

Servicios Telemáticos

Introducción a Flask



FLASK

- Microframework basado en Python
- Depende de dos librerías:
 - **Jinja2**: para generar plantillas
 - **Werkzeug**: es un WSGI (Web Server Gateway Interface) para Python que gestionar el diálogo la aplicación con el servidor web

Instalación – entorno virtual

- Se recomienda el uso de un entorno virtual (*virtual environmet*)
 - Aísla la configuración para Flask de otras configuraciones
- <http://flask.pocoo.org/docs/0.10/installation/>

```
$ sudo apt-get install python-dev  
$ sudo apt-get install python-virtualenv  
$ mkdir iroom  
$ cd iroom  
$ virtualenv flask  
$ . flask/bin/activate (ojo al principio es un punto y un espacio en blanco)  
$ pip install flask  
$ pip install flask-script
```

Con esto, crea un entorno virtual (llamado flask) e instala el flash y las extensiones flask-script

Cuando se apague la maquina virtual para volver a ejecutar el entorno ejecutar la línea de flask/bin/activate

Aplicación básica “Hello world”

- Cambiar algo del “Hello World!” del fichero en el server (Maq Virtual):

```
$ vi helloflask.py
```

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

@app.route("/")
def hello():
    return "Hello World!"
if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

- Averiguar la IP de la Maq virtual

```
$ ip a
```

- Arrancar la aplicación servidor

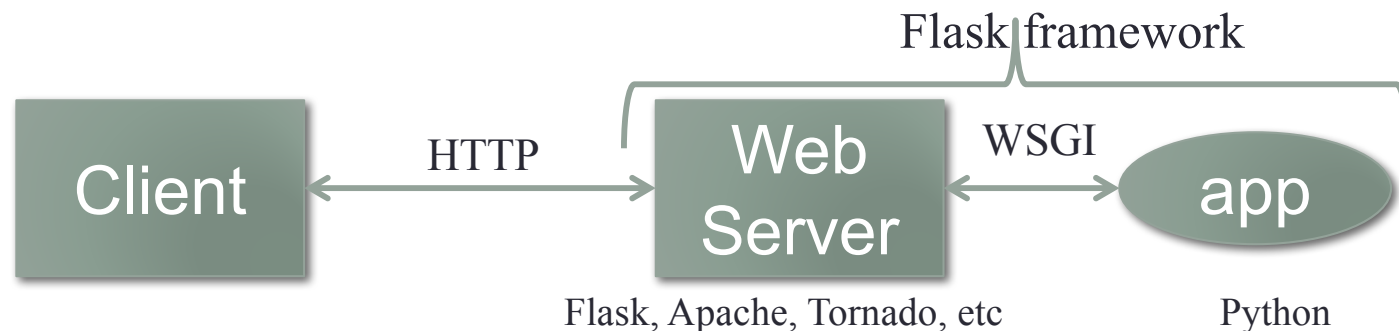
```
$ python helloflask.py
```

- En el PC real en un navegador <http://IP-Maq-virtual:5000> y aparecerá una página web con el mensaje “Hello World!”
- Para parar el servidor con Ctrl-C
- Ver enlace para mas detalles, <https://www.enigmaeta.com/blog/starting-flask/>

Aplicación básica: Inicialización

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

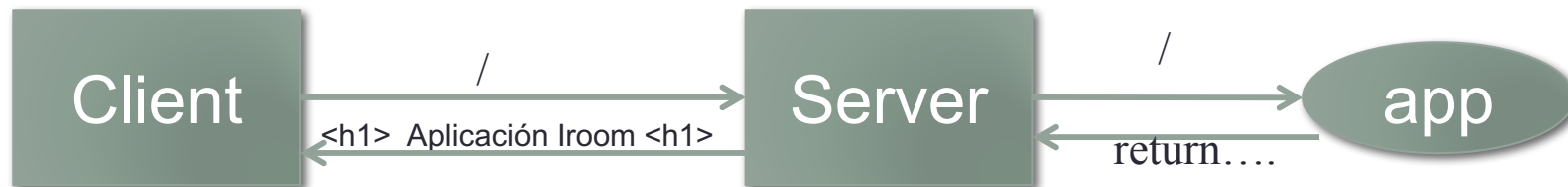
- Primero importamos la clase Flask
- Se crea una instancia de esa clase, con el valor `__name__` indicamos que la aplicación sólo tiene un módulo
- El servidor Web pasa todas las peticiones de clientes a la instancia.
- Usa el protocolo WSGI en esta comunicación.



Aplicación básica: Rutas y vistas

- Ruta (route): asociación entre **URL** solicitada y la **función** que la procesa
- Ejemplo:
 - Si solicitan el recurso **'/'** llama a la función **index()**
 - La función **index** devuelve (return), un código html al cliente

```
@app.route('/')  
def index():  
    return '<h1> Aplicación Iroom <h1>'  
  
@app.route('/loc')  
def location():  
    return '<p> Ubicación EL10 <p>'
```



- <http://runnable.com/Uh4qRmSwz8cHAAAN/how-to-perform-advanced-routing-in-flask-for-python-and-routes>

Aplicación básica: Inicio del servidor

- El método **run** inicia la aplicación.
- Indicamos con *host* que puede recibir **peticiones por cualquier interfaz** (por defecto sólo acepta peticiones locales)
- El atributo *debug* se puede poner a valor True en la **fase de desarrollo**
 - Por motivos de seguridad debe ser desactivado en la **fase de producción**

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

- Para entender como se conecta la VM con la real y con Internet, echar un vistazo a <https://rednectar.net/2011/07/20/vmware-interfaces-tutorial/>

Aplicación básica

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return '<h1> Aplicación Iroom </h1>'

@app.route('/loc')
def location():
    return '<p> UAH.EPS.EL10 </p>'

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

Arrancamos la aplicación
\$python iroom.py

Arranca por defecto en el puerto 5000

Accedemos desde un navegador:
<http://192.168.42.132:5000/>

Nota: ¿debemos terminar con slash "/"?

`app.route(/loc/)`
Si la petición es `http://server:5000/loc` el servidor la devuelve igualmente.

`app.route(/loc)`
Si la petición es `http://server:5000/loc/` el servidor devuelve error *404 Not found*

añadiendo variables a la URL

- Se usa la sintaxis: `<conversor:nombre_variable>`
 - Por defecto la variable se considera como **string**: `<name>`
 - Si la variable es de **otro tipo**, se usa un conversor: `<int:value>`
 - Tipos de conversores: int, float, path (string con '/')

```
@app.route('/desc/<sensor>')
def show_type(sensor):
    return 'Tipo de sensor: %s' % sensor

@app.route('/vai/<int:val_i>')
def show_value_i(val_i):
    return 'Valor del sensor: %d' % val_i

@app.route('/vaf/<float:val_f>')
def show_value_f(val_f):
    return 'Valor del sensor: %f' % val_f
```

Construcción de URLs

- Usando el método **url_for**
- Genera la URL del recurso
- Puede ser utilizado para generación dinámica

```
from flask import Flask, url_for

@app.route('/')
def index():
    return '<h1> Aplicación Iroom </h1>'

@app.route('/loc')
def location():
    return 'UAH.EPS.EL10'

@app.route('/desc/<sensor>')
def type(sensor):
    return 'Tipo de sensor: %s' % sensor

url_for('index')

url_for('location')

url_for('type', sensor='temp')

url_for('type', sensor=sensor) ???
```

Ficheros estáticos

- Hay ficheros que serán estáticos en la aplicación: css, js, imágenes
- Crear una carpeta denominada *static* para ubicarlos.
- Por ejemplo static puede tener: *style.css*, *iroom.jpeg*
- Para generar la URL (*static/style.css*) podemos usar `url_for()`:

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

Genera: *“/static/style.css”*

REQUEST

- Las peticiones (request) soportan los métodos HTTP: GET, HEAD, POST, PUT, DELETE, OPTIONS
- Importamos el **módulo request**
- El método usado por defecto es GET, pero se puede indicar otros en el método 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()
```

CONTEXTO DE UN REQUEST

- Capa petición tiene asociadas unas variables de entorno que se pueden consultar:
 - **Request:** contiene información HTTP de una petición.
 - Ejemplo: `user_agent = request.headers.get('User-Agent')`
Guarda en `user_agent` el valor de la cabecera de la petición HTTP
 - **Session:** contiene información asociada a una sesión, que puede ser enviada en las peticiones para refrescar valores de la sesión.
 - **g:** objeto que contiene información local que la aplicación almacena temporalmente durante la gestión de peticiones.

request: recogemos información de POST

- Un cliente puede enviar información (formulario) usando el método GET o POST.
- **request.form** recoge la información si se usa POST.
- La página que se devuelve, color.html, tiene una variable que se llama

color

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from flask import Flask, render_template, request
app = Flask(__name__)
@app.route('/led', methods=['POST'])
def led():
    error = None
    if request.method == 'POST':
        color = request.form['color']
        if valid_color(color):
            change_color(color)
        else:
            error = 'Invalid color'
    return render_template('color.html', color=color, error=error)
if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

Formulario con botón de envío

- my-form.html

```
<!DOCTYPE html>
<html lang="en">
<body>
  <h1>Enter some text</h1>
  <h2>(it will be converted to uppercase)</h2>
  <form action="." method="POST">
    <input type="text" name="text">
    <input type="submit" name="my-form" value="Send">
  </form>
</body>
</html>
```

- prog.py

```
#cabecera transparencia anterior
@app.route('/')
def my_form():
    return render_template("my-form.html")
@app.route('/', methods=['POST'])
def my_form_post():
    text = request.form['text']
    processed_text = text.upper()
    return processed_text
if __name__ == '__main__':
    app.run()
```

request: recogemos información de GET

- **request.args.get** recoge la información si se usa GET

```
@app.route("/submit", methods=['GET'])
def submit():
    nombre = request.args.get('Nombre', 'Anonymous')
    return render_template('submit_result.html', nombre=nombre, metodo=request.method)
```

- Anonymous es un valor por defecto que asume si no se envía Nombre.

REQUEST HOOKS

- Mediante hooks de request podemos definir funciones que se van a ejecutar siempre que se den los siguientes eventos referentes a request.
 - **before_first_request:** la primera vez que llega una petición, antes de gestionarla.
 - **before_request:** cuando llega una petición.
 - **after_request:** después de una petición.
 - **teardown_request:** después de una petición, incluso si se produce una excepción.

RESPONSE

- Podemos generar respuestas definiendo diferentes atributos de las mismas:
- Para ello podemos usar el módulo `make_response`

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from flask import make_response

app = Flask(__name__)

@app.route('/')
def index():
    response = make_response('<p>Respuesta con ckookies</p>')
    response.set_cookie('articulo', 'Samsung S3')
    return(response)
if __name__ == '__main__':
    app.debug = True
    app.run(host='0.0.0.0')
```

REDIRECCIONES

- Podemos programar redirecciones: cuando el cliente solicite una URL, se le redirige a otra.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from flask import redirect

app = Flask(__name__)

@app.route('/')
def index():
    return redirect(http://www.uah.es)
if __name__ == '__main__':
    app.debug = True
    app.run(host='0.0.0.0')
```

- Se suele utilizar para redirigir al cliente a la página de login si aún no se ha logeado.

-

TEMPLATES

- Jinja2 es un motor de plantillas (templates) de código html
- Para usarlo llamamos el método: **render_template()**
- Se le proporciona el nombre de la plantilla y las variables que debe usar
- Los templates se guardan en la carpeta **/templates**

TEMPLATES

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

```
from flask import Flask, render_template
```

```
app = Flask(__name__)
```

```
@app.route('/desc/<sensor>')
def show_type(sensor=None):
```

si no se especifica sensor en el URI, pone sensor=None

```
    return render_template('sensors.html', sens=sensor)
```

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

Dinamismo

```
<!doctype html>
<title>Sensors in iroom</title>
{% if sensor %}
    <div>Tipo de sensor: {{ sens }}!</div>
{% else %}
    <div>Sensor no especificado</div>
{% endif %}
```

TEMPLATES: USO DE FOR

Listamos los valores de la lista `sensors` que contiene nombres de sensores

`sensors = ['temperatura', 'humedad', 'sonido', 'luz']`

```
<!doctype html>
<ul>
{% for sensor in sensors %}
  <li>Sensor: {{ sensor }}!</li>
{% endfor %}
</ul>
```

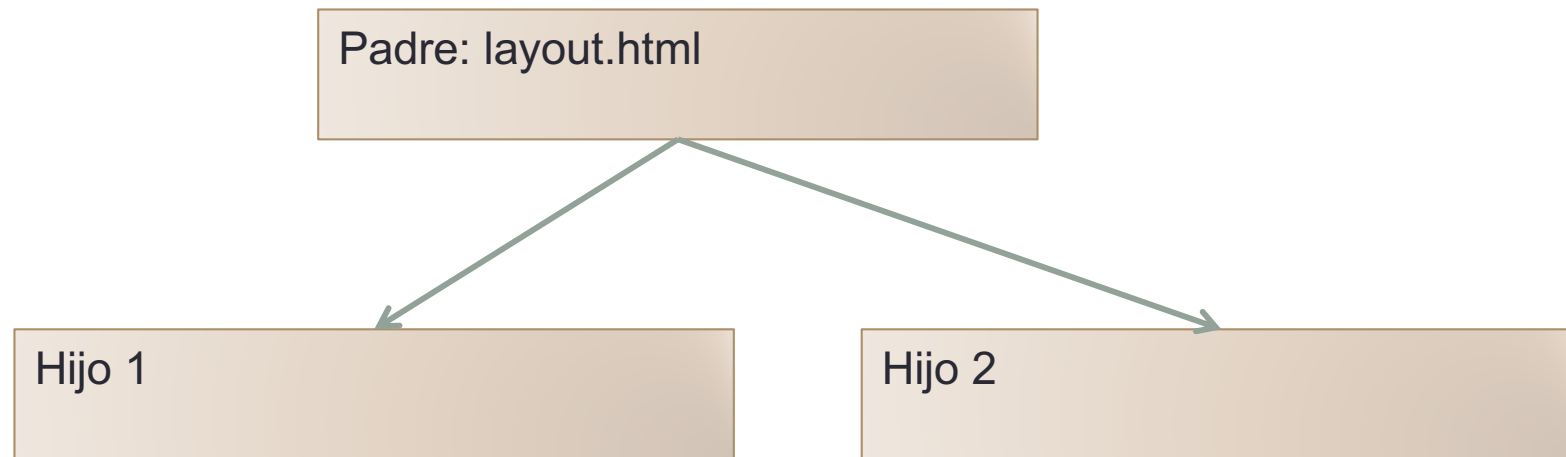
Parámetros en templates y filtros

- Los parámetros que se pasan a los templates pueden ser de varios tipos:
 - Variable: `{{temperatura.value}}`
 - Selector de tipo lista o diccionario: `{{sensor[2]}}`, `{{sensor['temperatura']}}`
 - El resultado de un método: `{{temperatura.get()}}`
- Puede llevar **filtros**: `{{nombre | capitalize}}`
 - `safe`: Visualiza el valor sin aplicar escaping
 - `capitalize`: pasa a mayúsculas el primer caracter, el resto en minúsculas
 - `lower`: minúsculas
 - `upper`: mayúsculas
 - `title`: mayúscula el inicio de cada palabra
 - `trim`: elimina los espacios en blanco extra.
 - `striptags`: elimina etiquetas HTML



HERENCIAS CON TEMPLATES

- Construye un esqueleto:
 - Template padre:
 - Información común
 - Bloques para rellenar
 - Templates hijos:
 - Heredan la información común del padre
 - Se personalizan rellenando los bloques




```

<!doctype html>
<html>
  <head>
    {% block head %}
    <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
    <title>{% block title %}{% endblock %} - Iroom Web</title>
    {% endblock %}
  </head>
  <body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
      {% block footer %}
      &copy; Copyright 2014 by
      <a href="http://www.uah.es/">iroom team</a>.
      {% endblock %}
    </div>
  </body>
</html>

```



Padre: [layout.html](#)

```

{% extends "layout.html" %}
{% block title %}Index{% endblock %}
{% block head %}
  {{ super() }}
  <style type="text/css">
    .important { color: #336699; }
  </style>
{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Página de sensores.
  {% endblock %}

```

[super\(\)](#): extiende el contenido
del bloque definido en el padre



Hijo

BASES DE DATOS MYSQL

- Podemos usar dos módulos **Flask-mysql** y **FlaskAlchemy**
- Veremos Flask.mysql
 - Instalación: `pip install flask-mysql` (<http://flask-mysql.readthedocs.org/en/latest/>)
 - Importamos: `from flaskext.mysql import MySQL`
- Variables de configuración de acceso:
 - `MYSQL_DATABASE_HOST`
 - `MYSQL_DATABASE_PORT`
 - `MYSQL_DATABASE_USER`
 - `MYSQL_DATABASE_PASSWORD`
 - `MYSQL_DATABASE_DB`
 - `MYSQL_DATABASE_CHARSET`

INICIALIZACIÓN

```
from flaskext.mysql import MySQL
.....
app = Flask(__name__)
mysql.init_app(app)
.....
```

USO

```
conn = mysql.connect()
cursor = conn.cursor()
cursor.execute ("select valor from sensors where nombre='temperature' order by time desc")
```

CONFIGURACION DE FLASK EN FICHERO

- Editamos en la carpeta home: nano .bashrc
- Introducimos la línea:

```
export IROOM_SETTINGS=/home/administrador/iroom/config/iroom.cfg
```

```
MYSQL_DATABASE_HOST = 'localhost'  
MYSQL_DATABASE_PORT = 3306  
MYSQL_DATABASE_USER = 'adroom'  
MYSQL_DATABASE_PASSWORD = 'admin'  
MYSQL_DATABASE_DB = 'iroom'  
DEBUG = True
```

iroom.cfg

iroom.py

```
app = Flask(__name__)  
app.config.from_object(__name__)  
app.config.from_envvar('IROOM_SETTINGS', silent=True)
```

- <http://flask.pocoo.org/docs/0.10/config/>

SSE: SERVER

- El servidor se envía datos al cliente cuando observa actualizaciones.
- Más eficiente que la alternativa de que el cliente pregunte.
- Importamos Response
- **Response función de SSE, el sse_request() se ejecuta continuamente y devuelve la temp cuando cambia**

```
@app.route('/update_sensor')
def sse_request():
    return Response(event_sensor(), mimetype='text/event-stream')
```

En el servidor: iroom.py

```
def event_sensor():
    while True:
        conn = mysql.connect()
        cursor = conn.cursor()
        cursor.execute("select valor from sensors where nombre='temperature' order by time desc")
        temperatura = int(cursor.fetchone()[0])
        if temperatura != last_value[0]:
            sensor = {"tipo": "temperatura", "valor": temperatura}
            data_json = json.dumps(sensor)
            print sensor
            yield 'data: %s\n\n' % str(data_json)
            last_value[0] = temperatura
```

SSE: CLIENT

- El servidor envía datos al cliente cuando observa actualizaciones.
- Más eficiente que la alternativa de que el cliente pregunte.
- <http://flask.pocoo.org/snippets/116/>

```
<script type="text/javascript">
  var sse = new EventSource("/update_sensor");
  sse.onmessage = function(event) {
    var sensor = event.data;
    obj = JSON.parse(sensor)
    if (obj.tipo == 'temperatura') {
      document.getElementById('ct').innerHTML = "Temperatura: "+obj.valor;
      document.getElementById('temperatura').setAttribute('value', obj.valor);
    };
  };
};
```

En el cliente: sensors.html

```
.....

<div id="ct"></div> <meter id="temperatura" value="0" min="0" max="50" low="18" high="26"></meter>
```

SSE BLOQUEANTES

- <http://es.slideshare.net/cppgohan/flask-with-serversent-event>
- <http://flask.pocoo.org/docs/0.10/deploying/wsgi-standalone/>
- Solución: usar gunicorn+gevent
- Instalar desde dentro del entorno de desarrollo (flask):
 - pip install gevent
 - pip install gunicorn
- Para arrancar la aplicación iroom.py usar:
gunicorn -k gevent -w 4 -b '0.0.0.0:5000' iroom:app

AJAX: CLIENT

- Permite enviar datos desde el cliente al servidor cuando ocurre un evento.
- <http://flask.pocoo.org/docs/0.10/patterns/jquery/>

```
<script type=text/javascript>
$(function() {
  $(".button").click(function() {
    $.ajax({
      type: "GET",
      url: $SCRIPT_ROOT + "/setcolor",
      contentType: "application/json; charset=utf-8",
      data: { color: $('input[name="colorLight"]').val() },
      success: function(data) {
        //$('#result').text(data.color);
      }
    });
  });
});
</script>
<div id="result"></div>
<input type="color" id="colorLight" name="colorLight" value="#0000ff">
<button class="button" type="button">Envía color</button>
```

En el cliente: iluminacion.html

AJAX: SERVER

- Permite enviar datos desde el cliente al servidor cuando ocurre un evento.
- <http://flask.pocoo.org/docs/0.10/patterns/jquery/>

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

```
@app.route('/setcolor', methods=['GET'])
def setcolor():
```

```
    color = request.args.get('color')
```

```
    red = int('0x'+color[1:3], 16)
```

```
    blue=...
```

```
    green=...
```

```
    conn = mysql.connect()
```

```
    cursor = conn.cursor()
```

```
    Escribe en la base de datos
```

```
    cursor.execute ("INSERT INTO sensors (nombre, valor)" "VALUES(%s, %s)", ('red', red))
```

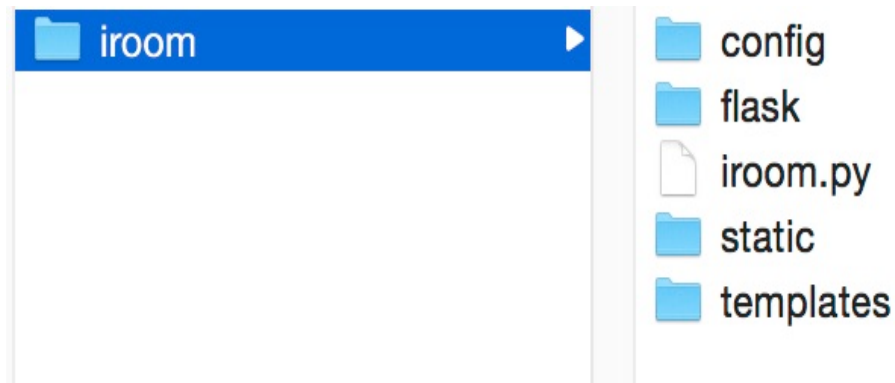
```
    .....
```

```
    conn.commit()
```

En el servidor: iroom.py

Estructura de la aplicación

- En el directorio del proyecto creamos dos subdirectorios:
 - static: css, js, jpeg....
 - templates: plantillas jinja2



Bibliografía

- <http://flask.pocoo.org/docs/0.10/>
- Flask Web Development. Miguel Grinberg. O´Reilly.
- <https://www.youtube.com/user/hermanmu/videos>
- <http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>