EBook Gratis

APRENDIZAJE Flask

Free unaffiliated eBook created from
Stack Overflow contributors.

#flask

Tabla de contenido

Acerca de

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

Capítulo 1: Comenzando con el matraz

Observaciones

Flask es un micro-marco de aplicaciones web de Python construido sobre la biblioteca WSGI de Werkzeug . El matraz puede ser "micro", pero está listo para el uso de producción en una variedad de necesidades.

El "micro" en el micro-marco significa que Flask pretende mantener el núcleo simple pero extensible. Flask no tomará muchas decisiones por usted, como la base de datos que utilizará y las decisiones que tome serán fáciles de cambiar. Todo depende de ti, para que Flask pueda ser todo lo que necesites y nada que no.

La comunidad admite un rico ecosistema de extensiones para que su aplicación sea más potente y aún más fácil de desarrollar. A medida que su proyecto crece, usted es libre de tomar las decisiones de diseño adecuadas para sus requisitos.

Versiones

Versión	Nombre clave	Fecha de lanzamiento
0.12	Punsch	2016-12-21
0.11	Ajenjo	2016-05-29
0.10	Licor de limón italiano	2013-06-13

Examples

Instalación - Estable

Utilice pip para instalar Flask en un virtualenv.

```
pip install flask
```

Instrucciones paso a paso para crear un virtualenv para tu proyecto:

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

Nunca uses `sudo pip install` menos que entiendas exactamente lo que estás haciendo.

Mantenga su proyecto en un virtualenv local, no lo instale en el sistema Python a menos que esté usando el administrador de paquetes del sistema.

Hola Mundo

Crear `hello.py` :

```
from flask import Flask

app = Flask(__name__)


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

Luego ejecútalo con:

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

Agregar el siguiente código permitirá ejecutarlo directamente con `python hello.py` .

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

Instalación - Últimas

Si desea utilizar el código más reciente, puede instalarlo desde el repositorio. Si bien es posible que obtenga nuevas funciones y correcciones, solo se admiten oficialmente las versiones numeradas.

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

Instalación - Desarrollo

Si desea desarrollar y contribuir al proyecto Flask, clone el repositorio e instale el código en modo de desarrollo.

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

También hay algunas dependencias y herramientas adicionales a tener en cuenta.

esfinge

Se utiliza para construir la documentación.

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

py.test

Se utiliza para ejecutar el conjunto de pruebas.

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

toxina

Se utiliza para ejecutar el conjunto de pruebas contra varias versiones de Python.

```
pip install tox
tox
```

Tenga en cuenta que tox solo utiliza intérpretes que ya están instalados, por lo que si no tiene Python 3.3 instalado en su ruta, no se probará.

Lea Comenzando con el matraz en línea: https://riptutorial.com/es/flask/topic/790/comenzando-con-el-matraz

Capítulo 2: Acceso a los datos de solicitud

Introducción

Cuando se trabaja con una aplicación web, a veces es importante acceder a los datos incluidos en la solicitud, más allá de la URL.

En Frasco, esto se almacena en el objeto de **solicitud** global, al que puede acceder en su código a través `from flask import request` .

Examples

Accediendo a la cadena de consulta

La cadena de consulta es la parte de una solicitud que sigue a la URL, precedida por un `?` marca.

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

Para este ejemplo, estamos creando un servidor web de eco simple que devuelve todo lo enviado a través del campo `echo` en las solicitudes `GET` .

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

Ejemplo de matraz :

```
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 combinada y cadena de consulta

Flask también permite el acceso a un CombinedMultiDict que da acceso a los atributos `request.form` y `request.args` bajo una variable.

Este ejemplo extrae datos de un `name` campo de formulario enviado junto con el campo de `echo` en la cadena de consulta.

Ejemplo de matraz :

```
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()
```

Acceso a campos de formulario

Puede acceder a los datos del formulario enviados a través de una `POST` o `PUT` en Flask a través del atributo `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()
```

Lea Acceso a los datos de solicitud en línea: https://riptutorial.com/es/flask/topic/8622/acceso-a-los-datos-de-solicitud

Capítulo 3: Archivos estáticos

Examples

Uso de archivos estáticos

Las aplicaciones web a menudo requieren archivos estáticos como archivos CSS o JavaScript. Para usar archivos estáticos en una aplicación de Flask, cree una carpeta llamada `static` en su paquete o junto a su módulo y estará disponible en `/static` en la aplicación.

Una estructura de proyecto de ejemplo para usar plantillas es la siguiente:

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

app.py es un ejemplo básico de Flask con representación de plantilla.

```
from flask import Flask, render_template

app = Flask(__name__)

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

Para usar el archivo CSS y JavaScript estático en la plantilla index.html, necesitamos usar el nombre del punto final 'estático' especial:

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

Entonces, **index.html** puede contener:

```
<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>
```

Después de ejecutar app.py veremos la página web en http: // localhost: 5000 / .

Archivos estáticos en producción (servidos por el servidor web frontend)

El servidor web incorporado de Flask es capaz de servir activos estáticos, y esto funciona bien para el desarrollo. Sin embargo, para implementaciones de producción que utilizan algo como uWSGI o Gunicorn para servir a la aplicación Flask, la tarea de servir archivos estáticos se descarga normalmente en el servidor web frontend (Nginx, Apache, etc.). Esta es una tarea pequeña / fácil con aplicaciones más pequeñas, especialmente cuando todos los activos estáticos están en una carpeta; sin embargo, para aplicaciones más grandes y / o las que usan complementos de Flask que proporcionan activos estáticos, puede ser difícil recordar las ubicaciones de todos esos archivos y copiarlos / recopilarlos manualmente en un directorio. Este documento muestra cómo usar el complemento Flask-Collect para simplificar esa tarea.

Tenga en cuenta que el foco de esta documentación está en la colección de activos estáticos. Para ilustrar esa funcionalidad, este ejemplo utiliza el complemento Flask-Bootstrap, que proporciona activos estáticos. También utiliza el complemento Flask-Script, que se utiliza para simplificar el proceso de creación de tareas de línea de comandos. Ninguno de estos complementos es crítico para este documento, solo se utilizan aquí para demostrar la funcionalidad. Si elige no usar Flask-Script, querrá revisar los documentos de Flask-Collect para encontrar formas alternativas de llamar al comando de `collect` .

También tenga en cuenta que la configuración de su servidor web frontend para servir a estos activos estáticos está fuera del alcance de este documento, por lo que le recomendamos consultar algunos ejemplos que utilizan Nginx y Apache para obtener más información. Basta con decir que aliasing URL que comienzan con "/ static" al directorio centralizado que Flask-Collect creará para usted en este ejemplo.

La aplicación está estructurada de la siguiente manera:

```
/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. Primero, cree su entorno virtual e instale los paquetes necesarios: (your-virtualenv) $ pip install flask-script flask-bootstrap flask-collect

2. Establecer la estructura de archivos descrita anteriormente:

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

3. Establezca el contenido de los `manage.py` , `app/__init__.py` y `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 esos archivos en su lugar, ahora puede usar el script de administración para ejecutar la aplicación:

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

5. Ahora, para recoger sus activos estáticos por primera vez. Antes de hacer esto, vale la pena señalar nuevamente que NO debe tener una carpeta `static/` en el nivel superior de su aplicación; Aquí es donde Flask-Collect colocará todos los archivos estáticos que recopilará de su aplicación y los diversos complementos que podría estar usando. Si usted tiene *un* `static/` carpeta en el nivel superior de la aplicación, debe eliminar por completo antes de continuar, ya que a partir de cero es una parte crítica del testimonio / comprensión de lo que hace Frasco-Collect. Tenga en cuenta que esta instrucción no es aplicable para el uso diario, es simplemente para ilustrar el hecho de que Flask-Collect va a crear este directorio para usted, y luego va a colocar un montón de archivos allí.

Dicho esto, puedes ejecutar el siguiente comando para recopilar tus activos estáticos:

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

Después de hacerlo, debería ver que Flask-Collect ha creado esta carpeta `static/` nivel superior, y contiene los siguientes archivos:

```
$ 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
```

Y eso es todo: use el comando `collect` cada vez que realice ediciones en el CSS o JavaScript de su aplicación, o cuando haya actualizado un complemento de Flask que proporcione activos estáticos (como Flask-Bootstrap en este ejemplo).

Lea Archivos estáticos en línea: https://riptutorial.com/es/flask/topic/3678/archivos-estaticos

Capítulo 4: Autorización y autenticación

Examples

Usando la extensión de inicio de sesión de matraz

Una de las formas más sencillas de implementar un sistema de autorización es usar la extensión de inicio de sesión de matraz . El sitio web del proyecto contiene un inicio rápido detallado y bien escrito, cuya versión más corta está disponible en este ejemplo.

Idea general

La extensión expone un conjunto de funciones utilizadas para:

- loguear usuarios en
- cerrar sesión de usuarios
- comprobar si un usuario ha iniciado sesión o no, y averiguar qué usuario es ese

Lo que no hace y lo que tiene que hacer por su cuenta:

- no proporciona una forma de almacenar los usuarios, por ejemplo, en la base de datos
- no proporciona una forma de verificar las credenciales del usuario, por ejemplo, nombre de usuario y contraseña

A continuación hay un conjunto mínimo de pasos necesarios para que todo funcione.

Recomendaría colocar todo el código relacionado con autenticación en un módulo o paquete separado, por ejemplo, `auth.py` De esa manera puede crear las clases, los objetos o las funciones personalizadas necesarias por separado.

Crear un `LoginManager`

La extensión utiliza una clase `LoginManager` que debe registrarse en el objeto de la aplicación `Flask` .

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

Como se mencionó anteriormente, `LoginManager` puede ser, por ejemplo, una variable global en un archivo o paquete separado. Luego se puede importar en el archivo en el que se crea el objeto `Flask` o en la función de fábrica de su aplicación y se inicializa.

Especifique una devolución de llamada utilizada para cargar usuarios

Un usuario normalmente se cargará desde una base de datos. La devolución de llamada debe devolver un objeto que represente a un usuario correspondiente a la ID proporcionada. Debe devolver `None` si el ID no es válido.

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

Esto se puede hacer directamente debajo de crear su `LoginManager` .

Una clase representando a tu usuario.

Como se mencionó, la `user_loader` llamada `user_loader` tiene que devolver un objeto que represente a un usuario. ¿Qué significa eso exactamente? Ese objeto puede, por ejemplo, ser un envoltorio alrededor de objetos de usuario almacenados en su base de datos o simplemente directamente un modelo de su base de datos. Ese objeto tiene que implementar los siguientes métodos y propiedades. Eso significa que si la devolución de llamada devuelve el modelo de su base de datos, debe asegurarse de que las propiedades y los métodos mencionados se agreguen a su modelo.

- `is_authenticated`

  Esta propiedad debe devolver `True` si el usuario está autenticado, es decir, ha proporcionado credenciales válidas. `user_loader` asegurarse de que los objetos que representan a sus usuarios devueltos por la `user_loader` llamada `user_loader` devuelvan `True` para ese método.

- `is_active`

  Esta propiedad debe devolver True si se trata de un usuario activo: además de autenticarse, también han activado su cuenta, no se han suspendido o cualquier condición que su aplicación tenga para rechazar una cuenta. Las cuentas inactivas pueden no iniciar sesión. Si no tiene un mecanismo presente, devuelva `True` con este método.

- `is_anonymous`

  Esta propiedad debería devolver True si este es un usuario anónimo. Eso significa que su objeto de usuario devuelto por la `user_loader` llamada `user_loader` debe devolver `True` .

- `get_id()`

  Este método debe devolver un código Unicode que identifique de forma única a este usuario y se puede utilizar para cargar al usuario desde la `user_loader` llamada `user_loader` . Tenga en cuenta que este debe ser un Unicode: si el ID es de forma nativa un int o algún otro tipo, deberá convertirlo a Unicode. Si la `user_loader` llamada `user_loader` devuelve objetos de la

base de datos, este método probablemente devolverá el ID de la base de datos de este usuario en particular. Por supuesto, la misma ID debería hacer que la devolución de llamada `user_loader` devuelva al mismo usuario más adelante.

Si desea hacer las cosas más fáciles para usted (** de hecho, se recomienda) puede heredar de `UserMixin` en el objeto devuelto por la `user_loader` llamada `user_loader` (probablemente un modelo de base de datos). Puede ver cómo esos métodos y propiedades se implementan de forma predeterminada en esta mezcla aquí .

Entrar a los usuarios en

La extensión deja la validación del nombre de usuario y la contraseña ingresados por el usuario. De hecho, a la extensión no le importa si usa un combo de nombre de usuario y contraseña u otro mecanismo. Este es un ejemplo para iniciar sesión en los usuarios con el nombre de usuario y la contraseña.

```
@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)
```

En general, el registro de los usuarios se realiza llamando a login_user y pasando una instancia de un objeto que representa a su usuario mencionado anteriormente. Como se muestra, esto generalmente ocurrirá después de recuperar al usuario de la base de datos y validar sus credenciales, sin embargo, el objeto del usuario simplemente aparece mágicamente en este ejemplo.

He iniciado sesión en un usuario, ¿y ahora qué?

Se puede acceder al objeto devuelto por la `user_loader` llamada `user_loader` de varias maneras.

- En plantillas:

  La extensión lo inyecta automáticamente bajo el nombre `current_user` utilizando un procesador de contexto de plantilla. Para deshabilitar ese comportamiento y usar su conjunto de procesadores personalizados `add_context_processor=False` en su constructor `LoginManager` .

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

- En el código de Python:

  La extensión proporciona un objeto vinculado a la solicitud llamado `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!'
  ```

- Limitar el acceso rápidamente usando un decorador Un decorador `login_required` para iniciar `login_required` puede usarse para limitar el acceso rápidamente.

  ```
  from flask_login import login_required

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

Desconectando usuarios

Los usuarios pueden `logout_user()` sesión llamando a `logout_user()` . Parece que es seguro hacerlo incluso si el usuario no ha iniciado sesión, por lo que `@login_required` muy probable que el decorador `@login_required` pueda ser omitido.

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

¿Qué sucede si un usuario no ha iniciado

sesión y `current_user` objeto `current_user` ?

Por defecto se devuelve un AnonymousUserMixin :

- `is_active` y `is_authenticated` son `False`
- `is_anonymous` es `True`
- `get_id()` devuelve `None`

Para usar un objeto diferente para los usuarios anónimos, proporcione una llamada (ya sea una función de clase o de fábrica) que crea usuarios anónimos en su `LoginManager` con:

```
login_manager.anonymous_user = MyAnonymousUser
```

¿Qué sigue?

Con esto concluye la introducción básica a la extensión. Para obtener más información sobre la configuración y las opciones adicionales, se recomienda encarecidamente leer la guía oficial .

Temporizar la sesión de inicio de sesión

Es una buena práctica cerrar la sesión de sesión después de un tiempo específico, puede lograrlo 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 duración predeterminada de la sesión es de 31 días, el usuario debe especificar la vista de actualización de inicio de sesión en caso de que se agote el tiempo de espera.

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

La línea anterior obligará al usuario a volver a iniciar sesión cada 5 minutos.

Lea Autorización y autenticación en línea: https://riptutorial.com/es/flask/topic/9053/autorizacion-y-autenticacion

Capítulo 5: Cargas de archivos

Sintaxis

- request.files ['name'] # único archivo requerido
- request.files.get ('nombre') # Ninguno si no está publicado
- request.files.getlist ('nombre') # lista de cero o más archivos publicados
- CombinedMultiDict ((request.files, request.form)) # combina datos de formulario y archivo

Examples

Cargando archivos

Formulario HTML

- Utilice una entrada de tipo de `file` y el navegador proporcionará un campo que le permite al usuario seleccionar un archivo para cargar.
- Sólo forma con el `post` método puede enviar datos de archivo.
- Asegúrese de establecer el `enctype=multipart/form-data` . De lo contrario, se enviará el nombre del archivo, pero no los datos del archivo.
- Use el atributo `multiple` en la entrada para permitir la selección de múltiples archivos para el campo único.

```
<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>
```

Solicitudes de Python

Requests es una potente biblioteca de Python para realizar solicitudes HTTP. Puede usarlo (u otras herramientas) para publicar archivos sin un navegador.

- Abre los archivos para leer en modo binario.
- Hay múltiples estructuras de datos que los `files` toman. Esto demuestra una lista de tuplas `(name, data)` , que permite múltiples archivos como el formulario anterior.

```
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)
```

Esto no pretende ser una lista exhaustiva. Para ver ejemplos de uso de su herramienta favorita o escenarios más complejos, consulte la documentación de esa herramienta.

Guardar subidas en el servidor

Los archivos cargados están disponibles en `request.files` , un `MultiDict` campo de asignación `MultiDict` a objetos de archivo. Use `getlist` , en lugar de `[]` u `get` , si se `getlist` varios archivos con el mismo nombre de campo.

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

Los objetos en `request.files` tienen un método de `save` que guarda el archivo localmente. Crear un directorio común para guardar los archivos.

El atributo de `filename` es el nombre con el que se cargó el archivo. Esto puede ser establecido arbitrariamente por el cliente, así que páselo a través del método `secure_filename` para generar un nombre válido y seguro para guardarlo. Esto no garantiza que el nombre sea *único* , por lo que los archivos existentes se sobrescribirán a menos que haga un trabajo extra para detectar eso.

```
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')
```

Pasando datos a WTForms y Flask-WTF

WTForms proporciona un `FileField` para representar una entrada de tipo de archivo. No hace

nada especial con los datos cargados. Sin embargo, dado que Flask divide los datos del formulario (`request.form`) y los datos del archivo (`request.files`), debe asegurarse de pasar los datos correctos al crear el formulario. Puede usar un `CombinedMultiDict` para combinar los dos en una única estructura que WTForms entiende.

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

Si está utilizando Flask-WTF , una extensión para integrar Flask y WTForms, el manejo de los datos correctos se realizará de manera automática.

Debido a un error en WTForms, solo un archivo estará presente para cada campo, incluso si se cargaron múltiples. Vea este tema para más detalles. Se arreglará en 3.0.

PARSE CSV ARCHIVO DE CARGA COMO LISTA DE DICCIONARIOS EN FLASK SIN AHORRO

Los desarrolladores a menudo necesitan diseñar sitios web que permitan a los usuarios cargar un archivo CSV. Por lo general, no **hay razón** para **guardar** el archivo CSV real, ya que los datos se procesarán y / o almacenarán en una base de datos una vez que se carguen. Sin embargo, muchos, si no la mayoría, de los métodos PYTHON de análisis de datos CSV requieren que los datos se lean como un archivo. Esto puede presentar un poco de dolor de cabeza si está utilizando **FLASK** para el desarrollo web.

Supongamos que nuestro CSV tiene una fila de encabezado y se parece a lo siguiente:

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

Ahora, supongamos que el formulario html para cargar un archivo es el siguiente:

```
<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>
```

Como nadie quiere reinventar la rueda, usted decide **IMPORTAR csv** en su script de **FLASK** . No hay garantía de que las personas suban el archivo csv con las columnas en el orden correcto. Si el archivo csv tiene una fila de encabezado, con la ayuda del método **csv.DictReader** puede leer el archivo CSV como una lista de diccionarios, con las entradas en la fila del encabezado. Sin embargo, **csv.DictReader** necesita un archivo y no acepta cadenas directamente. Puede pensar que necesita usar los métodos de **FLASK** para guardar primero el archivo cargado, obtener el nuevo nombre y ubicación del archivo, abrirlo con **csv.DictReader** y luego eliminar el archivo. Parece un poco de un desperdicio.

Afortunadamente, podemos obtener el contenido del archivo como una cadena y luego dividir la cadena por líneas terminadas. El método csv **csv.DictReader** aceptará esto como un sustituto de

un archivo. El siguiente código muestra cómo se puede lograr esto sin guardar temporalmente el archivo.

```
@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 variable **csv_dicts** es ahora la siguiente lista de diccionarios:

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

En caso de que sea nuevo en PYTHON, puede acceder a datos como los siguientes:

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

Otras soluciones implican importar el módulo **io** y usar el método **io.Stream** . Siento que este es un enfoque más directo. Creo que el código es un poco más fácil de seguir que usar el método **io** . Este enfoque es específico del ejemplo de analizar un archivo CSV cargado.

Lea Cargas de archivos en línea: https://riptutorial.com/es/flask/topic/5459/cargas-de-archivos

Capítulo 6: Enrutamiento

Examples

Rutas Básicas

Las rutas en Flask se pueden definir utilizando el decorador de `route` de la instancia de la aplicación Flask:

```
app = Flask(__name__)

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

El decorador de `route` toma una cadena que es la URL para que coincida. Cuando la aplicación recibe una solicitud de una URL que coincide con esta cadena, se invocará la función decorada (también llamada *función de vista*). Así que para una ruta tendríamos:

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

Es importante tener en cuenta que estas rutas **no** son expresiones regulares como lo son en Django.

También puede definir *reglas de variables* para extraer valores de segmento de URL en variables:

```
@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
```

Aquí la regla de la variable está en el último segmento de la URL. Cualquier valor que se encuentre en el último segmento de la URL se pasará a la función de vista (`get_blog_post`) como el parámetro `post_id` . Por lo tanto, una solicitud para `/blog/posts/42` recuperará (o intentará recuperar) la publicación del blog con un ID de 42.

También es común reutilizar las URL. Por ejemplo, tal vez queremos que `/blog/posts` devuelva una lista de todas las publicaciones de blog. Entonces podríamos tener dos rutas para la misma función de vista:

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

Tenga en cuenta que también tenemos que proporcionar el valor predeterminado de `None` para el

`post_id` en `get_blog_post` . Cuando la primera ruta coincida, no habrá ningún valor para pasar a la función de vista.

También tenga en cuenta que, de forma predeterminada, el tipo de una regla variable es una cadena. Sin embargo, puede especificar varios tipos diferentes, como `int` y `float` prefijando la variable:

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

Los convertidores de URL incorporados de Flask son:

```
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
```

Si intentamos visitar la URL `/blog/post/foo` con un valor en el último segmento de URL que no se puede convertir en un entero, la aplicación devolverá un error 404. Esta es la acción correcta porque no hay una regla con `/blog/post` y una cadena en el último segmento.

Finalmente, las rutas también pueden configurarse para aceptar métodos HTTP. El decorador de `route` toma un argumento de palabra clave de `methods` que es una lista de cadenas que representan los métodos HTTP aceptables para esta ruta. Como puede haber asumido, el valor predeterminado es `GET` solamente. Si tuviéramos un formulario para agregar una nueva publicación de blog y quisiéramos devolver el HTML para la solicitud `GET` y analizar los datos del formulario para la solicitud `POST` , la ruta se vería así:

```
@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` se encuentra en el paquete `flask` . Tenga en cuenta que al utilizar el argumento de palabras clave de `methods` , debemos ser explícitos sobre los métodos HTTP para aceptar. Si hubiéramos enumerado solo `POST` , la ruta ya no respondería a las solicitudes `GET` y devolvería un error 405.

Ruta de todo

Puede ser útil tener una vista general donde usted mismo maneje la lógica compleja basada en la ruta. Este ejemplo utiliza dos reglas: la primera regla específicamente detecta `/` y la segunda regla detecta rutas arbitrarias con el convertidor de `path` incorporado. El convertidor de `path` coincide con cualquier cadena (incluidas las barras) Ver reglas variables de matraz

```
@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/'
```

Métodos de enrutamiento y HTTP

Por defecto, las rutas solo responden a las solicitudes `GET` . Puede cambiar este comportamiento proporcionando el argumento de `methods` al decorador `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()
```

También puede asignar diferentes funciones al mismo punto final según el método HTTP utilizado.

```
@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'
```

Lea Enrutamiento en línea: https://riptutorial.com/es/flask/topic/2415/enrutamiento

Capítulo 7: Frasco en Apache con mod_wsgi

Examples

Contenedor de aplicación WSGI

Muchas aplicaciones de Flask se desarrollan en un *virtualenv* para mantener las dependencias de cada aplicación por separado de la instalación de Python en todo el sistema. Asegúrese de que *mod-wsgi* esté instalado en su *virtualenv* :

```
pip install mod-wsgi
```

Luego crea un contenedor wsgi para tu aplicación Flask. Por lo general, se mantiene en el directorio raíz de su aplicación.

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
```

Este contenedor activa el entorno virtual y todos sus módulos y dependencias instalados cuando se ejecuta desde Apache, y se asegura de que la ruta de la aplicación sea la primera en las rutas de búsqueda. Por convención, los objetos de aplicación WSGI se denominan `application` .

Configuración de sitios apache habilitados para WSGI

La ventaja de usar Apache sobre el servidor werkzeug integrado es que Apache es multiproceso, lo que significa que se pueden hacer múltiples conexiones a la aplicación simultáneamente. Esto es especialmente útil en aplicaciones que utilizan *XmlHttpRequest* (AJAX) en el front-end.

/etc/apache2/sites-available/050-my-application.conf (o configuración de apache predeterminada si no está alojado en un servidor web compartido)

```
<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>
```

Lea Frasco en Apache con mod_wsgi en línea: https://riptutorial.com/es/flask/topic/6851/frasco-en-apache-con-mod-wsgi

Capítulo 8: Frasco-SQLAlchemy

Introducción

Flask-SQLAlchemy es una extensión de Flask que agrega soporte para el popular mapeador relacional de objetos (ORM) Python (ORM) SQLAlchemy a las aplicaciones de Flask. Su objetivo es simplificar SQLAlchemy con Flask proporcionando algunas implementaciones predeterminadas para tareas comunes.

Examples

Instalación y ejemplo inicial

Instalación

```
pip install Flask-SQLAlchemy
```

Modelo simple

```
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)
```

El ejemplo de código anterior muestra un modelo simple de Flask-SQLAlchemy, podemos agregar un nombre de tabla opcional a la declaración del modelo; sin embargo, a menudo no es necesario, ya que Flask-SQLAlchemy usará automáticamente el nombre de la clase como nombre de la tabla durante la creación de la base de datos.

Nuestra clase heredará del modelo de clase base, que es una base declarativa configurada, por lo tanto, no es necesario que definamos explícitamente la base como lo haríamos al usar SQLAlchemy.

Referencia

- URL de Pypi: [https://pypi.python.org/pypi/Flask-SQLAlchemy] [1]
- URL de la documentación: [http://flask-sqlalchemy.pocoo.org/2.1/◆◆1]

Relaciones: uno a muchos

```
    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')
```

En este ejemplo, tenemos dos clases: la clase Usuario y la clase Publicación, la clase Usuario será nuestra principal y la Publicación será nuestra publicación ya que solo la publicación puede pertenecer a un usuario, pero un usuario puede tener varias publicaciones. Para lograr eso, colocamos una clave externa en el hijo que hace referencia al padre que es de nuestro ejemplo, colocamos una clave externa en la clase Post para hacer referencia a la clase de usuario. Luego usamos `relationship()` en el padre al que accedemos a través de nuestro `db` SQLAlchemy object. Eso nos permite hacer referencia a una colección de elementos representados por la clase Post que es nuestro hijo.

Para crear una relación bidireccional, usamos `backref` , esto le permitirá al niño hacer referencia al padre.

Lea Frasco-SQLAlchemy en línea: https://riptutorial.com/es/flask/topic/10577/frasco-sqlalchemy

Capítulo 9: Frasco-WTF

Introducción

Es una integración simple de Flask y WTForms. Permite la creación y administración más sencilla de formularios web, genera automáticamente un campo oculto de token CRSF en sus plantillas. También cuenta con funciones de validación de forma fácil

Examples

Una forma simple

```
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()])
```

Para renderizar la plantilla usarás algo como esto:

```
<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>
```

El código simple anterior generará nuestro formulario web flask-wtf muy simple con un campo de token CRSF oculto.

Lea Frasco-WTF en línea: https://riptutorial.com/es/flask/topic/10579/frasco-wtf

Capítulo 10: Implementando la aplicación Flask usando el servidor web uWSGI con Nginx

Examples

Usando uWSGI para ejecutar una aplicación matraz

El servidor `werkzeug` incorporado ciertamente no es adecuado para ejecutar servidores de producción. La razón más obvia es el hecho de que el servidor `werkzeug` es de un solo hilo y, por lo tanto, solo puede manejar una solicitud a la vez.

Debido a esto, queremos usar el servidor uWSGI para servir nuestra aplicación en su lugar. En este ejemplo instalaremos uWSGI y ejecutaremos una aplicación de prueba simple con él.

Instalando uWSGI :

```
pip install uwsgi
```

Es tan simple como eso. Si no está seguro de la versión de python que utiliza su pip, hágala explícita:

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

Ahora vamos a crear una aplicación de prueba simple:

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
```

En el matraz, el nombre convencional para la aplicación es `app` pero uWSGI busca la `application` por defecto. Es por eso que creamos un alias para nuestra aplicación en la última línea.

Ahora es el momento de ejecutar la aplicación:

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

Debería ver el mensaje "Hola uWSGI ..." apuntando su navegador a `localhost:5000`

Para no escribir el comando completo cada vez que creamos un archivo `uwsgi.ini` para almacenar esa configuración:

uwsgi.ini

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

Las opciones de `wsgi-file` `http` y `wsgi-file` son las mismas que en el comando manual. Pero hay tres opciones más:

- `single-interpreter` : se recomienda activar esto porque podría interferir con la siguiente opción
- `enable-threads` : esto debe estar activado si está utilizando subprocesos adicionales en su aplicación. No los usamos ahora, pero ahora no tenemos que preocuparnos por eso.
- `master` : el modo maestro debe habilitarse por varias razones

Ahora podemos ejecutar la aplicación con este comando:

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

Instalando nginx y configurándolo para uWSGI

Ahora queremos instalar nginx para servir nuestra aplicación.

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

Luego creamos una configuración para nuestro sitio 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;
    }
}
```

Esto le indica a nginx que escuche en el puerto 80 (predeterminado para http) y sirva algo en la ruta raíz (`/`). Allí le decimos a nginx que simplemente actúe como un proxy y pase cada solicitud a un socket llamado `flask.sock` ubicado en `/tmp/` .

Vamos a habilitar el sitio:

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

Es posible que desee eliminar la configuración predeterminada si está habilitada:

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

Luego reinicie nginx:

```
sudo service nginx restart
```

Apunte su navegador a `localhost` y verá un error: `502 Bad Gateway` .

Esto significa que nginx está activo y funcionando pero falta el zócalo. Así que vamos a crear eso.

Vuelva a su archivo `uwsgi.ini` y ábralo. Luego agrega estas líneas:

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

La primera línea le dice a uwsgi que cree un socket en la ubicación dada. El socket se utilizará para recibir solicitudes y enviar las respuestas. En la última línea permitimos que otros usuarios (incluido nginx) puedan leer y escribir desde ese socket.

Vuelva a iniciar uwsgi con `uwsgi --ini uwsgi.ini` . Ahora apunte su navegador de nuevo a `localhost` y verá nuevamente el saludo "Hola, uWSGI".

Tenga en cuenta que todavía puede ver la respuesta en `localhost:5000` porque uWSGI ahora sirve la aplicación a través de http **y** el socket. Así que vamos a deshabilitar la opción http en el archivo ini

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

Ahora solo se puede acceder a la aplicación desde nginx (o leer ese socket directamente :)).

Habilitar la transmisión desde el matraz

Flask tiene esa característica que le permite transmitir datos desde una vista mediante el uso de generadores.

Vamos a cambiar el archivo `app.py`

- añadir `from flask import Response`
- agregar `from datetime import datetime`
- añadir `from time import sleep`
- crear una nueva vista:

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

    return Response(streamer())
```

Ahora abra su navegador en `localhost/time/` . El sitio se cargará para siempre porque nginx espera hasta que se complete la respuesta. En este caso, la respuesta nunca estará completa porque enviará la fecha y hora actuales para siempre.

Para evitar que nginx espere, debemos agregar una nueva línea a la configuración.

Edite `/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 línea `uwsgi_buffering off;` le dice a nginx que no espere hasta que se complete una respuesta.

Reinicie nginx: `sudo service nginx restart` y observe `localhost/time/` again.

Ahora verás que a cada segundo aparece una nueva línea.

Configure la aplicación Flask, uWGSI, Nginx - Plantilla de caldera de configuraciones de servidor (predeterminado, proxy y caché)

Esta es una parte de la configuración que se obtiene del tutorial de DigitalOcean de *Cómo Servir las Aplicaciones de Flask con uWSGI y Nginx en Ubuntu 14.04*

y algunos recursos git útiles para servidores nginx.

Aplicación del matraz

Este tutorial asume que usas Ubuntu.

1. busque `var/www/` folder.

2. Crea tu carpeta de aplicaciones web `mkdir myexample`
3. `cd myexample`

opcional Es posible que desee configurar un entorno virtual para implementar aplicaciones web en el servidor de producción.

```
sudo pip install virtualenv
```

para instalar el entorno virtual.

```
virtualenv myexample
```

para configurar el entorno virtual para su aplicación.

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

para activar su entorno. Aquí instalarás todos los paquetes de python.

final opcional pero recomendado

Configurar frasco y puerta de enlace uWSGI

Instalar matraz y puerta de enlace uSWGI:

```
pip install uwsgi flask
```

Ejemplo de aplicación matraz en 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')
```

Cree un archivo para comunicarse entre su aplicación web y el servidor web: interfaz de puerta de enlace [https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface]

```
nano wsgi.py
```

luego importe su módulo de aplicación web y hágalo funcionar desde el punto de entrada de la puerta de enlace.

```
from myexample import application

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

Para probar uWSGI:

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

Para configurar uWSGI:

1. Crear un archivo de configuración `.ini`

   ```
   nano myexample.ini
   ```

2. Configuración básica para 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
```

Opcional si está usando env virtual . Puede `deactivate` su entorno virtual.

Configuración de Nginx Vamos a utilizar nginx como:

1. servidor predeterminado para pasar la solicitud al socket, utilizando el protocolo uwsgi
2. servidor proxy delante del servidor predeterminado
3. servidor de caché para almacenar en caché las solicitudes exitosas (como ejemplo, es posible que desee almacenar en caché las solicitudes GET si su aplicación web)

Localice su directorio de `sites-available` y cree un archivo de configuración para su aplicación:

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

Agregue el siguiente bloque, en comentarios lo que hace:

```
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;
    }
}
```

Finalmente, vincule el archivo al directorio de `sites-enabled` . Para obtener una explicación de los sitios disponibles y habilitados, consulte la respuesta: [http://serverfault.com/a/527644]

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

Ya has terminado con nginx. Sin embargo, es posible que desee revisar esta preciosa plantilla de caldera: [https://github.com/h5bp/server-configs-nginx]

Muy útil para el ajuste fino.

Ahora prueba Nginx:

```
sudo nginx -t
```

Lanzar Nginx:

```
sudo service nginx restart
```

Automatice Ubuntu para iniciar uWSGI Lo último es hacer que Ubuntu inicie la puerta de enlace wsgi comunicándose con su aplicación, de lo contrario debería hacerlo manualmente.

1. Localice el directorio para los scripts de inicialización en Ubuntu y cree un nuevo script:

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

2. Agregue el siguiente bloque, comentarios en línea para explicar lo que hace

```
# 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
```

Ahora puedes activar tu script: sudo start myexample

Lea Implementando la aplicación Flask usando el servidor web uWSGI con Nginx en línea: https://riptutorial.com/es/flask/topic/4637/implementando-la-aplicacion-flask-usando-el-servidor-

web-uwsgi-con-nginx

Capítulo 11: Mensaje intermitente

Introducción

Mensaje intermitente a la plantilla mediante función `flash()` .

Sintaxis

- flash (mensaje, categoría = 'mensaje')
- flash ('hola mundo!')
- flash ('Este es un mensaje de advertencia', 'advertencia')

Parámetros

mensaje	el mensaje a ser flasheado.
categoría	La categoría del mensaje, el predeterminado es el `message` .

Observaciones

- Herencia de plantillas
- API

Examples

Mensaje simple intermitente

Configure `SECKET_KEY` y luego el mensaje parpadeante en la función de visualización:

```
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')
```

Luego, renderice los mensajes en `layout.html` (que se extendió desde `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 %}
```

Intermitente con categorías

Establezca el segundo argumento cuando use `flash()` en la función de visualización:

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

En la plantilla, establezca `with_categories=true` en `get_flashed_messages()` , luego obtendrá una lista de tuplas en forma de `(message, category)` , por lo que puede usar la categoría como una clase 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 %}
```

Lea Mensaje intermitente en línea: https://riptutorial.com/es/flask/topic/10756/mensaje-intermitente

Capítulo 12: Paginación

Examples

Ejemplo de ruta de paginación con matraz-sqlalchemy Paginate

En este ejemplo, utilizamos un parámetro en la ruta para especificar el número de página. Establecemos un valor predeterminado de 1 en la `page=1` parámetros de función `page=1` . Tenemos un objeto `User` en la base de datos y lo consultamos, ordenando en orden descendente, mostrando primero a los usuarios más recientes. Luego usamos el método de `paginate` del objeto de `query` en flask-sqlalchemy. Luego pasamos esto a `render_template` para ser renderizado.

```
@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
    )
```

Paginación de render en jinja

Aquí utilizamos el objeto que pasamos a `render_template` para mostrar las páginas, la página activa actual y también los botones anterior y siguiente si puede ir a la página anterior / siguiente.

```
<!-- 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 %}
```

Lea Paginación en línea: https://riptutorial.com/es/flask/topic/6460/paginacion

Capítulo 13: Personalizado Jinja2 Filtros De Plantilla

Sintaxis

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

Parámetros

Parámetro	Detalles
valor	El valor pasado por Jinja, para ser filtrado
args	Argumentos extra para pasar a la función de filtro

Examples

Formato de fecha y hora en una plantilla de Jinja2.

Los filtros pueden definirse en un método y luego agregarse al diccionario de filtros de Jinja, o definirse en un método decorado con `Flask.template_filter` .

Definición y registro posterior:

```
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
```

Definiendo con decorador:

```
@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)
```

Lea Personalizado Jinja2 Filtros De Plantilla en línea:
https://riptutorial.com/es/flask/topic/1465/personalizado-jinja2-filtros-de-plantilla

Capítulo 14: Planos

Introducción

Los planos son un concepto poderoso en el desarrollo de aplicaciones Flask que permite que las aplicaciones Flask sean más modulares y puedan seguir múltiples patrones. Facilitan la administración de aplicaciones de matraz muy grandes y, como tal, se pueden usar para escalar aplicaciones de matraz. Puede reutilizar las aplicaciones Blueprint; sin embargo, no puede ejecutar un modelo solo, ya que debe estar registrado en su aplicación principal.

Examples

Un ejemplo básico de planos de matraz.

Una aplicación de matraz mínima se ve algo como esto:

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

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

Una aplicación grande de Flask puede separar un archivo en varios archivos por `blueprints` *.*

Propósito

Facilita a los demás el mantenimiento de la aplicación.

Estructura de carpetas de gran aplicación

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

vistas / 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)
```

Ejecutar aplicación

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

Lea Planos en línea: https://riptutorial.com/es/flask/topic/6427/planos

Capítulo 15: Plantillas de renderizado

Sintaxis

- `render_template(template_name_or_list, **context)`

Examples

Uso de render_template

Flask te permite usar plantillas para el contenido dinámico de páginas web. Una estructura de proyecto de ejemplo para usar plantillas es la siguiente:

```
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")
```

Tenga en cuenta que puede pasar contenido dinámico de su controlador de ruta a la plantilla agregando pares clave / valor a la función render_templates. En el ejemplo anterior, las variables "pagetitle" y "mycontent" se pasarán a la plantilla para su inclusión en la página representada. Incluya estas variables en la plantilla encerrándolas entre llaves dobles: `{{mytitle}}`

`index.html` :

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

Cuando se ejecute igual que en el primer ejemplo, `http://localhost:5000/` tendrá el título

"HomePage" y un párrafo con el contenido "Hello World".

Lea Plantillas de renderizado en línea: https://riptutorial.com/es/flask/topic/1641/plantillas-de-renderizado

Capítulo 16: Pruebas

Examples

Probando nuestra aplicación Hello World

Introducción

En este ejemplo minimalista, usando `pytest` vamos a probar que, de hecho, nuestra aplicación Hello World devuelve "Hello, World!" con un código de estado HTTP OK de 200, cuando se encuentra con una solicitud GET en la URL `/`

Primero `pytest` en nuestro virtualenv

```
pip install pytest
```

Y solo para referencia, esta es nuestra aplicación hola mundo:

```
# hello.py
from flask import Flask

app = Flask(__name__)

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

Definiendo la prueba

Junto a nuestro `hello.py` , definimos un módulo de prueba llamado `test_hello.py` que será descubierto por `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!'
```

Solo para revisar, en este punto nuestra estructura de proyecto obtenida con el comando de `tree` es:

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

Corriendo la prueba

Ahora podemos ejecutar esta prueba con el comando `py.test` que descubrirá automáticamente nuestro `test_hello.py` y la función de prueba en su interior.

```
$ py.test
```

Debería ver algunos resultados y una indicación de que ha pasado 1 prueba, por ejemplo,

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

Probando una API JSON implementada en Flask

Este ejemplo asume que sabe cómo probar una aplicación Flask usando pytest

A continuación se muestra una API que toma una entrada JSON con los valores enteros `a` y `b` por ejemplo, `{"a": 1, "b": 2}` , los suma y devuelve la suma `a + b` en una respuesta JSON, por ejemplo, `{"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']})
```

Probando esta API con `pytest`

Podemos probarlo 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
```

Ahora ejecuta la prueba con el comando `py.test` .

Acceso y manipulación de las variables de sesión en sus pruebas utilizando Flask-Testing

La mayoría de las aplicaciones web utilizan el objeto de sesión para almacenar información importante. Estos ejemplos muestran cómo puede probar dicha aplicación utilizando Flask-Testing. El ejemplo completo de trabajo también está disponible en github .

Así que primero instala Flask-Testing en tu virtualenv

```
pip install flask_testing
```

Para poder usar el objeto de sesión, debes configurar la clave secreta

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

Imaginemos que tiene en su función de aplicación que necesita almacenar algunos datos en variables de sesión como esta

```
@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'
```

Para probar esta función, puede importar flask_testing y dejar que su clase de prueba herede flask_testing.TestCase. Importar también todas las bibliotecas necesarias.

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

class TestMyApp(flask_testing.TestCase):
```

Es muy importante antes de comenzar la prueba implementar la función **create_app, de lo** contrario habrá una excepción.

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

Para probar tu aplicación está funcionando como lo deseas, tienes un par de posibilidades. Si solo quiere asegurarse de que su función está configurando valores particulares para una variable de sesión, puede mantener el contexto y acceder 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 truco más útil es diferenciar entre los métodos *GET* y *POST* como en la siguiente función de prueba

```
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')
```

Ahora imagine que su función espera que una variable de sesión se establezca y reacciona de manera diferente en valores particulares como este

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

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

Para probar esta función, debe utilizar la llamada *transacción de sesión* y abrir la sesión en el contexto del cliente de prueba. Esta función está disponible desde **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')
```

La ejecución de las pruebas es como siempre para unittest

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

Y en la linea de comando.

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

Otra buena manera de ejecutar tus pruebas es usar Unittest Discovery de esta manera:

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

Lea Pruebas en línea: https://riptutorial.com/es/flask/topic/1260/pruebas

Capítulo 17: Redirigir

Sintaxis

- redireccionar (ubicación, código, respuesta)

Parámetros

Parámetro	Detalles
ubicación	La ubicación a la que se debe redireccionar la respuesta.
código	(Opcional) El código de estado de redireccionamiento, 302 por defecto. Los códigos admitidos son 301, 302, 303, 305 y 307.
Respuesta	(Opcional) Una clase de Respuesta para usar cuando se instancia una respuesta. El valor predeterminado es werkzeug.wrappers.Response si no se especifica.

Observaciones

El parámetro de ubicación debe ser una URL. Puede ser de entrada en bruto, como ' http://www.webpage.com ' o puede construirse con la función url_for ().

Examples

Ejemplo simple

```
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'))
```

Transmitiendo datos

```
# ...
# 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')
```

Lea Redirigir en línea: https://riptutorial.com/es/flask/topic/6856/redirigir

Capítulo 18: Señales

Observaciones

Frasco soporta señales utilizando Blinker . El soporte de señal es opcional; solo se habilitarán si Blinker está instalado.

```
pip install blinker
```

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

Las señales no son asíncronas. Cuando se envía una señal, ejecuta de forma secuencial cada una de las funciones conectadas.

Examples

Conectando a señales

Utilice una señal de `connect` método para conectar una función a una señal. Cuando se envía una señal, cada función conectada se llama con el remitente y los argumentos con nombre que proporciona la señal.

```
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)
```

Consulte la documentación sobre señales incorporadas para obtener información sobre los argumentos que proporcionan. Un patrón útil es agregar un `**kwargs` argumento `**kwargs` para capturar cualquier argumento inesperado.

Señales personalizadas

Si desea crear y enviar señales en su propio código (por ejemplo, si está escribiendo una extensión), cree una nueva instancia de `Signal` y `send` llamada cuando se deba notificar a los suscriptores. Las señales se crean utilizando un `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):
    ...
```

Prefiere usar el soporte de señal de Flask en lugar de usar Blinker directamente. Envuelve la biblioteca para que las señales sigan siendo opcionales si los desarrolladores que usan su extensión no han optado por instalar Blinker.

Lea Señales en línea: https://riptutorial.com/es/flask/topic/2331/senales

Capítulo 19: Sesiones

Observaciones

Las sesiones se derivan de los diccionarios, lo que significa que funcionarán con los métodos de diccionario más comunes.

Examples

Usando el objeto sesiones dentro de una vista

En primer lugar, asegúrese de que ha importado sesiones de matraz

```
from flask import session
```

Para usar la sesión, una aplicación Flask necesita un **SECRET_KEY** definido.

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

Las sesiones se implementan de forma predeterminada mediante una **cookie** firmada con la clave secreta. Esto garantiza que los datos no se modifiquen excepto por su aplicación, ¡así que asegúrese de elegir uno seguro! Un navegador enviará la cookie a su aplicación junto con cada solicitud, permitiendo la persistencia de los datos en todas las solicitudes.

Para usar una sesión, simplemente haga referencia al objeto (se comportará como un diccionario)

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

Para liberar una variable de sesión usa **el** método **pop ()** .

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

Código de ejemplo:

```
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()
```

Lea Sesiones en línea: https://riptutorial.com/es/flask/topic/2748/sesiones

Capítulo 20: Trabajando con JSON

Examples

Devolver una respuesta JSON desde la API de Flask

Flask tiene una utilidad llamada `jsonify()` que hace que sea más conveniente devolver respuestas 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"}
```

Pruébalo con `curl`

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

Otras formas de usar `jsonify()`

Usando un diccionario existente:

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

Usando una lista:

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

Recibiendo JSON de una solicitud HTTP

Si el mimetype de la solicitud HTTP es `application/json` , llamar a `request.get_json()` devolverá los datos JSON analizados (de lo contrario, devuelve `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)
```

Pruébalo con `curl`

El parámetro `-H 'Content-Type: application/json'` especifica que se trata de una solicitud JSON:

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

Para enviar solicitudes utilizando otros métodos HTTP, sustituya `curl -X POST` con el método deseado, por ejemplo, `curl -X GET` , `curl -X PUT` , etc.

Lea Trabajando con JSON en línea: https://riptutorial.com/es/flask/topic/1789/trabajando-con-json

Capítulo 21: Vistas basadas en clase

Examples

Ejemplo basico

Con vistas basadas en clases, usamos clases en lugar de métodos para implementar nuestras vistas. Un ejemplo simple de usar vistas basadas en clase se ve como sigue:

```
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)
```

Lea Vistas basadas en clase en línea: https://riptutorial.com/es/flask/topic/7494/vistas-basadas-en-clase

Creditos

S. No	Capítulos	Contributors
1	Comenzando con el matraz	arsho, bakkal, Community, davidism, ettanany, Martijn Pieters, mmenschig, Sean Vieira, Shrike
2	Acceso a los datos de solicitud	RPi Awesomeness
3	Archivos estáticos	arsho, davidism, MikeC, YellowShark
4	Autorización y autenticación	boreq, Ninad Mhatre
5	Cargas de archivos	davidism, sigmasum
6	Enrutamiento	davidism, Douglas Starnes, Grey Li, junnytony, Luke Taylor, MikeC, mmenschig, sytech
7	Frasco en Apache con mod_wsgi	Aaron D
8	Frasco-SQLAlchemy	Achim Munene, arsho, Matt Davis
9	Frasco-WTF	Achim Munene
10	Implementando la aplicación Flask usando el servidor web uWSGI con Nginx	Gal Dreiman, Tempux, user305883, wimkeir, Wombatz
11	Mensaje intermitente	Grey Li
12	Paginación	hdbuster
13	Personalizado Jinja2 Filtros De Plantilla	Celeo, dylanj.nz
14	Planos	Achim Munene, Kir Chou, stamaimer
15	Plantillas de renderizado	arsho, atayenel, Celeo, fabioqcorreia, Jon Chan, MikeC
16	Pruebas	bakkal, oggo

17	Redirigir	coralvanda
18	Señales	davidism
19	Sesiones	arsho, PsyKzz, this-vidor
20	Trabajando con JSON	bakkal, g3rv4
21	Vistas basadas en clase	ettanany