e password.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    # Here we use a class of some kind to represent and validate our
    # client-side form data. For example, WTForms is a library that will
    # handle this for us, and we use a custom LoginForm to validate.
    form = LoginForm()
    if form.validate_on_submit():
        # Login and validate the user.
        # user should be an instance of your `User` class
        login_user(user)

        flask.flash('Logged in successfully.')

        next = flask.request.args.get('next')
        # is_safe_url should check if the url is safe for redirects.
        # See http://flask.pocoo.org/snippets/62/ for an example.
        if not is_safe_url(next):
            return flask.abort(400)

        return flask.redirect(next or flask.url_for('index'))
    return flask.render_template('login.html', form=form)
```

In generale, gli utenti di logging si ottengono chiamando login_user e passando un'istanza di un oggetto che rappresenta l'utente menzionato in precedenza. Come mostrato, ciò accade di solito dopo aver recuperato l'utente dal database e aver convalidato le sue credenziali, tuttavia l'oggetto utente appare magicamente in questo esempio.

Ho effettuato l'accesso a un utente, e ora?

L'oggetto restituito dal callback `user_loader` è accessibile in più modi.

- Nei modelli:

  L'estensione la inietta automaticamente sotto il nome `current_user` usando un processore di contesto del modello. Per disabilitare tale comportamento e utilizzare il set di processori personalizzato `add_context_processor=False` nel costruttore di `LoginManager` .

```
{% if current_user.is_authenticated %}
  Hi {{ current_user.name }}!
{% endif %}
```

- Nel codice Python:

  L'estensione fornisce un oggetto associato alla richiesta denominato `current_user` .

```
from flask_login import current_user

@app.route("/hello")
def hello():
    # Assuming that there is a name property on your user object
    # returned by the callback
    if current_user.is_authenticated:
        return 'Hello %s!' % current_user.name
    else:
        return 'You are not logged in!'
```

- Limitare l'accesso rapidamente usando un decoratore Un decoratore `login_required` può essere utilizzato per limitare l'accesso rapidamente.

```
from flask_login import login_required

@app.route("/settings")
@login_required
def settings():
    pass
```

Registrazione degli utenti

Gli utenti possono essere disconnessi chiamando `logout_user()` . Sembra che sia sicuro farlo anche se l'utente non ha effettuato l'accesso in modo che il decoratore `@login_required` possa molto probabilmente essere ommited.

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(somewhere)
```

Cosa succede se un utente non ha effettuato l'accesso e `current_user` all'oggetto `current_user` ?

Per difetto viene restituito un AnonymousUserMixin :

- `is_active` e `is_authenticated` sono `False`
- `is_anonymous` è `True`

- `get_id()` restituisce `None`

Per utilizzare un oggetto diverso per gli utenti anonimi, è possibile chiamare un oggetto chiamabile (di classe o di fabbrica) che crea utenti anonimi su `LoginManager` con:

```
login_manager.anonymous_user = MyAnonymousUser
```

Cosa succederà?

Questo conclude l'introduzione di base all'estensione. Per ulteriori informazioni sulla configurazione e sulle opzioni aggiuntive, si consiglia vivamente di leggere la guida ufficiale .

Timeout della sessione di accesso

È buona norma interrompere la sessione di accesso dopo un determinato periodo di tempo, è possibile farlo con Flask-Login.

```
from flask import Flask, session
from datetime import timedelta
from flask_login import LoginManager, login_require, login_user, logout_user

# Create Flask application

app = Flask(__name__)

# Define Flask-login configuration

login_mgr = LoginManager(app)
login_mgr.login_view = 'login'
login_mgr.refresh_view = 'relogin'
login_mgr.needs_refresh_message = (u"Session timedout, please re-login")
login_mgr.needs_refresh_message_category = "info"


@app.before_request
def before_request():
    session.permanent = True
    app.permanent_session_lifetime = timedelta(minutes=5)
```

La durata predefinita della sessione è di 31 giorni, l'utente deve specificare la vista di aggiornamento del login in caso di timeout.

```
app.permanent_session_lifetime = timedelta(minutes=5)
```

Sopra la linea costringerà l'utente a riconnettersi ogni 5 minuti.

Leggi Autorizzazione e autenticazione online:
https://riptutorial.com/it/flask/topic/9053/autorizzazione-e-autenticazione

Capitolo 5: Blueprints

introduzione

I progetti sono un potente concetto nello sviluppo di applicazioni Flask che consentono alle applicazioni di flask di essere più modulari e di essere in grado di seguire più schemi. Rendono più facile l'amministrazione di applicazioni molto grandi di Flask e come tale possono essere utilizzate per ridimensionare le applicazioni di Flask. È possibile riutilizzare le applicazioni Blueprint, tuttavia non è possibile eseguire un progetto da solo in quanto deve essere registrato sull'applicazione principale.

Examples

Esempio di esempio di un pallone base

Un'applicazione minima di Flask è simile a questa:

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

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

Un'ampia applicazione Flask può separare un file in più file mediante i `blueprints` *.*

Scopo

Rendi più facile per gli altri mantenere l'applicazione.

Struttura di cartelle di grandi applicazioni

```
/app
    /templates
    /static
    /views
        __init__.py
        index.py
    app.py
```

views / index.py

```
from flask import Blueprint, render_template

index_blueprint = Blueprint('index', __name__)

@index_blueprint.route("/")
def index():
```

```
    return "Hello World!"
```

app.py

```
from flask import Flask
from views.index import index_blueprint

application = Flask(__name__)
application.register_blueprint(index_blueprint)
```

Esegui l'applicazione

```
$ export FLASK_APP=app.py
$ flask run
```

Leggi Blueprints online: https://riptutorial.com/it/flask/topic/6427/blueprints

Capitolo 6: Distribuzione dell'applicazione Flask usando il server web uWSGI con Nginx

Examples

Usando uWSGI per eseguire un'applicazione flask

Il server `werkzeug` integrato non è certamente adatto per l'esecuzione di server di produzione. La ragione più ovvia è il fatto che il server `werkzeug` è a thread singolo e quindi può gestire solo una richiesta alla volta.

Per questo motivo, vogliamo utilizzare uWSGI Server per servire la nostra applicazione. In questo esempio installeremo uWSGI ed eseguiremo una semplice applicazione di test.

Installare uWSGI :

```
pip install uwsgi
```

È così semplice. Se non sei sicuro della versione python che usi, pip lo rende esplicito:

```
python3 -m pip install uwsgi  # for python3
python2 -m pip install uwsgi  # for python2
```

Ora creiamo una semplice applicazione di test:

app.py

```
from flask import Flask
from sys import version

app = Flask(__name__)

@app.route("/")
def index():
    return "Hello uWSGI from python version: <br>" + version

application = app
```

In flask il nome convenzionale dell'applicazione è `app` ma uWSGI cerca l' `application` per impostazione predefinita. Ecco perché creiamo un alias per la nostra app nell'ultima riga.

Ora è il momento di eseguire l'app:

```
uwsgi --wsgi-file app.py --http :5000
```

Dovresti vedere il messaggio "Hello uWSGI ..." puntando il tuo browser su `localhost:5000`

Per non digitare il comando completo ogni volta che creeremo un file `uwsgi.ini` per memorizzare quella configurazione:

uwsgi.ini

```
[uwsgi]
http = :9090
wsgi-file = app.py
single-interpreter = true
enable-threads = true
master = true
```

Le opzioni `http` e `wsgi-file` sono le stesse del comando manuale. Ma ci sono altre tre opzioni:

- `single-interpreter` : si consiglia di attivare questa opzione perché potrebbe interferire con l'opzione successiva
- `enable-threads` : questo deve essere attivato se si stanno utilizzando thread aggiuntivi nell'applicazione. Non li usiamo in questo momento ma ora non dobbiamo preoccuparcene.
- `master` : la modalità Master dovrebbe essere abilitata per vari motivi

Ora possiamo eseguire l'app con questo comando:

```
uwsgi --ini uwsgi.ini
```

Installare nginx e configurarlo per uWSGI

Ora vogliamo installare nginx per servire la nostra applicazione.

```
sudo apt-get install nginx  # on debian/ubuntu
```

Quindi creiamo una configurazione per il nostro sito web

```
cd /etc/nginx/site-available  # go to the configuration for available sites
# create a file flaskconfig with your favourite editor
```

flaskconfig

```
server {
    listen 80;
    server_name localhost;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:///tmp/flask.sock;
    }
}
```

Questo dice a nginx di ascoltare sulla porta 80 (predefinita per http) e di servire qualcosa nel percorso root (`/`). Qui diciamo a nginx di agire come un proxy e passare ogni richiesta a un

socket chiamato `flask.sock` trova in `/tmp/` .

Abilitiamo il sito:

```
cd /etc/nginx/sites-enabled
sudo ln -s ../sites-available/flaskconfig .
```

Potresti voler rimuovere la configurazione predefinita se è abilitata:

```
# inside /etc/sites-enabled
sudo rm default
```

Quindi riavvia nginx:

```
sudo service nginx restart
```

Indirizza il tuo browser su `localhost` e vedrai un errore: `502 Bad Gateway` .

Ciò significa che nginx è attivo e funzionante, ma manca il socket. Quindi lasciamolo creare.

Torna al tuo file `uwsgi.ini` e aprilo. Quindi aggiungi queste righe:

```
socket = /tmp/flask.sock
chmod-socket = 666
```

La prima riga dice a uwsgi di creare un socket nella posizione specificata. Il socket verrà utilizzato per ricevere richieste e inviare le risposte. Nell'ultima riga permettiamo ad altri utenti (incluso nginx) di essere in grado di leggere e scrivere da quel socket.

Avvia di nuovo uwsgi con `uwsgi --ini uwsgi.ini` . Ora indirizza nuovamente il browser a `localhost` e vedrai di nuovo il saluto "Hello uWSGI".

Si noti che è ancora possibile vedere la risposta su `localhost:5000` perché uWSGI ora serve l'applicazione tramite http **e** il socket. Quindi disabilitiamo l'opzione http nel file ini

```
http = :5000  # <-- remove this line and restart uwsgi
```

Ora è possibile accedere all'app solo da nginx (o leggendo direttamente da tale socket :)).

Abilita lo streaming dal pallone

Flask ha quella caratteristica che ti permette di trasmettere i dati da una vista usando i generatori.

Cambiamo il file `app.py`

- aggiungere `from flask import Response`
- aggiungi `from datetime import datetime`
- aggiungere `from time import sleep`
- crea una nuova vista:

```
@app.route("/time/")
def time():
    def streamer():
        while True:
            yield "<p>{}</p>".format(datetime.now())
            sleep(1)

    return Response(streamer())
```

Ora apri il tuo browser su `localhost/time/` . Il sito verrà caricato per sempre perché nginx attende fino al completamento della risposta. In questo caso la risposta non sarà mai completa perché invierà per sempre la data e l'ora correnti.

Per evitare l'attesa di nginx, è necessario aggiungere una nuova riga alla configurazione.

Modifica `/etc/nginx/sites-available/flaskconfig`

```
server {
    listen 80;
    server_name localhost;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:///tmp/flask.sock;
        uwsgi_buffering off;  # <-- this line is new
    }
}
```

La linea `uwsgi_buffering off;` dice a nginx di non aspettare fino al completamento di una risposta.

Riavvia nginx: il `sudo service nginx restart` e guarda `localhost/time/` again.

Ora vedrai che ogni secondo appare una nuova linea.

Configurazione di Flask Application, uWGSI, Nginx - Modello di configurazione delle configurazioni del server (predefinito, proxy e cache)

Questo è un porting di setup originato dal tutorial di DigitalOcean su *How to Serve Flask Applications con uWSGI e Nginx su Ubuntu 14.04*

e alcune utili risorse git per i server nginx.

Applicazione del matraccio

Questo tutorial presume che tu usi Ubuntu.

1. locate `var/www/` folder.
2. Crea la tua cartella web app `mkdir myexample`
3. `cd myexample`

facoltativo Si consiglia di configurare l'ambiente virtuale per la distribuzione di applicazioni Web sul server di produzione.

```
sudo pip install virtualenv
```

per installare l'ambiente virtuale.

```
virtualenv myexample
```

per configurare l'ambiente virtuale per la tua app.

```
source myprojectenv/bin/activate
```

per attivare il tuo ambiente. Qui installerai tutti i pacchetti python.

fine opzionale ma raccomandato

Configura flask e gateway uWSGI

Installa il pallone e il gateway uSWGI:

```
pip install uwsgi flask
```

Esempio di app per flask in myexample.py:

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

@application.route("/")
def hello():
    return "<h1>Hello World</h1>"

if __name__ == "__main__":
    application.run(host='0.0.0.0')
```

Crea un file per comunicare tra la tua app Web e il server web: interfaccia gateway [https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface]

```
nano wsgi.py
```

quindi importa il modulo webapp e fallo partire dal punto di ingresso del gateway.

```
from myexample import application

if __name__ == "__main__":
    application.run()
```

Per testare uWSGI:

```
uwsgi --socket 0.0.0.0:8000 --protocol=http -w wsgi
```

Per configurare uWSGI:

1. Creare un file di configurazione `.ini`

   ```
   nano myexample.ini
   ```

2. Configurazione di base per gateway uWSGI

```
# include header for using uwsgi
[uwsgi]
# point it to your python module wsgi.py
module = wsgi
# tell uWSGI to start a master node to serve requests
master = true
# spawn number of processes handling requests
processes = 5
# use a Unix socket to communicate with Nginx. Nginx will pass connections to uWSGI through a
socket, instead of using ports. This is preferable because Nginx and uWSGI stays on the same
machine.
socket = myexample.sock
# ensure file permission on socket to be readable and writable
chmod-socket = 660
# clean the socket when processes stop
vacuum = true
# use die-on-term to communicate with Ubuntu versions using Upstart initialisations: see:
# http://uwsgi-docs.readthedocs.io/en/latest/Upstart.html?highlight=die%20on%20term
die-on-term = true
```

facoltativo se si utilizza l'ambiente virtuale È possibile `deactivate` l'ambiente virtuale.

Configurazione di Nginx Useremo nginx come:

1. server predefinito per passare la richiesta al socket, usando il protocollo uwsgi
2. server proxy davanti al server predefinito
3. cache server per memorizzare le richieste riuscite (ad esempio, potresti voler memorizzare nella cache richieste GET se la tua applicazione web)

Individua la directory dei `sites-available` e crea un file di configurazione per la tua applicazione:

```
sudo nano /etc/nginx/sites-available/myexample
```

Aggiungi il seguente blocco, nei commenti cosa fa:

```
server {


    # setting up default server listening to port 80
    listen 8000 default_server;
    server_name myexample.com; #you can also use your IP

    # specify charset encoding, optional
    charset utf-8;

    # specify root of your folder directory
    root /var/www/myexample;

    # specify locations for your web apps.
    # here using /api endpoint as example
```

```
    location /api {
        # include parameters of wsgi.py and pass them to socket
        include uwsgi_params;
        uwsgi_pass unix:/var/www/myexample/myexample.sock;
    }

}

# Here you will specify caching zones that will be used by your virtual server
# Cache will be stored in /tmp/nginx folder
# ensure nginx have permissions to write and read there!
# See also:
# http://nginx.org/en/docs/http/ngx_http_proxy_module.html

proxy_cache_path /tmp/nginx levels=1:2 keys_zone=my_zone:10m inactive=60m;
proxy_cache_key "$scheme$request_method$host$request_uri";

# set up the virtual host!
server {
    listen   80  default_server;

    # Now www.example.com will listen to port 80 and pass request to http://example.com
    server_name www.example.com;

    # Why not caching responses

    location /api {
        # set up headers for caching
        add_header X-Proxy-Cache $upstream_cache_status;

        # use zone specified above
        proxy_cache my_zone;
        proxy_cache_use_stale updating;
        proxy_cache_lock on;

        # cache all responses ?
        # proxy_cache_valid 30d;

        # better cache only 200 responses :)
        proxy_cache_valid 200 30d;

        # ignore headers to make cache expire
        proxy_ignore_headers X-Accel-Expires Expires Cache-Control;

        # pass requests to default server on port 8000
        proxy_pass http://example.com:8000/api;
    }
}
```

Infine, collega il file alla directory `sites-enabled` ai `sites-enabled` . Per una spiegazione dei siti disponibili e abilitati, vedi risposta: [http://serverfault.com/a/527644]

```
sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled
```

Hai finito ora con nginx. Tuttavia, potresti voler controllare questo modello di caldaia molto prezioso: [https://github.com/h5bp/server-configs-nginx]

Molto utile per la messa a punto.

Ora prova Nginx:

```
sudo nginx -t
```

Avvia Nginx:

```
sudo service nginx restart
```

Automatizzare Ubuntu per avviare uWSGI L'ultima cosa è fare in modo che Ubuntu avvii il gateway wsgi comunicando con l'applicazione, altrimenti dovresti farlo manualmente.

1. Individua la directory per gli script di inizializzazione in Ubuntu e crea un nuovo script:

```
sudo nano /etc/init/myexample.conf
```

2. Aggiungi blocco successivo, commenti in linea per spiegare cosa fa

```
# description for the purpose of this script
description "uWSGI server instance configured to serve myproject"

# Tell to start on system runtime 2, 3, 4, 5. Stop at any other level (0,1,6).
# Linux run levels: [http://www.debianadmin.com/debian-and-ubuntu-linux-run-levels.html]
start on runlevel [2345]
stop on runlevel [!2345]

# Set up permissions! "User" will be the username of your user account on ubuntu.
setuid user
# Allow www-data group to read and write from the socket file.
# www-data is normally the group Nginx and your web applications belong to.
# you may have all web application projects under /var/www/ that belongs to www-data
group
setgid www-data

# tell Ubunutu which environment to use.
# This is the path of your virtual environment: python will be in this path if you
installed virtualenv. Otherwise, use path of your python installation
env PATH=/var/www/myexample/myexample/bin
# then tell to Ubuntu to change and locate your web application directory
chdir /var/www/myexample
# finally execute initialisation script, that load your web app myexample.py
exec uwsgi --ini myexample.ini
```

Ora puoi attivare il tuo script: sudo start myexample

Leggi Distribuzione dell'applicazione Flask usando il server web uWSGI con Nginx online: https://riptutorial.com/it/flask/topic/4637/distribuzione-dell-applicazione-flask-usando-il-server-web-uwsgi-con-nginx

Capitolo 7: File statici

Examples

Utilizzo di file statici

Le applicazioni Web richiedono spesso file statici come file CSS o JavaScript. Per utilizzare i file statici in un'applicazione Flask, creare una cartella denominata `static` nel pacchetto o accanto al modulo e sarà disponibile in `/static` sull'applicazione.

Una struttura di esempio di progetto per l'utilizzo di modelli è la seguente:

```
MyApplication/
    /static/
        /style.css
        /script.js
    /templates/
        /index.html
    /app.py
```

app.py è un esempio di base di Flask con il rendering del modello.

```
from flask import Flask, render_template

app = Flask(__name__)

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

Per utilizzare il file CSS statico e JavaScript nel modello index.html, è necessario utilizzare il nome di endpoint "statico" speciale:

```
{{url_for('static', filename = 'style.css')}}
```

Quindi, **index.html** può contenere:

```
<html>
    <head>
        <title>Static File</title>
        <link href="{{url_for('static', filename = 'style.css')}}" rel="stylesheet">
        <script src="{{url_for('static', filename = 'script.js')}}"></script>
    </head>
    <body>
        <h3>Hello World!</h3>
    </body>
</html>
```

Dopo aver eseguito app.py vedremo la pagina Web in http: // localhost: 5000 / .

File statici in produzione (offerti dal server Web frontend)

Il webserver integrato di Flask è in grado di servire asset statici, e questo funziona bene per lo sviluppo. Tuttavia, per le distribuzioni di produzione che utilizzano qualcosa come uWSGI o Gunicorn per servire l'applicazione Flask, l'attività di servire i file statici è solitamente scaricata sul server Web frontend (Nginx, Apache, ecc.). Questo è un compito piccolo / facile con le app più piccole, specialmente quando tutte le risorse statiche si trovano in una cartella; tuttavia, per le app più grandi e / o quelle che utilizzano i plug-in di Flask che forniscono risorse statiche, può diventare difficile ricordare le posizioni di tutti questi file e copiarli / raccoglierli manualmente in una directory. Questo documento mostra come utilizzare il plugin Flask-Collect per semplificare tale compito.

Si noti che l'obiettivo di questa documentazione è sulla raccolta di asset statici. Per illustrare questa funzionalità, in questo esempio viene utilizzato il plug-in Flask-Bootstrap, che fornisce risorse statiche. Utilizza anche il plug-in Flask-Script, che viene utilizzato per semplificare il processo di creazione di attività da riga di comando. Nessuno di questi plugin è fondamentale per questo documento, sono solo in uso qui per dimostrare la funzionalità. Se si sceglie di non utilizzare Flask-Script, è necessario rivedere i documenti Flask-Collect in modi alternativi per chiamare il comando `collect` .

Nota anche che la configurazione del tuo server web frontend per servire queste risorse statiche è al di fuori dello scopo di questo documento, ti consigliamo di controllare alcuni esempi usando Nginx e Apache per maggiori informazioni. Basti dire che sarete aliasing degli URL che iniziano con "/ static" nella directory centralizzata che Flask-Collect creerà per voi in questo esempio.

L'app è strutturata come segue:

```
/manage.py - The app management script, used to run the app, and to collect static assets
/app/ - this folder contains the files that are specific to our app
    | - __init__.py - Contains the create_app function
    | - static/ - this folder contains the static files for our app.
        | css/styles.css - custom styles for our app (we will leave this file empty)
        | js/main.js - custom js for our app (we will leave this file empty)
    | - templates/index.html - a simple page that extends the Flask-Bootstrap template
```

1. Innanzitutto, crea il tuo ambiente virtuale e installa i pacchetti richiesti: (your-virtualenv) $ pip installa flask-script flask-bootstrap flask-collect

2. Stabilire la struttura del file descritta sopra:

   $ touch manage.py; mkdir -p app / {static / {css, js}, templates}; touch app / { **init** .py, static / {css / styles.css, js / main.js}}

3. Stabilisci i contenuti per i file `manage.py` , `app/__init__.py` e `app/templates/index.html` :

```
# manage.py
#!/usr/bin/env python
import os
from flask_script import Manager, Server
from flask import current_app
```

```
from flask_collect import Collect
from app import create_app

class Config(object):
   # CRITICAL CONFIG VALUE: This tells Flask-Collect where to put our static files!
   # Standard practice is to use a folder named "static" that resides in the top-level of the
project directory.
   # You are not bound to this location, however; you may use basically any directory that you
wish.
   COLLECT_STATIC_ROOT = os.path.dirname(__file__) + '/static'
   COLLECT_STORAGE = 'flask_collect.storage.file'

app = create_app(Config)

manager = Manager(app)
manager.add_command('runserver', Server(host='127.0.0.1', port=5000))

collect = Collect()
collect.init_app(app)

@manager.command
def collect():
  """Collect static from blueprints. Workaround for issue: https://github.com/klen/Flask-
Collect/issues/22"""
  return current_app.extensions['collect'].collect()

if __name__ == "__main__":
   manager.run()
```

```
# app/__init__.py
from flask import Flask, render_template
from flask_collect import Collect
from flask_bootstrap import Bootstrap

def create_app(config):
  app = Flask(__name__)
  app.config.from_object(config)

  Bootstrap(app)
  Collect(app)

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

  return app
```

```
# app/templates/index.html
{% extends "bootstrap/base.html" %}
{% block title %}This is an example page{% endblock %}

{% block navbar %}
<div class="navbar navbar-fixed-top">
  <!-- ... -->
</div>
{% endblock %}

{% block content %}
```

```
  <h1>Hello, Bootstrap</h1>
{% endblock %}
```

4. Con questi file, puoi ora utilizzare lo script di gestione per eseguire l'app:

```
$ ./manage.py runserver # visit http://localhost:5000 to verify that the app works correctly.
```

5. Ora, per raccogliere le risorse statiche per la prima volta. Prima di fare ciò, vale la pena di notare che NON dovresti avere una cartella `static/` nel livello principale della tua app; è qui che Flask-Collect posizionerà tutti i file statici che raccoglierà dalla tua app e dai vari plug-in che potresti utilizzare. Se *si* dispone di una `static/` cartella nel livello superiore della vostra app, è necessario cancellarlo del tutto prima di procedere, come punto di partenza con una lavagna pulita è una parte fondamentale della testimonianza / capire che cosa fa Flask-Collect. Nota che questa istruzione non è applicabile per l'uso quotidiano, ma è semplicemente per illustrare il fatto che Flask-Collect creerà questa directory per te, quindi inserirà un gruppo di file.

Detto ciò, è possibile eseguire il seguente comando per raccogliere le risorse statiche:

```
$ ./manage.py collect
```

Dopo aver fatto ciò, dovresti vedere che Flask-Collect ha creato questa cartella `static/` primo livello e contiene i seguenti file:

```
$ find ./static -type f # execute this from the top-level directory of your app, same dir that
contains the manage.py script
static/bootstrap/css/bootstrap-theme.css
static/bootstrap/css/bootstrap-theme.css.map
static/bootstrap/css/bootstrap-theme.min.css
static/bootstrap/css/bootstrap.css
static/bootstrap/css/bootstrap.css.map
static/bootstrap/css/bootstrap.min.css
static/bootstrap/fonts/glyphicons-halflings-regular.eot
static/bootstrap/fonts/glyphicons-halflings-regular.svg
static/bootstrap/fonts/glyphicons-halflings-regular.ttf
static/bootstrap/fonts/glyphicons-halflings-regular.woff
static/bootstrap/fonts/glyphicons-halflings-regular.woff2
static/bootstrap/jquery.js
static/bootstrap/jquery.min.js
static/bootstrap/jquery.min.map
static/bootstrap/js/bootstrap.js
static/bootstrap/js/bootstrap.min.js
static/bootstrap/js/npm.js
static/css/styles.css
static/js/main.js
```

E questo è tutto: usa il comando `collect` ogni volta che apporti modifiche al CSS o JavaScript della tua app, o quando hai aggiornato un plugin Flask che fornisce risorse statiche (come Flask-Bootstrap in questo esempio).

Leggi File statici online: https://riptutorial.com/it/flask/topic/3678/file-statici

Capitolo 8: Filtri modello Jinja2 personalizzati

Sintassi

- {{my_date_time | my_custom_filter}}
- {{my_date_time | my_custom_filter (args)}}

Parametri

Parametro	Dettagli
valore	Il valore trasmesso da Jinja, da filtrare
args	Argomenti aggiuntivi da passare nella funzione filtro

Examples

Formato datetime in un modello Jinja2

I filtri possono essere definiti in un metodo e quindi aggiunti al dizionario dei filtri di Jinja o definiti in un metodo decorato con `Flask.template_filter` .

Definire e registrare successivamente:

```
def format_datetime(value, format="%d %b %Y %I:%M %p"):
    """Format a date time to (Default): d Mon YYYY HH:MM P"""
    if value is None:
        return ""
    return value.strftime(format)

# Register the template filter with the Jinja Environment
app.jinja_env.filters['formatdatetime'] = format_datetime
```

Definendo con decoratore:

```
@app.template_filter('formatdatetime')
def format_datetime(value, format="%d %b %Y %I:%M %p"):
    """Format a date time to (Default): d Mon YYYY HH:MM P"""
    if value is None:
        return ""
    return value.strftime(format)
```

Leggi Filtri modello Jinja2 personalizzati online: https://riptutorial.com/it/flask/topic/1465/filtri-modello-jinja2-personalizzati

Capitolo 9: Flask on Apache con mod_wsgi

Examples

WSGI Application wrapper

Molte applicazioni Flask sono sviluppate in un *virtualenv* per mantenere le dipendenze per ogni applicazione separata dall'installazione Python a livello di sistema. Assicurati che *mod-wsgi* sia installato nel tuo *virtualenv* :

```
pip install mod-wsgi
```

Quindi crea un wrapper wsgi per l'applicazione Flask. Di solito è conservato nella directory principale dell'applicazione.

my-application.wsgi

```
activate_this = '/path/to/my-application/venv/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
import sys
sys.path.insert(0, '/path/to/my-application')

from app import app as application
```

Questo wrapper attiva l'ambiente virtuale e tutti i suoi moduli e dipendenze installati quando viene eseguito da Apache e si assicura che il percorso dell'applicazione sia il primo nei percorsi di ricerca. Per convenzione, gli oggetti dell'applicazione WSGI sono chiamati `application` .

Configurazione abilitata per i siti Apache per WSGI

Il vantaggio di utilizzare Apache sul server werkzeug incorporato è che Apache è multi-thread, il che significa che è possibile effettuare più connessioni simultaneamente all'applicazione. Ciò è particolarmente utile nelle applicazioni che utilizzano *XmlHttpRequest* (AJAX) sul front-end.

/etc/apache2/sites-available/050-my-application.conf (o configurazione di apache predefinita se non ospitata su un server web condiviso)

```
<VirtualHost *:80>
        ServerName my-application.org

        ServerAdmin admin@my-application.org

        # Must be set, but can be anything unless you want to serve static files
        DocumentRoot /var/www/html

        # Logs for your application will go to the directory as specified:

        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined
```

```
        # WSGI applications run as a daemon process, and need a specified user, group
        # and an allocated number of thread workers. This will determine the number
        # of simultaneous connections available.
        WSGIDaemonProcess my-application user=username group=username threads=12

        # The WSGIScriptAlias should redirect / to your application wrapper:
        WSGIScriptAlias / /path/to/my-application/my-application.wsgi
        # and set up Directory access permissions for the application:
        <Directory /path/to/my-application>
                WSGIProcessGroup my-application
                WSGIApplicationGroup %{GLOBAL}

                AllowOverride none
                Require all granted
        </Directory>
</VirtualHost>
```

Leggi Flask on Apache con mod_wsgi online: https://riptutorial.com/it/flask/topic/6851/flask-on-apache-con-mod-wsgi

Capitolo 10: Flask-SQLAlchemy

introduzione

Flask-SQLAlchemy è un'estensione Flask che aggiunge il supporto per il popolare Python object relational mapper (ORM) SQLAlchemy alle applicazioni Flask. Ha lo scopo di semplificare SQLAlchemy con Flask fornendo alcune implementazioni predefinite per le attività comuni.

Examples

Installazione ed esempio iniziale

Installazione

```
pip install Flask-SQLAlchemy
```

Modello semplice

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))
    email = db.Column(db.String(120), unique=True)
```

L'esempio di codice sopra mostra un semplice modello Flask-SQLAlchemy, possiamo aggiungere un nometabella facoltativo alla dichiarazione del modello, ma spesso non è necessario in quanto Flask-SQLAlchemy utilizzerà automaticamente il nome della classe come nome della tabella durante la creazione del database.

La nostra classe erediterà dal modello di baseclass che è una base dichiarativa configurata, quindi non è necessario per noi definire esplicitamente la base come faremmo con SQLAlchemy.

Riferimento

- URL Pypi: [https://pypi.python.org/pypi/Flask-SQLAlchemy][1]
- URL di documentazione: [http://flask-sqlalchemy.pocoo.org/2.1/][1]

Relazioni: da uno a molti

```
    class User(db.Model):
        id = db.Column(db.Integer, primary_key=True)
        name = db.Column(db.String(80))
        email = db.Column(db.String(120), unique=True)
        posts = db.relationship('Post', backref='user')


    class Post(db.Model):
        id = db.Column(db.Integer, primary_key=True)
        content = db.Column(db.Text)
```

```
        user_id = db.Column(db.Integer, db.ForeignKey('user.id')
```

In questo esempio abbiamo due classi la classe User e la classe Post, la classe User sarà nostra madre e il post sarà il nostro post in quanto solo post può appartenere a un utente ma un utente può avere più post. Per riuscirci, mettiamo una chiave esterna sul figlio che fa riferimento al genitore che proviene dal nostro esempio, poniamo una chiave esterna sulla classe Post per fare riferimento alla classe User. Quindi usiamo `relationship()` sul genitore al quale accediamo tramite il nostro oggetto SQLAlchemy `db` . Questo ci consente quindi di fare riferimento a una collezione di articoli rappresentati dalla classe Post che è nostra figlia.

Per creare una relazione bidirezionale usiamo `backref` , questo permetterà al bambino di fare riferimento al genitore.

Leggi Flask-SQLAlchemy online: https://riptutorial.com/it/flask/topic/10577/flask-sqlalchemy

Capitolo 11: Flask-WTF

introduzione

È una semplice integrazione di Flask e WTForms. Consente la creazione e la gestione semplificata di moduli Web, genera automaticamente un campo nascosto token CRSF nei modelli. Dispone anche di semplici funzioni di convalida del modulo

Examples

Una semplice forma

```
from flask_wtf import FlaskForm
from wtforms import StringField, IntegerField
from wtforms.validators import DataRequired

class MyForm(FlaskForm):
    name = StringField('name', validators=[DataRequired()])
    age = InterField('age', validators=[DataRequired()])
```

Per rendere il template userai qualcosa come questo:

```
<form method="POST" action="/">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(size=20) }}
    <br/>
    {{ form.age.label }} {{ form.age(size=3) }}
    <input type="submit" value="Go">
</form>
```

Il semplice codice sopra generato genererà il nostro semplice modulo web di flask-wtf con un campo token CRSF nascosto.

Leggi Flask-WTF online: https://riptutorial.com/it/flask/topic/10579/flask-wtf

Capitolo 12: Lavorare con JSON

Examples

Restituisce una risposta JSON dall'API Flask

Flask ha un'utilità chiamata `jsonify()` che rende più conveniente restituire le risposte JSON

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/get-json')
def hello():
    return jsonify(hello='world') # Returns HTTP Response with {"hello": "world"}
```

Provalo con `curl`

```
curl -X GET http://127.0.0.1:5000/api/get-json
{
  "hello": "world"
}
```

Altri modi per usare `jsonify()`

Utilizzando un dizionario esistente:

```
person = {'name': 'Alice', 'birth-year': 1986}
return jsonify(person)
```

Utilizzando una lista:

```
people = [{'name': 'Alice', 'birth-year': 1986},
          {'name': 'Bob', 'birth-year': 1985}]
return jsonify(people)
```

Ricezione di JSON da una richiesta HTTP

Se il mimetype della richiesta HTTP è `application/json` , chiamando `request.get_json()` restituirà i dati JSON analizzati (altrimenti restituisce `None`)

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/echo-json', methods=['GET', 'POST', 'DELETE', 'PUT'])
```

```
def add():

    data = request.get_json()
    # ... do your business logic, and return some response
    # e.g. below we're just echo-ing back the received JSON data
    return jsonify(data)
```

Provalo con `curl`

Il parametro `-H 'Content-Type: application/json'` specifica che questa è una richiesta JSON:

```
 curl -X POST -H 'Content-Type: application/json' http://127.0.0.1:5000/api/echo-json -d
'{"name": "Alice"}'
{
  "name": "Alice"
}
```

Per inviare richieste usando altri metodi HTTP, sostituire `curl -X POST` con il metodo desiderato, ad esempio `curl -X GET` , `curl -X PUT` , ecc.

Leggi Lavorare con JSON online: https://riptutorial.com/it/flask/topic/1789/lavorare-con-json

Capitolo 13: Messaggio lampeggiante

introduzione

Messaggio lampeggiante al modello tramite `flash()` funzione `flash()` .

Sintassi

- flash (messaggio, categoria = 'messaggio')
- flash ('ciao, mondo!')
- flash ('Questo è un messaggio di avviso', 'avviso')

Parametri

Messaggio	il messaggio da far lampeggiare.
categoria	la categoria del messaggio, il valore predefinito è il `message` .

Osservazioni

- Ereditarietà dei modelli
- API

Examples

Messaggio semplice lampeggiante

Imposta `SECKET_KEY` , quindi il messaggio lampeggiante nella funzione di visualizzazione:

```
from flask import Flask, flash, render_template

app = Flask(__name__)
app.secret_key = 'some_secret'

@app.route('/')
def index():
    flash('Hello, I'm a message.')
    return render_template('index.html')
```

Quindi visualizza i messaggi in `layout.html` (da cui è stato esteso `index.html`):

```
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
```

```
      <li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

Lampeggiante con le categorie

Imposta il secondo argomento quando si usa la funzione `flash()` in vista:

```
flash('Something was wrong!', 'error')
```

Nel modello, set `with_categories=true` in `get_flashed_messages()` , quindi si ottiene un elenco di tuple sotto forma di `(message, category)` , quindi è possibile utilizzare la categoria come classe HTML.

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
    {% for category, message in messages %}
      <li class="{{ category }}">{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

Leggi Messaggio lampeggiante online: https://riptutorial.com/it/flask/topic/10756/messaggio-lampeggiante

Capitolo 14: Modelli di rendering

Sintassi

- `render_template(template_name_or_list, **context)`

Examples

render_template Utilizzo

Flask ti consente di utilizzare modelli per il contenuto dinamico della pagina web. Una struttura di esempio di progetto per l'utilizzo di modelli è la seguente:

```
myproject/
    /app/
        /templates/
            /index.html
        /views.py
```

`views.py` :

```
from flask import Flask, render_template


app = Flask(__name__)

@app.route("/")
def index():
    pagetitle = "HomePage"
    return render_template("index.html",
                           mytitle=pagetitle,
                           mycontent="Hello World")
```

Si noti che è possibile passare il contenuto dinamico dal gestore di route al modello aggiungendo coppie chiave / valore alla funzione render_templates. Nell'esempio sopra, le variabili "pagetitle" e "mycontent" saranno passate al template per l'inclusione nella pagina renderizzata. Includere queste variabili nel modello racchiudendole in doppie parentesi: `{{mytitle}}`

`index.html` :

```
<html>
    <head>
        <title>{{ mytitle }}</title>
    </head>
    <body>
        <p>{{ mycontent }}</p>
    </body>
</html>
```

Quando viene eseguito come il primo esempio, `http://localhost:5000/` avrà il titolo "HomePage" e

un paragrafo con il contenuto "Hello World".

Leggi Modelli di rendering online: https://riptutorial.com/it/flask/topic/1641/modelli-di-rendering

Capitolo 15: paginatura

Examples

Esempio di percorso di impaginazione con flacone-sqlalchemy Paginate

In questo esempio utilizziamo un parametro nel percorso per specificare il numero di pagina. Impostiamo un valore predefinito di 1 nella `page=1` parametri della funzione `page=1` . Abbiamo un oggetto `User` nel database e lo interrogiamo, ordinando in ordine decrescente, mostrando prima gli utenti più recenti. Quindi usiamo il metodo `paginate` dell'oggetto `query` in flask-sqlalchemy. Passiamo quindi a `render_template` per essere renderizzati.

```
@app.route('/users')
@app.route('/users/page/<int:page>')
def all_users(page=1):
    try:
        users_list = User.query.order_by(
            User.id.desc()
        ).paginate(page, per_page=USERS_PER_PAGE)
    except OperationalError:
        flash("No users in the database.")
        users_list = None

    return render_template(
        'users.html',
        users_list=users_list,
        form=form
    )
```

Rendering di paginazione in Jinja

Qui usiamo l'oggetto che abbiamo passato a `render_template` per visualizzare le pagine, la pagina attiva corrente e anche i pulsanti precedente e successivo se è possibile passare alla pagina precedente / successiva.

```
<!-- previous page -->
{% if users_list.has_prev %}
<li>
    <a href="{{ url_for('users', page=users_list.prev_num) }}">Previous</a>
</li>
{% endif %}

<!-- all page numbers -->
{% for page_num in users_list.iter_pages() %}
    {% if page_num %}
        {% if page_num != users_list.page %}
            <li>
                <a href="{{ url_for('users', page=page_num) }}">{{ page_num }}</a>
            </li>
        {% else %}
            <li class="active">
                <a href="#">{{ page_num }}</a>
```

```
            </li>
        {% endif %}
    {% else %}
        <li>
            <span class="ellipsis" style="white-space; nowrap; overflow: hidden; text-overflow:
ellipsis">…</span>
        </li>
    {% endif %}
{% endfor %}

<!-- next page -->
{% if users_list.has_next %}
<li>
    <a href="{{ url_for('users', page=users_list.next_num) }}">Next</a></li>
{% endif %}
{% endif %}
```

Leggi paginatura online: https://riptutorial.com/it/flask/topic/6460/paginatura

Capitolo 16: Reindirizzare

Sintassi

- reindirizzamento (posizione, codice, risposta)

Parametri

Parametro	Dettagli
Posizione	La posizione a cui la risposta dovrebbe reindirizzare.
codice	(Facoltativo) Il codice di stato di reindirizzamento, 302 per impostazione predefinita. I codici supportati sono 301, 302, 303, 305 e 307.
Risposta	(Facoltativo) Una classe di risposta da utilizzare durante l'istanziazione di una risposta. Il valore predefinito è werkzeug.wrappers.Response se non specificato.

Osservazioni

Il parametro location deve essere un URL. Può essere inserito come grezzo, ad esempio " http://www.webpage.com " oppure può essere creato con la funzione url_for ().

Examples

Semplice esempio

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

app = Flask(__name__)

@app.route('/')
def main_page():
    return render_template('main.html')

@app.route('/main')
def go_to_main():
    return redirect(url_for('main_page'))
```

Passando lungo i dati

```
# ...
# same as above

@app.route('/welcome/<name>')
```

```
def welcome(name):
    return render_template('main.html', name=name)

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # ...
        # check for valid login, assign username
        if valid:
            return redirect(url_for('main_page', name=username))
        else:
            return redirect(url_for('login_error'))
    else:
        return render_template('login.html')
```

Leggi Reindirizzare online: https://riptutorial.com/it/flask/topic/6856/reindirizzare

Capitolo 17: Routing

Examples

Percorsi di base

Le `route` in Flask possono essere definite utilizzando il `route` decorator dell'istanza dell'applicazione Flask:

```
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask'
```

Il decoratore del `route` prende una stringa che corrisponde all'URL corrispondente. Quando una richiesta per un URL che corrisponde a questa stringa viene ricevuta dall'applicazione, verrà richiamata la funzione decorata (chiamata anche *funzione di visualizzazione*). Quindi per un percorso di circa avremmo:

```
@app.route('/about')
def about():
    return 'About page'
```

È importante notare che questi percorsi **non** sono espressioni regolari come in Django.

È anche possibile definire *regole variabili* per estrarre i valori del segmento URL in variabili:

```
@app.route('/blog/posts/<post_id>')
def get_blog_post(post_id):
    # look up the blog post with id post_id
    # return some kind of HTML
```

Qui la regola della variabile si trova nell'ultimo segmento dell'URL. Qualsiasi valore si trovi nell'ultimo segmento dell'URL verrà passato alla funzione di visualizzazione (`get_blog_post`) come parametro `post_id` . Quindi una richiesta a `/blog/posts/42` recupererà (o tenterà di recuperare) il post del blog con un id di 42.

È anche comune riutilizzare gli URL. Ad esempio, forse vogliamo che `/blog/posts` restituiscano un elenco di tutti i post del blog. Quindi potremmo avere due percorsi per la stessa funzione di visualizzazione:

```
@app.route('/blog/posts')
@app.route('/blog/posts/<post_id>')
def get_blog_post(post_id=None):
    # get the post or list of posts
```

Nota qui che dobbiamo anche fornire il valore predefinito di `None` per `post_id` in `get_blog_post` .

Quando viene abbinato il primo percorso, non ci sarà alcun valore da passare alla funzione di visualizzazione.

Si noti inoltre che, per impostazione predefinita, il tipo di una regola variabile è una stringa. Tuttavia, puoi specificare diversi tipi come `int` e `float` aggiungendo il prefisso alla variabile:

```
@app.route('/blog/post/<int:post_id>')
```

I convertitori di URL incorporati di Flask sono:

```
string | Accepts any text without a slash (the default).
int    | Accepts integers.
float  | Like int but for floating point values.
path   | Like string but accepts slashes.
any    | Matches one of the items provided
uuid   | Accepts UUID strings
```

Se dovessimo provare a visitare l'URL `/blog/post/foo` con un valore nell'ultimo segmento di URL che non può essere convertito in un numero intero, l'applicazione restituirebbe un errore 404. Questa è l'azione corretta perché non esiste una regola con `/blog/post` e una stringa nell'ultimo segmento.

Infine, i percorsi possono essere configurati per accettare anche i metodi HTTP. Il decoratore del `route` accetta un argomento parola chiave `methods` che è un elenco di stringhe che rappresentano i metodi HTTP accettabili per questa rotta. Come si può presumere, il valore predefinito è solo `GET` . Se avessimo un modulo per aggiungere un nuovo post del blog e volessimo restituire l'HTML per la richiesta `GET` e analizzare i dati del modulo per la richiesta `POST` , il percorso sarebbe simile al seguente:

```
@app.route('/blog/new', methods=['GET', 'POST'])
def new_post():
    if request.method == 'GET':
        # return the form
    elif request.method == 'POST':
        # get the data from the form values
```

La `request` si trova nel pacchetto del `flask` . Si noti che quando si utilizzano i `methods` ragionamento parola chiave, dobbiamo essere espliciti sui metodi HTTP ad accettare. Se avessimo elencato solo `POST` , il percorso non rispondeva più alle richieste `GET` e restituiva un errore 405.

Percorso tutto compreso

Può essere utile avere una vista generale in cui si gestisce la logica complessa in base al percorso. Questo esempio utilizza due regole: La prima regola cattura specifico `/` e la seconda regola cattura percorsi arbitrari con incorporata `path` convertitore. Il convertitore di `path` corrisponde a qualsiasi stringa (comprese le barre) Vedi Regole variabili flask

```
@app.route('/', defaults={'u_path': ''})
@app.route('/<path:u_path>')
def catch_all(u_path):
```

```
    print(repr(u_path))
    ...
```

```
c = app.test_client()
c.get('/')  # u_path = ''
c.get('/hello')  # u_path = 'hello'
c.get('/hello/stack/overflow/')  # u_path = 'hello/stack/overflow/'
```

Routing e metodi HTTP

Per impostazione predefinita, le rotte rispondono solo alle richieste `GET` . È possibile modificare questo comportamento fornendo l'argomento `methods` al decoratore `route()` .

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

È anche possibile mappare diverse funzioni allo stesso endpoint in base al metodo HTTP utilizzato.

```
@app.route('/endpoint', methods=['GET'])
def get_endpoint():
    #respond to GET requests for '/endpoint'


@app.route('/endpoint', methods=['POST', 'PUT', 'DELETE'])
def post_or_put():
    #respond to POST, PUT, or DELETE requests for '/endpoint'
```

Leggi Routing online: https://riptutorial.com/it/flask/topic/2415/routing

Capitolo 18: segnali

Osservazioni

Flask supporta i segnali usando Blinker . Il supporto del segnale è facoltativo; saranno abilitati solo se Blinker è installato.

```
pip install blinker
```

http://flask.pocoo.org/docs/dev/signals/

I segnali non sono asincroni. Quando viene inviato un segnale, esegue immediatamente ciascuna delle funzioni collegate in sequenza.

Examples

Collegamento ai segnali

Utilizzare di un segnale `connect` metodo per collegare una funzione per un segnale. Quando viene inviato un segnale, ciascuna funzione connessa viene chiamata con il mittente e gli eventuali argomenti denominati forniti dal segnale.

```
from flask import template_rendered

def log_template(sender, template, context, **kwargs):
    sender.logger.info(
        'Rendered template %(template)r with context %(context)r.',
        template=template, context=context
    )

template_rendered.connect(log_template)
```

Vedere la documentazione sui segnali integrati per informazioni su quali argomenti forniscono. Un pattern utile è l'aggiunta di un argomento `**kwargs` per catturare qualsiasi argomento imprevisto.

Segnali personalizzati

Se si desidera creare e inviare segnali nel proprio codice (ad esempio, se si sta scrivendo un'estensione), creare una nuova istanza `Signal` e chiamare `send` quando gli abbonati devono essere avvisati. I segnali vengono creati utilizzando uno `Namespace` .

```
from flask import current_app
from flask.signals import Namespace

namespace = Namespace()
message_sent = namespace.signal('mail_sent')
```

```
def message_response(recipient, body):
    ...
    message_sent.send(
        current_app._get_current_object(),
        recipient=recipient,
        body=body
    )

@message_sent.connect
def log_message(app, recipient, body):
    ...
```

Preferisci usare il supporto del segnale di Flask sull'uso diretto di Blinker. Racchiude la libreria in modo che i segnali rimangano facoltativi se gli sviluppatori che utilizzano la tua estensione non hanno scelto di installare Blinker.

Leggi segnali online: https://riptutorial.com/it/flask/topic/2331/segnali

Capitolo 19: sessioni

Osservazioni

Le sessioni derivano dai dizionari, il che significa che funzioneranno con i metodi di dizionario più comuni.

Examples

Utilizzando l'oggetto sessioni all'interno di una vista

Innanzitutto, assicurati di aver importato le sessioni dal pallone

```
from flask import session
```

Per utilizzare la sessione, un'applicazione Flask richiede una **SECRET_KEY** definita.

```
app = Flask(__name__)
app.secret_key = 'app secret key'
```

Le sessioni vengono implementate per impostazione predefinita utilizzando un **cookie** firmato con la chiave segreta. Ciò garantisce che i dati non vengano modificati tranne che dalla tua applicazione, quindi assicurati di sceglierne uno sicuro! Un browser invierà i cookie all'applicazione insieme a ciascuna richiesta, consentendo la persistenza dei dati tra le richieste.

Per usare una sessione basta fare riferimento all'oggetto (si comporterà come un dizionario)

```
@app.route('/')
def index():
    if 'counter' in session:
        session['counter'] += 1
    else:
        session['counter'] = 1
    return 'Counter: '+str(session['counter'])
```

Per rilasciare una variabile di sessione usa il metodo **pop ()** .

```
session.pop('counter', None)
```

Codice di esempio:

```
from flask import Flask, session

app = Flask(__name__)
app.secret_key = 'app secret key'

@app.route('/')
```

```
def index():
    if 'counter' in session:
        session['counter'] += 1
    else:
        session['counter'] = 1
    return 'Counter: '+str(session['counter'])

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

Leggi sessioni online: https://riptutorial.com/it/flask/topic/2748/sessioni

Capitolo 20: Upload di file

Sintassi

- request.files ['name'] # singolo file richiesto
- request.files.get ('name') # Nessuno se non pubblicato
- request.files.getlist ('name') # lista di zero o più file pubblicati
- CombinedMultiDict ((request.files, request.form)) # combina i dati di forma e file

Examples

Caricamento di file

Modulo HTML

- Utilizza un input di tipo `file` e il browser fornirà un campo che consente all'utente di selezionare un file da caricare.
- Solo i moduli con il metodo `post` possono inviare dati di file.
- Assicurati di impostare l' `enctype=multipart/form-data` . Altrimenti verrà inviato il nome del file ma non i dati del file.
- Utilizzare l'attributo `multiple` sull'input per consentire la selezione di più file per il singolo campo.

```
<form method=post enctype=multipart/form-data>
    <!-- single file for the "profile" field -->
    <input type=file name=profile>
    <!-- multiple files for the "charts" field -->
    <input type=file multiple name=charts>
    <input type=submit>
</form>
```

Richieste Python

Requests è una potente libreria Python per effettuare richieste HTTP. Puoi usarlo (o altri strumenti) per pubblicare file senza browser.

- Apri i file da leggere in modalità binaria.
- Ci sono più strutture di dati che i `files` richiedono. Questo dimostra un elenco di tuple `(name, data)` , che consente più file come il modulo sopra.

```
import requests

with open('profile.txt', 'rb') as f1, open('chart1.csv', 'rb') as f2, open('chart2.csv', 'rb')
as f3:
```

```
    files = [
        ('profile', f1),
        ('charts', f2),
        ('charts', f3)
    ]
    requests.post('http://localhost:5000/upload', files=files)
```

Questo non vuole essere un elenco esaustivo. Per esempi che utilizzano il tuo strumento preferito o scenari più complessi, consulta i documenti per quello strumento.

Salva i caricamenti sul server

I file caricati sono disponibili in `request.files` , i nomi dei campi di mappatura `MultiDict` in oggetti file. Utilizza `getlist` - anziché `[]` o `get` - se più file sono stati caricati con lo stesso nome di campo.

```
request.files['profile']  # single file (even if multiple were sent)
request.files.getlist('charts')  # list of files (even if one was sent)
```

Gli oggetti in `request.files` hanno un metodo di `save` che salva il file localmente. Creare una directory comune in cui salvare i file.

L'attributo `filename` è il nome con cui è stato caricato il file. Questo può essere impostato arbitrariamente dal client, quindi passarlo attraverso il metodo `secure_filename` per generare un nome valido e sicuro per salvare come. Ciò non garantisce che il nome sia *univoco* , quindi i file esistenti verranno sovrascritti a meno che non si eseguano ulteriori operazioni per rilevarlo.

```
import os
from flask import render_template, request, redirect, url_for
from werkzeug import secure_filename

# Create a directory in a known location to save files to.
uploads_dir = os.path.join(app.instance_path, 'uploads')
os.makedirs(uploads_dir, exists_ok=True)

@app.route('/upload', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        # save the single "profile" file
        profile = request.files['profile']
        profile.save(os.path.join(uploads_dir, secure_filename(profile.filename)))

        # save each "charts" file
        for file in request.files.getlist('charts'):
            file.save(os.path.join(uploads_dir, secure_filename(file.name)))

        return redirect(url_for('upload'))

    return render_template('upload.html')
```

Trasmissione dei dati a WTForms e Flask-WTF

WTForms fornisce un `FileField` per `FileField` rendering di un input di tipo di file. Non fa nulla di speciale con i dati caricati. Tuttavia, poiché Flask suddivide i dati del modulo (`request.form`) e i

dati del file (`request.files`), è necessario assicurarsi di passare i dati corretti durante la creazione del modulo. Puoi usare `CombinedMultiDict` per combinare i due in un'unica struttura che WTForms comprende.

```
form = ProfileForm(CombinedMultiDict((request.files, request.form)))
```

Se stai usando Flask-WTF , un'estensione per integrare Flask e WTForms, il passaggio dei dati corretti verrà gestito automaticamente.

A causa di un bug in WTForms, sarà presente un solo file per ogni campo, anche se sono stati caricati più file. Vedi questo problema per maggiori dettagli. Sarà corretto in 3.0.

FILE CSV PARSE CARICARE COME ELENCO DEI DIZIONARI IN BANCO SENZA RISPARMIO

Gli sviluppatori spesso devono progettare siti Web che consentano agli utenti di caricare un file CSV. Di solito non vi è **alcun motivo** per **salvare** il file CSV effettivo poiché i dati verranno elaborati e / o memorizzati in un database una volta caricati. Tuttavia, molti se non la maggior parte, i metodi PYTHON per analizzare i dati CSV richiedono che i dati vengano letti come un file. Questo potrebbe presentare un po 'di mal di testa se si utilizza **FLASK** per lo sviluppo web.

Supponiamo che il nostro CSV abbia una riga di intestazione e assomiglia al seguente:

```
h1,h2,h3
'yellow','orange','blue'
'green','white','black'
'orange','pink','purple'
```

Ora, supponiamo che il modulo html per caricare un file sia il seguente:

```
<form action="upload.html" method="post" enctype="multipart/form-data">
    <input type="file" name="fileupload" id="fileToUpload">
    <input type="submit" value="Upload File" name="submit">
</form>
```

Dal momento che nessuno vuole reinventare la ruota, decidi di **importare CSV** nel tuo script **FLASK** . Non vi è alcuna garanzia che le persone caricheranno il file csv con le colonne nell'ordine corretto. Se il file csv ha una riga di intestazione, con l'aiuto del metodo **csv.DictReader** puoi leggere il file CSV come un elenco di dizionari, immesso dalle voci nella riga di intestazione. Tuttavia, **csv.DictReader** necessita di un file e non accetta direttamente le stringhe. Si potrebbe pensare che sia necessario utilizzare i metodi **FLASK** per salvare prima il file caricato, ottenere il nuovo nome e il percorso del file, aprirlo utilizzando **csv.DictReader** e quindi eliminare il file. Sembra un po 'uno spreco.

Fortunatamente, possiamo ottenere il contenuto del file come una stringa e quindi dividere la stringa in righe terminate. Il metodo csv **csv.DictReader** lo accetterà come sostituto di un file. Il seguente codice mostra come ciò può essere realizzato senza salvare temporaneamente il file.

```
@application.route('upload.html',methods = ['POST'])
def upload_route_summary():
    if request.method == 'POST':

        # Create variable for uploaded file
        f = request.files['fileupload']

        #store the file contents as a string
        fstring = f.read()

        #create list of dictionaries keyed by header row
        csv_dicts = [{k: v for k, v in row.items()} for row in
csv.DictReader(fstring.splitlines(), skipinitialspace=True)]

        #do something list of dictionaries
    return "success"
```

La variabile **csv_dicts** è ora la seguente lista di dizionari:

```
   csv_dicts =
    [
        {'h1':'yellow','h2':'orange','h3':'blue'},
        {'h1':'green','h2':'white','h3':'black'},
        {'h1':'orange','h2':'pink','h3':'purple'}
    ]
```

Se sei nuovo su PYTHON, puoi accedere ai dati come segue:

```
csv_dicts[1]['h2'] = 'white'
csv_dicts[0]['h3'] = 'blue'
```

Altre soluzioni implicano l'importazione del modulo **io** e l'uso del metodo **io.Stream** . Sento che questo è un approccio più diretto. Credo che il codice sia un po 'più semplice da seguire rispetto all'utilizzo del metodo **io** . Questo approccio è specifico dell'esempio di analisi di un file CSV caricato.

Leggi Upload di file online: https://riptutorial.com/it/flask/topic/5459/upload-di-file

Capitolo 21: Viste basate sulla classe

Examples

Esempio di base

Con le visualizzazioni basate sulla classe, utilizziamo le classi anziché i metodi per implementare le nostre visualizzazioni. Un semplice esempio di utilizzo di visualizzazioni basate su classi è il seguente:

```
from flask import Flask
from flask.views import View

app = Flask(__name__)


class HelloWorld(View):

    def dispatch_request(self):
        return 'Hello World!'


class HelloUser(View):

    def dispatch_request(self, name):
        return 'Hello {}'.format(name)

app.add_url_rule('/hello', view_func=HelloWorld.as_view('hello_world'))
app.add_url_rule('/hello/<string:name>', view_func=HelloUser.as_view('hello_user'))

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

Leggi Viste basate sulla classe online: https://riptutorial.com/it/flask/topic/7494/viste-basate-sulla-classe

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Flask	arsho, bakkal, Community, davidism, ettanany, Martijn Pieters, mmenschig, Sean Vieira, Shrike
2	Accesso ai dati della richiesta	RPi Awesomeness
3	analisi	bakkal, oggo
4	Autorizzazione e autenticazione	boreq, Ninad Mhatre
5	Blueprints	Achim Munene, Kir Chou, stamaimer
6	Distribuzione dell'applicazione Flask usando il server web uWSGI con Nginx	Gal Dreiman, Tempux, user305883, wimkeir, Wombatz
7	File statici	arsho, davidism, MikeC, YellowShark
8	Filtri modello Jinja2 personalizzati	Celeo, dylanj.nz
9	Flask on Apache con mod_wsgi	Aaron D
10	Flask-SQLAlchemy	Achim Munene, arsho, Matt Davis
11	Flask-WTF	Achim Munene
12	Lavorare con JSON	bakkal, g3rv4
13	Messaggio lampeggiante	Grey Li
14	Modelli di rendering	arsho, atayenel, Celeo, fabioqcorreia, Jon Chan, MikeC
15	paginatura	hdbuster
16	Reindirizzare	coralvanda
17	Routing	davidism, Douglas Starnes, Grey Li, junnytony, Luke Taylor,

		MikeC, mmenschig, sytech
18	segnali	davidism
19	sessioni	arsho, PsyKzz, this-vidor
20	Upload di file	davidism, sigmasum
21	Viste basate sulla classe	ettanany