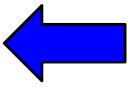
Sistemas Informáticos

Tema 2: Sistemas distribuidos basados en WWW

2.3 Web interactiva - Python

Python para la Web

- El servidor web
  - Recibe la comunicación HTTP(S)
  - Extrae la URL (referencia a recursos o servicio)
  - Extrae parámetros
  - Invoca el código correspondiente…



- Python es un lenguaje generalista no exclusivo para el desarrollo web
  - Las aplicaciones siempre se podrían programar desde cero, pero lo habitual es utilizar bibliotecas o frameworks

Vínculo URL - Código

- En PHP
  - http://www.misitioweb.com/index.php

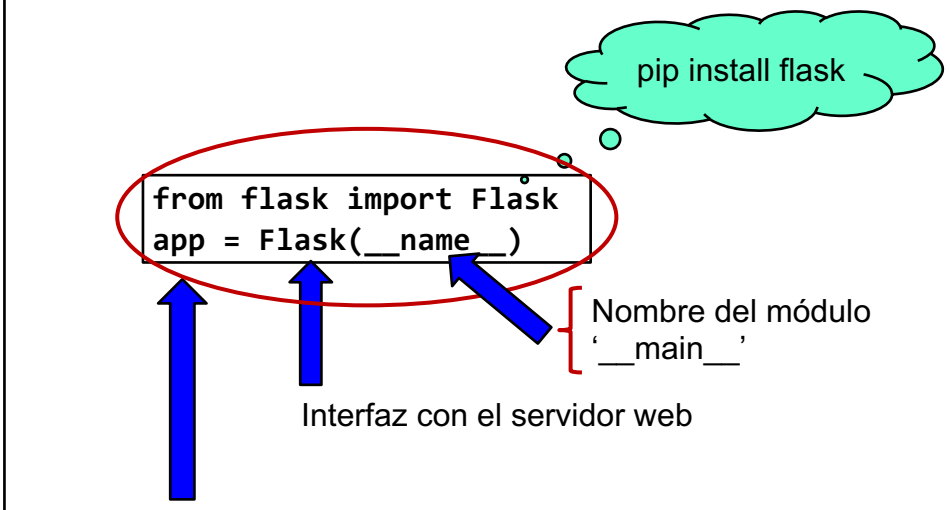
→ – Busca un fichero llamado 'index.php' y lo interpreta

- En Python depende de la biblioteca que usemos
  - Aquí vamos a ver Flask y Django

Flask

- Micro-framework (en contraste con frameworks más completos como Django)
  - Núcleo simple pero extensible
  - No toma decisiones por el programador (como por ejemplo, tipo de base de datos, e incluso si usa BdD o no!)
  - Provee librerías (extensiones) para, por ejemplo:
    - Integración de BdD
    - Validación de formularios
    - Gestión de ficheros (uploads)
    - Autenticación
  - … pero puedo ignorar todo esto e implementarlo por mi mismo!

Flask: uso



Flask: lanzar servidor HTTP

- Flask viene con un servidor HTTP propio, **sólo apto** para desarrollo
- Dos métodos para lanzar el servidor
  - Antes de versión 1 de Flask

1

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

En el propio fichero 'hello.py'

```
$ python hello.py
```

En una terminal

Flask: lanzar servidor HTTP

- Flask viene con un servidor HTTP propio, **sólo apto** para desarrollo
- Dos métodos para lanzar el servidor
  - Antes de versión 1 de Flask

①

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

Alternativa

En el propio fichero 'hello.py'

```
$ python hello.py
```

En una terminal

Flask: lanzar servidor HTTP

- Flask viene con un servidor HTTP propio, **sólo apto** para desarrollo
- Dos métodos para lanzar el servidor
  - A partir de la versión 1 de Flask

②

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

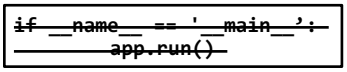
Nada en el propio fichero 'hello.py'

```
$ export FLASK_APP=hello.py
$ flask run
 * Running on http://127.0.0.1:5000/
```

En una terminal

Flask: lanzar servidor HTTP

- Flask viene con un servidor HTTP propio, **sólo apto** para desarrollo
- Dos métodos para lanzar el servidor
  - A partir de la versión 1 de Flask



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

Nada en el propio fichero 'hello.py'

Alternativa

```
$ export FLASK_APP=hello.py
$ python -m flask run
     * Running on http://127.0.0.1:5000/
```

En una terminal

Algunas opciones

- Para que sea visible desde el exterior

```
$ flask run --host=0.0.0.0
```

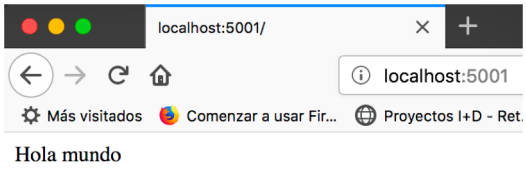
- Modo Debug

```
$ export FLASK_ENV=development
$ flask run
```

Encaminamiento de URLs (routing)

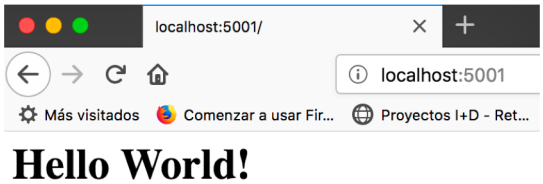
- Establece las relaciones entre URLs y código Python (funciones)

```
@app.route('/')
def index():
    return 'Hola mundo'
```



Sería mejor devolver HTML

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

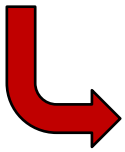


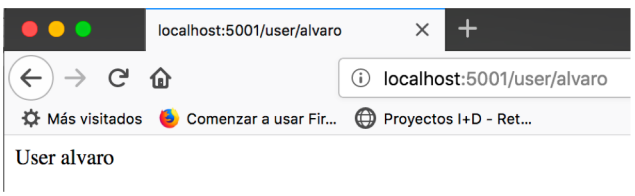
Vistas

- Las funciones como index() se convierten en manipuladores (*handlers*) de la URL asociada (la raíz, en este caso).
- En Flask (y otros frameworks) las funciones creadas para ser *handlers* se denominan vistas (*views*)
- Si basamos nuestra aplicación en una arquitectura MVC, los *handlers* suelen hacer las veces de controlador (cuidado con la posible confusión del nombre *view*)

Caminos variables

```
@app.route('/user/<username>’)
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username
```





Convertidores

Converter types:

`string`	(default) accepts any text without a slash
`int`	accepts positive integers
`float`	accepts positive floating point values
`path`	like `string` but also accepts slashes
`uuid`	accepts UUID strings

```
@app.route('/post/<int:post_id>’)
def show_post(post_id):
 # show the post with the given id, the id is an integer
 return 'Post %d' % post_id

@app.route('/path/<path:subpath>’)
def show_subpath(subpath):
 # show the subpath after /path/
 return 'Subpath %s' % subpath
```

Método HTTP

- Los caminos pueden ser diferentes dependiendo del método utilizado para la petición HTTP

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

Se puede complicar rápidamente

```
@app.route('/ex2')
def function_for_ex2():
    """
    It process the '/' url.
    :return: basic HTML for /ex2
    """
    return '<!DOCTYPE html> ' \
           '<html lang="es">' \
           '<head>' \
           '<title> This is the page title </title>' \
           '</head>' \
           '<body> <div id ="container">' \
           '<h1>Example of HTML Content</h1>' \
           'Return to the homepage by clicking <a href="/"> here</a> </body>' \
           '</html>'
```

Solución: separación de responsabilidades

```
@app.route('/ex2')
def function_for_ex2():
    """
    It process the '/' url.
    :return: basic HTML for /ex2

    """

    return app.send_static_file("ejemplo.html")
```

Todo parece llevar
a una arquitectura MVC

static/ejemplo.html

```
<!DOCTYPE html>
  <html lang="es">
  <head>
    <title> This is the page title </title>
  </head>'
  <body> <div id ="container">
    <h1>Example of HTML Content</h1>
    Return to the homepage by clicking <a href="/"> here</a>
  </body>
</html>
```

¿Pero entonces HTML estático o "la muerte"?

- Entran los *templates*
- Por *default*, Flask utiliza el motor (intérprete) de templates Jinja2
  - http://jinja.pocoo.org/

```
from flask import render_template

@app.route('/hello/') @app.route('/hello/<name>')

def hello(name=None):
 return render_template('hello.html', name=name)
```

```
/application.py
/templates
    /hello.html
```

Templates

```
<!doctype html>
 <title>Hello from Flask</title>
 {% if name %}
   <h1>Hello {{ name }}!</h1>
 {% else %}
   <h1>Hello, World!</h1>
 {% endif %}
```

```
/application.py
/templates
    /hello.html
```

Cookies

- Guardar cookies:

```
@app.route("/setcookie/<user>")
def setcookie(user):
    msg = "user cookie set to: " + user
    response = make_response(render_template('mensaje.html', mensaje=msg))
    response.set_cookie('helloflask_user',user)
    return response
```

```
From flask import make_response
```

- Leer cookies:

```
@app.route("/getcookie")
def getcookie():
    user_id = request.cookies.get('helloflask_user')
    if user_id:
        msg = "user is: " + user_id
    else:
        msg = "no user cookie"
    return render_template('mensaje.html', mensaje=msg)
```

Sesiones

- Aunque Flask no obliga a usar un sistema de sesiones en particular, viene con uno predefinido.
- Construido sobre cookies firmadas criptográficamente
  - Ayuda a evitar Session hijacking y otros ataques informáticos

```
from flask import session

@app.route("/setsession/<data>")
def setsession(data):
    msg = "session data set to: " + data
    session['data'] = data
    session.modified = True
    return render_template('mensaje.html', mensaje=msg)

@app.route("/getsession")
def getsession():
    if 'data' in session:
        msg = "session data: " + session['data']
    else:
        msg = "no session data"
    return render_template('mensaje.html', mensaje=msg)
```

Django

- Framework de alto nivel para el desarrollo de aplicaciones web basadas en Python
- De mayor calado que Flask, sin llegar a ser un macro-framework
- Gestiona e integra prácticamente todos los aspectos relevantes del desarrollo de una aplicación web desde una perspectiva de **proyecto cerrado**
- La poca flexibilidad en el diseño de la arquitectura software se consigue vía customización
- Paquetes necesarios para usar Django
  - Framework (django)
  - Herramientas para la gestión de proyectos Django (django-admin)

Django: lanzar servidor HTTP

- Como todo, a través del script de administración

```
$ python manage.py runserver
```

- Dentro de la estructura de archivos que define el servidor con sus aplicaciones
- Lee los parámetros de configuración del fichero *settings.py*

Sesiones

- Gestionadas por un middleware que se debe activar en el fichero de configuración
  - Permite abstraer al desarrollador de todo el problema del envío, recepción y gestión de datos
- Por defecto, se basan en el almacenamiento de datos en el servidor y el envío de una cookie con un id de sesión
  - También es posible un mecanismo basado en el almacenamiento de datos en cookies (menos recomendable)
- Los datos de sesión por defecto se almacenan en la base de datos en unas tablas que se crean vía una migración
  - También es posible configurarlo para que se almacenen en disco o en memoria

Sesiones

- Cada vez que se recibe una nueva petición HTTP en el servidor, el middleware obtiene los datos de sesión del sistema de almacenamiento y los guarda en la Request en un objeto similar a un diccionario

```
from django.http import HttpResquest

request.session[nombre_variable]          // Lectura
request.session[nombre_variable] = valor // Asignación
```

- Cuando se termina el procesamiento de una petición, el middleware persiste los datos almacenados en la Request.
  - Si han cambiado
  - Se puede forzar la actualización:

```
request.session.modified = True
```

- Por defecto, los datos se serializan en JSON