

Curso Python en 8 clases

Clase 8: Python en la web

Autor: Sebastián Bassi 

Versión: 1.5

Licencia: Creative Commons BY-NC-SA 2.5. ([ver texto completo](#))



Introducción

Hay distintos modos de ejecutar código Python en el servidor. Cada modo tiene sus ventajas y desventajas. El método mas fácil es via CGI, que ejecuta una instancia de Python cada vez que se lo llama. Es a su vez el método mas lento. El código Python puede correrse "embebido" en el HTML al estilo PHP, aunque no se lo recomienda debido a que el código resultante es difícil de mantener. Para evitar tener que levantar un interprete cada vez que se ejecuta una página hecha dinámicamente con Python, existen alternativas como `mod_python` y `WSGI`.

En esta clase veremos los fundamentos de cada método hasta llegar a usar un web framework completo como *web2py*.

CGI

Configurando el servidor web para CGI

Basicamente hay que indicar el archivo donde se van a ejecutar los scripts, la extensión de los mismos y un alias para no mostrar el path a los scripts.

Si el directorio de scripts se encuentra en `/var/www/apache2-default/cgi-bin/`, tenemos que incluir las siguientes líneas en el archivo de configuración de Apache:

```
<Directory /var/www/apache2-default/cgi-bin>
    Options +ExecCGI
</Directory>
```

La extensión se indica asi:

```
AddHandler cgi-script .py
```

El alias se indica con la directiva *ScriptAlias*:

```
ScriptAlias /cgi-bin/ /var/www/apache2-default/cgi-bin/
```

¡No olvidar!

Hay que especificar que el script tenga permiso de ejecución:

```
chmod a+x MyScript.py
```

Ejemplo de código en CGI:

```
#!/usr/bin/env python

import time
print("Content-Type: text/html\n")
print("<html><head><title>Test page</title></head><body>")
print("<h1>HELLO WORLD!</h1>")
print("Local time: %s"%time.asctime(time.localtime()))
print("</body></html>")
```

Este código genera el siguiente HTML:

```
<html><head><title>Test page</title></head><body>
<h1>HELLO WORLD!</h1>
Local time: Mon Apr 12 15:11:06 2010
</body></html>
```

(ver pastecode.com.ar/cgi-bin/test1.cgi.)

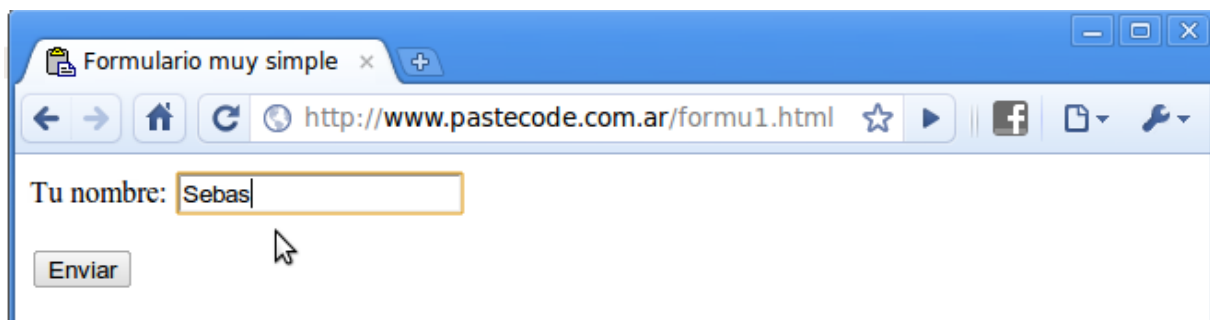
Si bien el resultado no aporta mucho porque no tiene interactividad, sirve para comprobar que el servidor está preparado para correr aplicaciones Python.

Página generada dinámicamente con datos provistos por el usuario

Una manera típica de ingresar datos es vía un *form* de HTML:

```
<html><head><title>Formulario muy simple</title></head>
<body>
<form action='saludos.cgi' method='post'>
Tu nombre: <input type='text' name='username'> <p>
<input type='submit' value='Enviar'>
</form></body></html>
```

Este form es mostrado de esta manera:

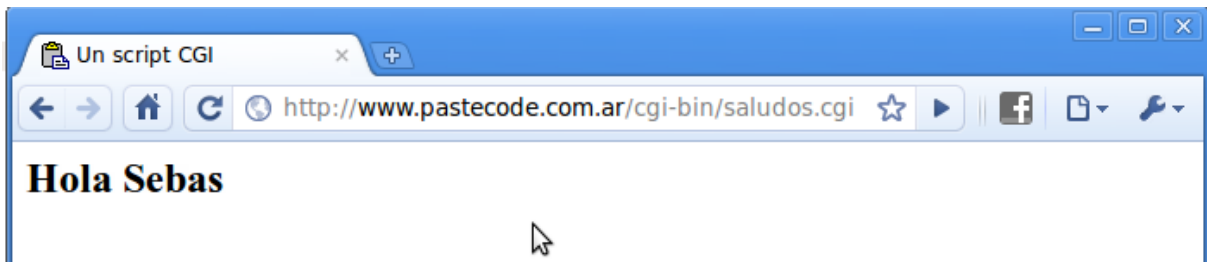


El siguiente código procesa los datos ingresados:

```
#!/usr/bin/env python

import cgi
print("Content-Type: text/html\n")
form = cgi.FieldStorage()
name = form.getvalue("username", "NN")
print("<html><head><title>Un script CGI</title></head>")
print("<body><h2>Hola %s</h2></body></html>"%name)
```

Por lo que obtenemos una respuesta como esta:



Nota: Para ver los errores via HTML, usar `cgitb.enable()` (usarlo solo en diseño, no en producción).

Mod_Python

El problema principal con CGI es que cada vez que llamamos a una página de este tipo, el interprete de Python se ejecuta. Esto no escala bien debido a los requerimientos de RAM y CPU que convella levantar un interprete por cada request. Otro problema asociado con CGI es la velocidad de ejecución (que es baja por la razón citada).

Una manera de evitar estos inconvenientes es tener un interprete corriendo permanentemente y ejecutar en él todas las peticiones. Esto es provisto por el módulo de Apache *mod_python*.

Con Mod_Python se puede ejecutar 1200 request/s contra 23 request/s como en CGI.

Configurando el servidor web para Mod_Python

Instalación en Linux:

```
apt-get install libapache-mod-python2.5
```

Luego hay que modificar el archivo de configuración de Apache de la siguiente manera:

```
<Directory /var/www/apache2-default>
    AddHandler mod_python .py
    PythonHandler mptest
    PythonDebug On
</Directory>
```

Note que *mptest* es el nombre de la aplicación que se encargará de la petición. Se puede cambiar. Entonces la aplicación se llama *mptest.py* y se encuentra en el directorio */var/www/apache2-default*.

Ejemplo de uso:

```
#!/usr/bin/env python
from mod_python import apache
from mod_python.util import FieldStorage

def chargeandprop(AAseq):
    # ..RESUMIDO..
    return (c,p)

def handler(req):
    req.content_type='text/html'
    fs = dict(FieldStorage(req))
    charge,pvalue = chargeandprop(fs['seq'])
    html = '<html><body>Titulo:%s<br/>\'
           \'Tu secuencia:<br/>%s<br/>\'\'
```

```

        'Carga neta:%s<br/>' \
        %(fs['title'], fs['seq'], charge)
req.write(html)
if fs['prop']=="y":
    html = "Proporcion: %.2f" %(pvalue)
    req.write(html)
req.write("<br/></body></html>")
return apache.OK

```

Notar las diferencias con CGI.

Para mas información sobre Mod_Python: modpython.org

Mod_Python con Publisher

Se muestra *Publisher* porque muestra practicas actuales en desarrollo web.

Para usar *Publisher* hay que modificar el archivo de configuración de Apache:

```

<Directory /var/www/apache2-default>
    SetHandler mod_python
    PythonHandler mod_python.publisher
    PythonDebug On
</Directory>

```

En este archivo se establece que todos los archivos de ese directorio serán tratado con *Mod_Python*. Ahora no importa el nombre del archivo (todos los archivos serán tratados con *Publisher*) pero hay que especificar la función en el URL:

```

<form action='/apache2-default/handler.py/netc' method='POST'>

```

Veamos el mismo programa con *Publisher*:

```

def chargeandprop(AAseq):
    # ..RESUMIDO..
    return (c,p)

def netc(req,seq,title,prop):
    req.content_type = 'text/html'
    charge,propval = chargeandprop(seq)
    html = '<html><body>Titulo:%s<br/>' \
           'Tu secuencia:<br/>%s<br/>' \
           'Carga neta:%s<br/>' \
           %(title,seq,charge)
    req.write(html)
    if prop=="y":
        html = "Proportion of charged AA: %.2f" %propval
        req.write(html)
    req.write("<br/></body></html>")
    return None

```

Las diferencias mas importantes pasan por las variables, el modo de llamar a la función y que tiene características de mas alto nivel.

WSGI

Hay varios frameworks de aplicaciones web que limitan las opciones de web servers (los pares aplicaciones/web server suelen ser fijos). En JAVA hay un "servlet API" que hace que cualquier aplicación que funcione sobre esa API puede andar en cualquier servidor que soporta dicha API. Por esta razón se crea WSGI (ver [PEP 333](#)), para tener una interface común para desarrollar aplicaciones web.

Preparando el servidor para WSGI

Se puede bajar el módulo de code.google.com/p/modwsgi o desde los repositorios como *libapache2-mod-wsgi*.

Una vez instalado, hay que agregar estas líneas en el archivo de configuración de Apache:

```
WSGIScriptAlias webpath path_en_servidor
WSGIScriptAlias / /var/www/sitepath/htdocs/test.wsgi
```

La primera línea indica que *path_en_servidor* será el path que recibirá todas las peticiones mientras que en la segunda se especifica que los pedidos a / serán manejados por el script indicado (*test.wsgi*).

Webserver WSGI incorporado

Desde Python 2.5 hay un servidor WSGI incluido. Con 3 líneas de código podemos tener un servidor web funcionado:

```
from wsgiref.simple_server import make_server
server = make_server('localhost', 8888, application)
server.serve_forever()
```

Este servidor es útil para diseño y pruebas, pero no fue diseñado para ser usado en producción por limitaciones de escala y seguridad.

El uso de WSGI "puro" implica usar código de bajo nivel. Veamos un "Hola mundo!":

```
def application(environ, start_response):
    status = '200 OK'
    output = "<html><head><title>WSGI</title></head>\"
            "<body>Hola mundo!</body></html>"
    response_headers = [ ('Content-type', 'text/html'),
                          ('Content-Length', str(len(output))) ]
    start_response(status, response_headers)
    return output
```

application es una función que el servidor llama en cada petición. Sus parámetros son *environ* (un diccionario con las variables) y *start_response* (función que devuelve los encabezados HTTP).

Esto puede evitarse usando *middleware* (componentes intermedios entre WSGI y el programa del usuario). Como WSGI es parte de Python, hay muchos *middleware* disponibles. (ver: wsgi.org/wsgi/Middleware_and_Uilities)

Yaro como ejemplo de middleware

Yaro es un *middleware* para WSGI, se baja desde lukearno.com/projects o usando *easy_install*.

Primero un "Hola mundo" en Yaro:

```
from yaro import Yaro

def _application(req):
    output = "<html><head><title>HW WSGI</title></head>"\
            "<body>Hola mundo!</body></html>"
    return output

application = Yaro(_application)
```

Comparado con el código anterior se nota que es de mas alto nivel.

Ahora veamos una aplicación que usa mas opciones de *Yaro*, incluyendo *selector*, otro middleware del mismo autor.

```
#!/usr/bin/env python

from selector import Selector
from yaro import Yaro

rootdir = '/var/www/mywebsite/htdocs/'

def index(req):
    return open(rootdir+'form.html').read()

def _chargeandprop(AAseq):
    # ..RESUMIDO..
    return (c,p)

def netc(req):
    seq = req.form.get('seq','')
    title = req.form.get('title','')
    prop = req.form.get('prop','')
    charge,propval = _chargeandprop(seq)
    html = '<html><body>Titulo:%s<br/>'\
            'Tu secuencia:<br/>%s<br/>'\
            'Carga neta:%s<br/>'\
            %(title,seq,charge)
    yield html
    if prop=="y":
        yield "Proporcion: %.2f"%propval
    yield "<br/></body></html>"

s = Selector(wrap=Yaro)
s.add('/', GET=index)
s.add('/netc', POST=netc)
application = s
```

Ver código completo en <http://py3.us/57>

Seguridad: La web es el "salvaje Oeste"

Siempre hay que tener en cuenta que los usuarios pueden ingresar datos de manera inesperada, pero esto es mas importante en las aplicaciones online. No solo los usuarios ingresarán datos,

sino que una vez que la aplicación está disponible online, está expuesta a scripts que buscan explotar debilidades de sitios web. Por lo tanto hay que tener en cuenta algunos temas extras:

- XSS (aprox. 80% vulnerabilidades)
- Inyección de SQL
- Validación de entradas

Web frameworks

Hay infinidad de web frameworks para Python (ver: wiki.python.org/moin/WebFrameworks). Todos proveen servicios que son típicamente usados en entornos web como acceso a bases de datos, sistema de templates, administración de usuarios, sesiones, manejo de cookies, i18n, redirecciones y otros.

El framework mas usado es Django, aunque la dispersión de uso es muy grande y no hay ninguno que predomine como para ser considerado "standard". Debido al tiempo disponible vamos a presentar uno solo. En este caso veremos *web2py* debido a las prestaciones que ofrece, la facilidad de uso y por sus aspectos didácticos.

Web2py

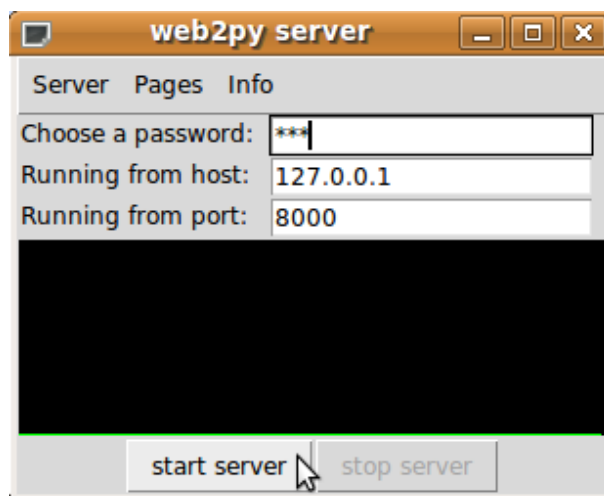
web2py ofrece una serie de facilidades que ayudan en el desarrollo de aplicaciones web. Entre ellos: DAL (Capa de abstracción de datos), seguridad web, manejo de errores via tickets, separación de código según el paradigma MVC (Modelo, Vista y Controlador), formularios generados automáticamente, i18n, desarrollo via IDE basado en web, generación automática de CSV, RSS y XML, cron portable, distribución compilada (optativo) y otros servicios. La licencia es GPL 2.0.

Instalación e inicio

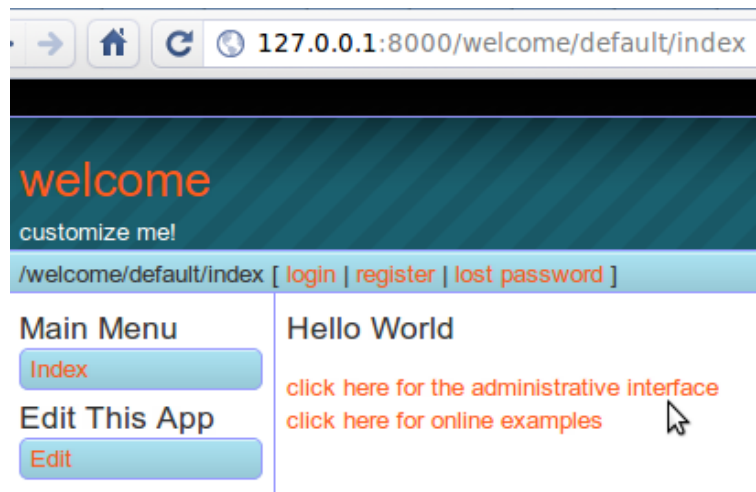
Funciona como una *aplicación portable*. Se baja la versión correspondiente al SO desde <http://www.web2py.com/examples/default/download> y se ejecuta. En el caso de la versión en fuente (para *nix), se ejecuta así:

```
python2.5 web2py.py
```

Al ejecutarse pide los parámetros del servidor en una ventana como esta:



Por defecto carga el programa *welcome*, que nos permite acceder a la interfaz de administración:

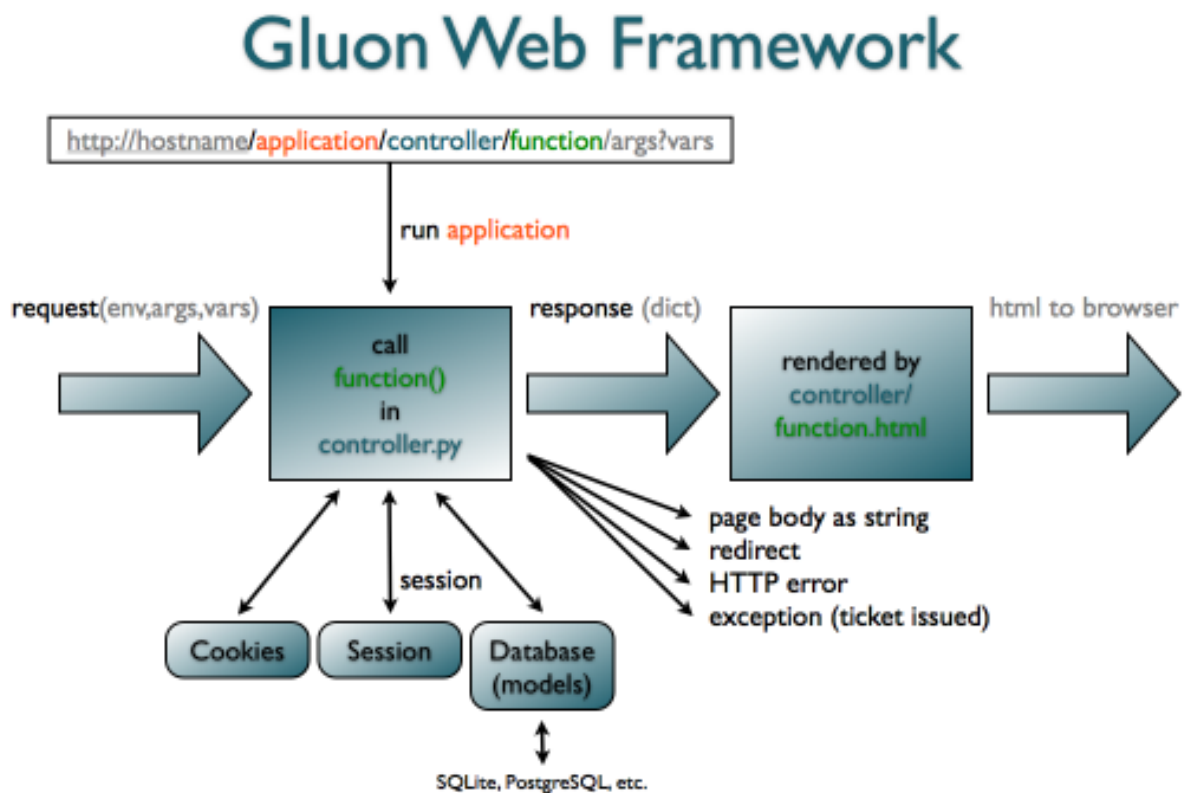


Funcionamiento general

El framework se maneja completamente via web. Separa claramente la aplicación en 3 capas (MVC):

- Modelo: Archivo donde se establece el *modelo* de los datos. Se usa un DAL propio.
- Vista: HTMLs (con código Python optativo) que "decoran" los resultados de los controladores.
- Controladores: Archivos .py con funciones que son llamadas desde el navegador (o desde otras funciones).

El siguiente es un esquema que muestra la relación entre los URLs y los componentes principales de Web2py:



Componentes

A continuación se desarrollan algunos de los servicios provistos por web2py.

DAL

Permite describir la base de datos (y sus operaciones) sin tener que usar sintaxis SQL. Este módulo se encarga de "traducir" los comandos en el código SQL de acuerdo a la base que estemos usando. Las tablas de las bases de datos se mapean como instancias de una clase, y los registros como instancias de otra clase.

Uso de DAL para crear una base, definir una tabla, insertar, contar, borrar, actualizar un registro y un *select*.

```
db = DAL('postgres://user:password@hostname/db')
db.define_table('person', Field('name', 'string'))
id= db.person.insert(name='max')
query=(db.person.id==id)
db(query).count()
db(query).delete()
db(query).update(name='Max')
rows = db(query).select(orderby=db.person.name)
for row in rows: print row.name
```

También es posible agregar "restricciones" usando DAL, ver:

```
1. import datetime; now=datetime.date.today()
2. db=SQLDB('sqlite://db.db')
3.
4. db.define_table('category', SQLField('name'))
5.
6. db.define_table('recipe',
7.                 SQLField('title'),
8.                 SQLField('description', length=256),
9.                 SQLField('category', db.category),
10.                SQLField('date', 'date', default=now),
11.                SQLField('instructions', 'text'))
12.
13. db.category.name.requires=[IS_NOT_EMPTY(), IS_NOT_IN_DB(db, 'category.name')]
14. db.recipe.title.requires=[IS_NOT_EMPTY()]
15. db.recipe.description.requires=IS_NOT_EMPTY()
16. db.recipe.category.requires=IS_IN_DB(db, 'category.id', 'category.name')
17. db.recipe.date.requires=IS_DATE()
```

Existe una aplicación web (hecha en web2py) que permite crear un diseño de DB de manera gráfica y lo convierte al formato DB: <http://gaesql.appspot.com/>

Una vez generada la tabla, *web2py* provee de una interfaz estilo "phpadmin" para administrar los datos.

Mas información sobre DAL: <http://www.web2py.com/examples/default/dal>

Controladores

Funciones que controla el programa, no suelen necesitar parámetros (levantan siempre el objeto *request*) y devuelven un diccionario. Es donde está la lógica del programa. Quedan expuestas como URLs, salvo que tengan `__` adelante.

Vistas

Por defecto todos los diccionarios retornados por los controladores son mostrados en una vista predeterminada. Esto sirve para debugging, no para mostrar en producción. Para eso se hacen las vistas asociadas a cada controlador. Web2py asocia las vistas con los controladores en función del path y el nombre. Así el archivo *default/index.html* sirve para presentar a la función *index* del controlador *default.py*.

HTML

Se usan `{{ y }}` para embeber código Python dentro de los HTML. Para incluir un archivo de código en otro se usa *include* y *extend* (son directivas propias del template). El siguiente html (*layout.html*) tiene un *include*:

```
<html><head><title>Page Title</title></head>
<body>
  {{include}}
</body>
</head>
```

Que es extendido por ejemplo de esta manera:

```
{{extend 'layout.html'}}
<h1>{{=m}}</h1>
```

Este código asume que el controlador asociado a esta vista retorna via un diccionario la clave *m*.

También provee con los elementos típicos de HTML (a, p, etc):

```
{{=P(A(T("click here for the administrative interface"),
_href=URL('admin','default','index')),_style="padding-top:1em;"))}}
```

Lo cual es renderizado a:

```
<p style="padding-top:1em;"><a href="/admin/default/index">
click here for the administrative interface</a></p>
```

Ver mas: <http://web2py.com/book/default/section/5/4>

i18n

El soporte de i18n automáticamente detecta el idioma del navegador para mostrar el string correspondiente. Los string a traducir se marcan `T("string...")`. Luego se genera el archivo para cargar las traducciones. El archivo se genera con la opción *update all languages* en la sección *Languages*. En esa sección se pueden crear y/o editar archivos de traducción. No hace falta saber el formato de los mismos porque las traducciones se agregan via web.

URL

Genera URLs dentro de la aplicación. Modo de uso:

```
>>> URL(a='a', c='c', f='f', args=['x', 'y', 'z'],
...      vars={'p':1, 'q':2}, anchor='1')
'/a/c/f/x/y/z#1?q=2&p=1'
```

a es aplicación, *b* es controlador y *f*, función. Normalmente se usa con el parámetro *r* que contiene estos datos: `URL(r=request, f='index')`

Ver mas: <http://web2py.com/book/default/docstring/URL>

Generación de formularios en base a tabla SQL

Con el método `SQLFORM` podemos autogenerar un formulario en base a una tabla. Incluso traduce a JS las validaciones relacionadas a las restricciones de los campos (especificadas con el DAL). También permite incluir nuestras propias restricciones (con el parámetro `onvalidation=fn_de_validacion`).

Ejemplo de uso:

```
form = SQLFORM(db.receta, campos=[ 'titulo', 'descripcion',
                                   'categoria', 'instrucciones' ])
if form.accepts(request.vars, session):
    redirect(URL(r=request, f="recetas"))
return dict(form=form)
```

Ver mas: <http://web2py.com/book/default/section/7/2> y <http://web2py.com/book/default/section/7/7>

Manejo de errores via tickets

Los errores no son mostrados al usuario. En su lugar hay un enlace con un código que identifica al error. El administrador puede acceder al reporte de error desde ese enlace o desde el menú de errores (*errors* arriba a la derecha).

Soporte de archivos estáticos

Los archivos estáticos se suben en el directorio correspondiente de la aplicación desde la sección *Static files*. Son archivos que no son procesados para ser servidos.

Ejemplo de aplicación en webpy: Redireccionador de URLs

La siguiente aplicación web permite elegir un "URL corto" para mapearlo a un "largo". Así podemos hacer que `midominio.com/p=algo` sea equivalente a un URL de hasta 500 caracteres (el límite puede ampliarse). Se tiene en cuenta que no se permite ingresar 2 veces el mismo URL corto y en ese caso se sugiere uno al azar que no está en la DB.

Modelo

db.py:

```
db = DAL('sqlite://storage.sqlite')
db.define_table('urls',
                SQLField('corto', unique=True, length=32),
                SQLField('largo', requires=IS_NOT_EMPTY()))
```

Controlador

default.py:

```
import string
import random

def index():
    form1 = SQLFORM(db.urls)
```

```

if form1.accepts(request.vars, session, onvalidation=__agregohhttp):
    response.flash = 'form aceptado'
    redirect(URL(r=request, f='proxima',vars=form1.vars))

elif form1.errors:
    propuesta = __clavenotindb()
    response.flash = propuesta
else:
    response.flash = 'por favor complete el formulario'
return dict(form = form1)

def redirecciono():
    # si no hay argumentos:
    if len(request.args) == 0:
        redirect('http://' + request.env.http_host)
    else:
        urlpd = request.args[0][2:]
        # busco urlpd en la DB!
        recs = db(db.urls.corto==urlpd).select(db.urls.largo)
        if len(recs) != 0:
            rec = recs[0].largo
            redirect(rec)
        else:
            # llamar a pagina que no existe!
            rec = 0
    return dict(urlpd = urlpd, rec = rec)

def proxima():
    m = T('The new URL is ready!')
    u_ori = request.vars.largo
    u_new = request.vars.corto
    return dict(m=m, u_ori=u_ori, u_new=u_new)

def __agregohhttp(form):
    if not form.vars.largo.startswith('http://'):
        form.vars.largo = 'http://' + form.vars.largo
    return None

def __clavenotindb():
    # genero clave al azar
    def _azar():
        crs = string.letters + string.digits
        ns = ''
        for i in range(6):
            ns += random.choice(crs)
        return ns

    while True:
        ns = _azar()
        largo_tmps = db(db.urls.corto==ns).select()
        if len(largo_tmps)==0:
            corto_prop = ns
            break
        else:
            pass
    return corto_prop

```

Vistas

default/index.html:

```
{{extend 'layout.html'}}
{{=form.custom.begin}}
{{=T('Desired name:')}} {{=form.custom.widget.corto}} <br />
{{=T('Actual URL:')}} {{=form.custom.widget.largo}} <br />
{{=form.custom.submit}}
{{=form.custom.end}}
```

default/proxima.html:

```
{{extend 'layout.html'}}
<h1>{{=m}}</h1>
{{=T("The original URL was: ")}} {{=u_ori}} <br />
{{=T("The new URL is: ")}}
{{=A('http://' + request.env.http_host + URL(r=request, f='redirecciono/p='+u_new),
href=URL(r=request, f='redirecciono/p='+u_new))}}<br /><br /><p></p>
{{=A(T("Add another URL!"),_href=URL(r=request, f='index'))}}
```

NOTA:

Para acortar el URL uso un mapeador (directo e inverso) provisto por web2py. Es el archivo *routes.py* cuyo contenido relevante es:

```
routes_in = (
    ('/testme', '/examples/default/index'),
    ('/p=(?P<ff>.*)', '/redirect/default/redirecciono/p=\g<ff>'),
)
routes_out = (
    ('/redirect/default/redirecciono/p=(?P<ff>.*)', '/p=\g<ff>'),
)
```

Mas información

- <http://wsgi.org>
- <http://web2py.com/book/>
- <http://www.mengu.net/post/django-vs-web2py>