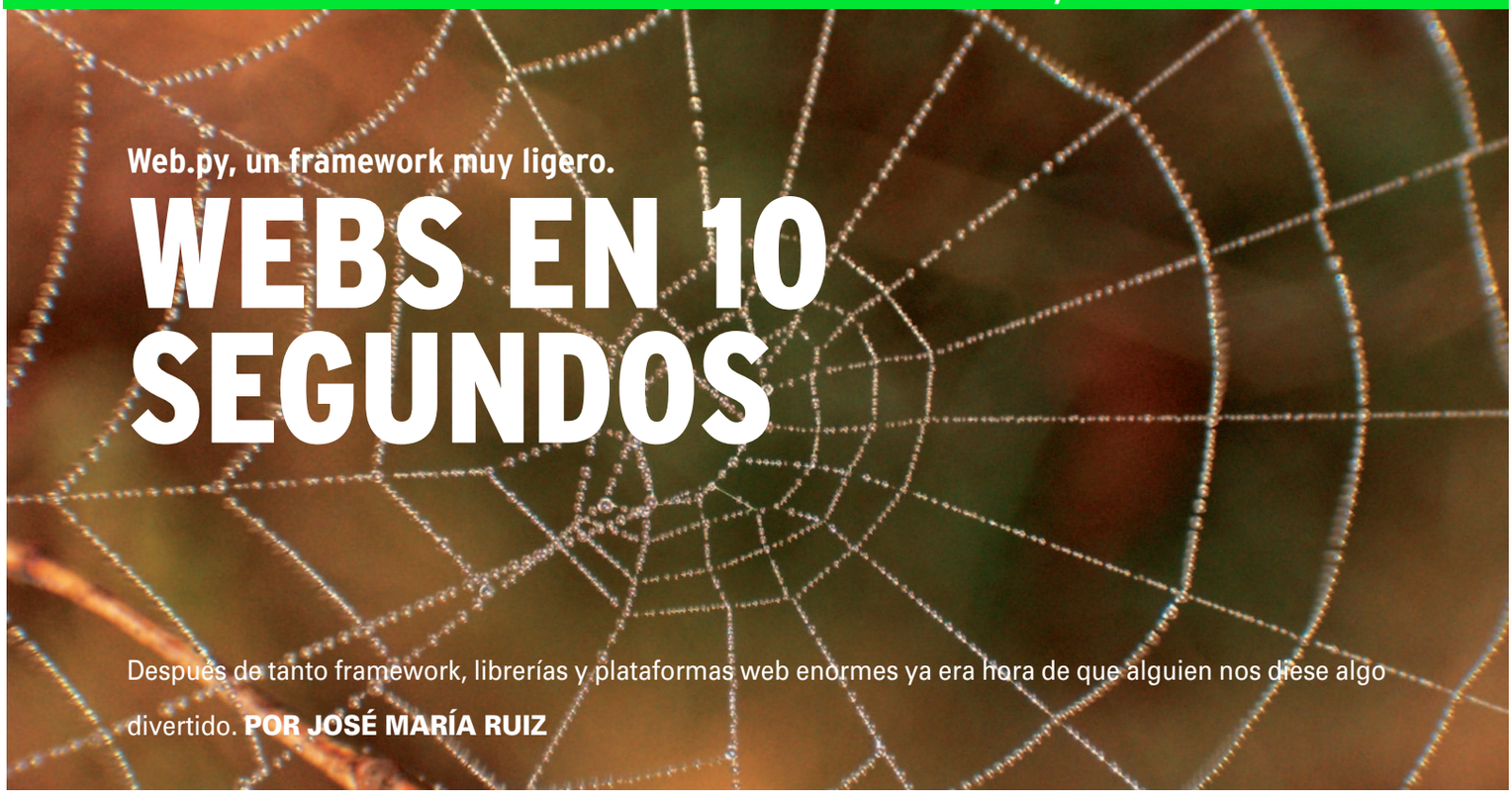

Web.py, un framework muy ligero.

WEBS EN 10 SEGUNDOS

Después de tanto framework, librerías y plataformas web enormes ya era hora de que alguien nos diese algo divertido. **POR JOSÉ MARÍA RUIZ**

Conocí a Aaron Swartz en la Startup School 2005, en Cambridge, EEUU. Es muy joven, no tendrá más de 20 años, y uno de los protegidos de Paul Graham en YCombinator. Hablé con él durante una media hora y me pareció muy inteligente. En ciertos ámbitos es considerado un genio. Aaron ha crecido con la Web, fue uno de los creadores del estándar RSS, para más detalles ver Recursos [1].

Recuperando la web

Web.py es su apuesta para revitalizar la web, para volverla simple, como parte del movimiento Web2.0. Si Ruby on Rails simplificó las monstruosidades de J2EE y compañía, Web.py convierte el desarrollo web en una cosa de niños. En menos de 10 segundos se puede tener funcionando una página web dinámica.

Y es que mucha gente de la llamada Web2.0 quiere volver a los tiempos en los que usar la web era divertido, así que Aaron ha hecho de Web.py una herramienta muy potente y sencilla.

EL AUTOR

José María Ruiz actualmente está realizando el Proyecto Fin de Carrera de Ingeniería Técnica en Informática de Sistemas. Lleva 8 años usando y desarrollando software libre y, desde hace dos, se está especializando en FreeBSD.

Web.py

Cuando se escribe este artículo, Web.py no pasa de la versión 0.13 y tiene un tamaño de poco más de 55KBytes, pero no hay que dejarse engañar por su tamaño. En su programación, Aaron ha hecho uso de las más sofisticadas técnicas de Python, la MetaProgramación entre ellas.

Web.py genera métodos y clases bajo demanda, convirtiendo toda la interacción con bases de datos y la web en un juego de niños. Todo se realiza a través de clases y métodos. Los nombres de ambos han sido escogidos con cuidado para que sean cortos y fáciles de recordar. ¡Adiós a las montañas de documentación!

En su propia página se auto define de «anti-framework».

Ingredientes

Necesitaremos de Web.py, que se puede descargar de la URL que aparece en el Recurso [2]. Además necesitaremos las librerías de Python:

- *Cheetah* (Recurso [3]), que sirve para crear templates.
- *Psycopg* (Recurso [4]), para el acceso a la base de datos Postgresql.

Y por supuesto de Postgresql (Recurso [5]), la base de datos relacional. Haremos uso del sistema de paquetes que nuestra distribución emplee para instalar estas librerías y arrancaremos Postgresql si fuese necesario.

Tanto *Cheetah* como *Psycopg* son opcionales, Web.py ha sido diseñado con una idea en mente: que sea completamente modular. Así podríamos usar otro sistema de templates u otro conector con bases de datos, por ejemplo con MySQL.

Si tenemos todo instalado podemos crear un directorio, donde trabajaremos y copiaremos en el mismo el fichero Web.py.

Primera web... en 10 segundos.

Comencemos con un ejemplo muy simple usando el código que aparece en el Listado 1. Para arrancar el servidor sólo tenemos que ejecutar:

```
# python holamundo.py
```

Desgranemos el código. Lo primero que hacemos es importar *web*, Web.py. De esta manera tendremos acceso a todas sus funciones. Posteriormente definimos una lista que hemos llamado *rutas*. En ella definimos pares de ruta/objeto asociado. Cada vez que accedamos a Web.py éste buscará la ruta que encaje con la que hemos pedido y hará uso del objeto asociado.

Definimos una clase llamada *hola*. En Web.py las clases son quienes «responden» a las peticiones. Para ello debemos definir un método *GET* o *POST*, dependiendo de qué tipo de petición queramos responder. Usando la función estándar *print* podemos crear la respuesta. En nuestro ejemplo, es tan simple, que ni siquiera creamos una página web, solo devolvemos «¡Hola mundo!» (Figura 1).

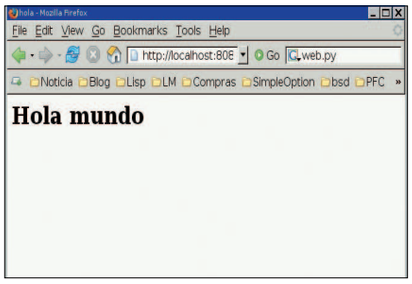

Figura 1: Nuestra primera Web creada con Web.py.

Listado 1 holamundo.py

```
01 #!/usr/local/bin/python
02 import web
03 rutas = (
04   '/', 'hola'
05 )
06
07 class hola:
08   def GET(self):
09    print "<html><head>"
10    print
   "<title>hola<title><head>"
11    print "<body><h1>Hola
   mundo</h1>"
12    print "</body></html>"
13
14 web.internalerror =
   web.debugerror
15
16 if __name__ == '__main__':
17   web.run(rutas, web.reloader)
```

Y por último debemos indicar a Web.py qué rutas usar, para ello empleamos:

```
if __name__ == '__main__':
web.run(rutas, web.reloader)
```

Con esta sencilla llamada nuestro servidor estará listo para ejecutarse. Como ya dijimos antes, el servidor de Web.py se arranca por defecto en el puerto 8080, de manera que no necesitamos permisos de root para hacerlo (son necesarios para abrir cualquier puerto por debajo del 1024). Al acceder veremos «hola mundo». Podemos modificar el puerto en el que se arranca el servidor especificándolo en la linea de comando, así:

```
# python holamundo.py 8003
```

arrancará el servidor en el puerto *8003*. Este es el modo más simple de mostrar información usando HTML: se imprime directamente desde la clase *vista* mediante la función *print*. El problema aparece cuando los ficheros HTML se vuelven más complejos y comenzamos a mezclar presentación (HTML) y control (el fichero Python). Y por ello Web.py dispone de los templates.

Si queremos parar el servidor sólo tenemos que pulsar «Control C» dos veces.

Listado 2: hola.html

```
01 <html>
02
   <head><title>Hola</title></hea
   d>
03 <body><h1>
04 #if $nombre
05   Hola $nombre
06 #else
07   Hola mundo
08 #end if
09 </h1><body></html>
```

Una web más dinámica

Ahora que hemos visto cómo hacer el famoso «hola mundo», pasemos a algo más potente. Y lo haremos con la introducción de los templates. Vamos a crear un directorio llamado *templates* dentro del directorio donde tengamos *holamundo.py*. En su interior crearemos un fichero llamado *hola.html* con el contenido del Listado 2. En lugar de *holamundo.py* usaremos *holamundo2.py*, cuyo código aparece en el Listado 3. El contenido del método *GET* de *hola* ha cambiado a:

```
nombre = 'Isaac Newton'
web.render('hola.html')
```

Cuando ejecutamos *holamundo2.py* aparecerá en el navegador una página donde podremos leer «Hola Isaac Newton». *web.render()* es un método que carga el template que le hemos indicado, *hola.html*, y lo interpreta en el contexto del objeto, y en este contexto *nombre* está asignado a la cadena «Isaac Newton».

Debemos observar con más detenimiento el Listado 2. Este lenguaje tan extraño, con lineas que comienzan con # es el que emplea Cheetah, un sistema de templates para Python. Su objetivo principal es la simplicidad, así que en lugar de emplear las ya famosas etiquetas < % y % > de PHP o Java, usa # para las lineas con código Cheetah y $ para las variables. Web.py lo usa debido a que genera templates que son fáciles de leer y modificar. La ventaja de emplear templates radica en que podremos modificarlas sin tener que volver a arrancar el servidor. Además podremos separar el diseño de la web respecto de la programación.

Programadores y diseñadores son como el agua y el aceite, nunca se mezclan, así que hay que tenerles separados.

```
Filosofía Web.py
```

Si el lector ha cometido algún fallo al escribir parte del template o del fichero *holamundo.py*, cosa que el autor ha hecho y en repetidas ocasiones, se habrá percatado de lo dicharachero que es Web.py en cuanto al informe sobre errores. Es parte de la filosofía de desarrollo de Web.py, ser muy ruidoso cuando algo va mal para que el desarrollador detecte los problemas pronto. El desarrollo rápido de programas requiere al menos de dos factores:

- Que se escriba poco.
- Que se corrijan los errores muy rápido.

Y Web.py ha sido diseñado con ambos en mente, el código escrito es muy compacto y fácil de leer, los nombres de las funciones son cortos y existen numerosos convenios en lugar de excesiva flexibilidad. Algunos de estos principios también rigen el diseño de Ruby on Rails, pero Ruby on Rails tiene un tamaño monstruoso comparado con Web.py.

Hace unas semanas Guido van Rossum, creador de Python, publicó un artículo donde decía que su nuevo trabajo en Google incluía la creación de una página web para la Intranet interna de Google. Estuvo echando un vistazo a los distintos frameworks y le parecían todos demasiado

Listado 3: holamundo2.py

```
01 #!/usr/local/bin/python
02 import web
03
04 rutas = (
05   '/(.*)', 'hola'
06   )
07
08 class hola:
09   def GET(self,nombre):
10     web.render ('hola.html')
11
12 web.internalerror =
   web.debugerror
13
14 if __name__ == '__main__':
15   web.run(rutas, web.reloader)
```

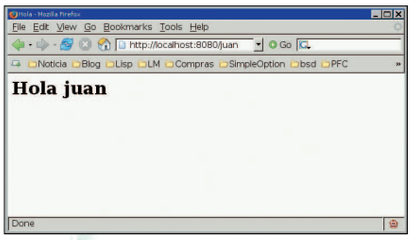

Figura 2: Personalizamos el saludo gracias a expresiones regulares.

pesados y complejos. Guido no es desarrollador web, sino de lenguajes de programación, así que prefiere las cosas sencillas. Web.py ha sido uno de los pocos frameworks que según él recogen el espíritu de Python.

Rutas inteligentes

Ahora vamos a presenciar una de las habilidades de Web.py. Cambiemos de nuevo el fichero *holamundo.py* para que quede como en la Figura 2. Lo paramos y volvemos a arrancar e introduzcamos en el navegador la ruta:

```
http://localhost:8080/juan
```

El resultado será una página donde se saluda a *juan*. ¿Cómo es esto posible? Se ha añadido la expresión *(.*)* a la ruta, una expresión regular. De hecho la ruta en sí misma es una E.R. (expresión regular).

En una E.R. los paréntesis indican la existencia de subexpresiones que pueden ser separadas. En la ruta «/(.*)» tenemos dos expresiones: «/» y «.*». Web.py asocia la segunda a una variable y se la pasa a *GET*, por eso le hemos añadido el parámetro *nombre* en la definición. ¿y si añadimos otra subexpresión? Vamos a verlo en el Listado 4. Necesitaremos también de *hola2.html* que aparece en el Listado 5.

Listado 4: holamundo2b.py

```
01 #!/usr/local/bin/python
02 import web
03
04 rutas = (
05   '/(.*)/(.*)', 'hola'
06   )
07
08 class hola:
09   def
   GET(self,nombre,apellido):
10     web.render ('hola2.html')
11
12 web.internalerror =
   web.debugerror
13
14 if __name__ == '__main__':
15   web.run(rutas, web.reloader)
```

Al añadir otro «/(.*)» en la ruta automáticamente Web.py asocia otra variable a la segunda subexpresión. Podríamos crear cualquier esquema con las rutas, por ejemplo «http://localhost:8080/fecha/id-mensaje». ¿Qué ocurre si dejamos la ruta básica «http://localhost:8080»?

Pues que aparecerá el mensaje «Hola mundo», porque en el template se comprueba la existencia de las variables *nombre* y *apellidos* y al no encontrarlas ejecuta el *else* del *if*.

MVC

Estamos dando vueltas a un esquema de desarrollo bastante popular estos días, hablamos de MVC, el Modelo-Vista-Controlador. Este esquema divide el desarrollo de un sistema en tres componentes:

- La Vista sería el template, que genera código de presentación de algún tipo, como HTML o XML.
- El Controlador sería la clase *hola*, que recoge la información y prepara el entorno para la Vista.
- El Modelo es la información pura y dura, generalmente almacenada en una base de datos, y que se encarga de comprobar la integridad de la misma.

MVC separa el trabajo en tres partes, de manera que sean independientes unas de otras. Una modificación en cualquiera de ellas no implica cambiar el resto. También se aumenta la reutilización, puesto que un mismo controlador puede servir para distintas vistas. A la mente me viene un ejemplo muy recurrido: un controlador que genere un listado de artículos para dos vistas, la primera genera HTML y la segunda RSS.

Por el momento hemos tratado con la Vista y el Controlador ¿dónde está escondido el Modelo en Web.py?

El Modelo

Uno de los requisitos para nuestro ejemplo de uso de Web.py es la librería *Psycopg*. Es un interfaz para la base de datos Postgresql muy pequeño y eficiente, pensado para un uso intensivo de hebras (la eficiencia y la simplicidad serán la tónica general con Web.py). No vamos a interactuar en ningún momento con *Psycopg*, será Web.py quien lo haga.

Listado 5: hola2.html

```
01 <html>
02
   <head><title>Hola</title></hea
   d>
03 <body><h1>
04 #if $nombre and $apellido
05   Hola $nombre $apellido
06 #else
07   Hola mundo
08 #end if
09 </h1><body></html>
```

Comencemos creando una tabla que nos permita almacenar información (Listado 6). En el Listado 7 podemos ver el código de *anotaciones.py*. En él podemos observar cómo se establecen los parámetros de configuración de la base de datos. No conectamos explícitamente con ella, será Web.py quien lo haga:

```
web.db_parameters = dict➥
(dbn='postgres',➥
user='nombre_usuario',➥
pw='clave',➥
db='nombre-base-datos')
```

Como queremos recoger el contenido de la tabla *anotaciones* para poder mostrarlo tenemos que ejecutar un *SELECT*, pero en Web.py es tan sencillo como invocar dentro de *GET*:

```
anotaciones = web.select(➥
"anotaciones")
```

Con esta simple función recogemos en la variable *anotaciones* el resultado de hacer un *select* contra la tabla del mismo nombre. A partir de ese momento *anotaciones* estará disponible en el template *anota.html*.

El código de *anota.html* aparece en el Listado 8. En él usamos una lista html, *UL*, y el lenguaje de templates de Cheetah para iterar sobre *anotaciones*.

```
#for anota in $anotaciones
<li id="t$anota.id">➥
```

Listado 6: anotaciones.sql

```
01 CREATE TABLE anotaciones (
02   id serial primary key,
03   titulo text,
04   created timestamp default
   now()
05 );
```

Listado 7: Anotaciones.py

```
01 #!/usr/local/bin/python
02 import web
03 rutas = (
04   '/', 'index',
05   '/nueva', 'nuevaAnotacion'
06   )
07
08 class index:
09   def GET(self):
10     anotaciones =
   web.select("anotaciones")
11     web.render ('anota.html')
12
13 class nuevaAnotacion:
14   def POST(self):
15     params = web.input()
16     n =
   web.insert('anotaciones',titul
   o = params.titulo)
17
   web.seeother('./#t'+str(n))
18
19 web.internalerror =
   web.debugerror
20
21 if __name__ == '__main__':
22   web.db_parameters =
   dict(dbn='postgres',
   user='josemaria', pw='',
   db='prueba')
23   web.run(rutas, web.reloader)
```

```
$anota.titulo</li>
#end for
```

Entrada de datos

Para volver interactiva nuestra página debemos añadir un formulario, de manera que el usuario pueda añadir nuevas anotaciones:

```
<form method="post"➥
action="/nueva">
  <p><input type="text"➥
name="titulo" />
    <input type="submit"➥
value="Añadir" /></p>
</form>
```

Enviamos el contenido del campo *título* a la ruta */nueva*, pero este último aún no existe, así que creamos en el fichero Python una nueva entrada en *rutas*:

```
rutas = ( '/', 'index',
'/nueva', 'nuevaAnotacion')
```

rutas es una lista y las entradas impares se corresponden con rutas, mientras que las pares lo hacen con clases asociadas. De esta manera relacionamos la ruta */nueva* con la clase *nuevaAnotacion*. Este sistema es muy cómodo porque desvincula el nombre de las rutas respecto del nombre de las clases. La clase *nuevaAnotacion* tendrá la forma:

```
class nuevaAnotacion:
  def POST(self):
    params = web.input()
    n = web.insert(➥
'anotaciones',➥
titulo = params.titulo)
    web.seeother('./#t'+str(n))
```

Analicemos esta clase. Al igual que la clase *index*, aquí definimos un método, pero en esta ocasión es *POST* en lugar de *GET*. Es costumbre enviar los datos de los formularios como POST, así que eso haremos. ¿Pero cómo recogemos los datos enviados desde el navegador?

Aquí entran en juego las funciones de Web.py. Con *web.input()* se recoge la información recibida en una variable. Esta variable será un objeto, y tendrá un atributo por cada parámetro transmitido por POST. Así podremos obtener el titulo de la anotación simplemente accediendo a *params.titulo*.

Para guardar esta información en la base de datos hemos de usar *web.insert()*. Es el equivalente a la orden INSERT de SQL, pero mucho más simple. Le indicamos el nombre de la tabla y un conjunto de valores asignados que corresponden a las columnas de la tabla. De esta manera podemos pasar valores a las columnas por nombre y no importa el orden en que se pongan los parámetros. *web.insert()* devuelve como resultado el *ID*, que en la base de datos se asigne a esa nueva fila. Así podremos hacer referencia a ella.

Ahora podemos añadir entradas, como en la Figura 3, a nuestra web de anotaciones, y el número de lineas de código que hemos escrito es muy pequeño. No sólo eso, sino que son muy sencillas de entender.

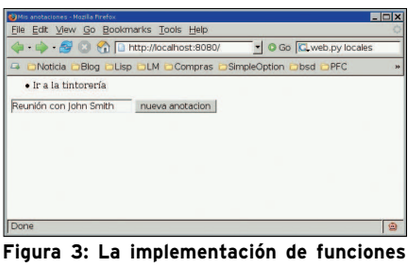

Figura 3: La implementación de funciones con Web.py exige muy poco código.

Conclusión

Con Web.py podemos comenzar a crear webs de manera muy sencilla y rápida. Es un herramienta muy buena para crear prototipos o explorar nuevos conceptos. Y eso que, cuando se escribe este artículo, sólo está disponible la versión 0.13. El sitio web *www.reddit.com* emplea Web.py y su uso se está extendiendo a la vez que más personas se unen a su desarrollo. Desde luego, es un proyecto interesante y debemos estar atentos a su futuro desarrollo. ■

Listado 8: Anota.html

```
01 <html>
02   <head>
03     <title>Mis
   anotaciones</title>
04   </head>
05   <body>
06     <ul>
07       #for anota in
   $anotaciones
08       <li
   id="t$anota.id">$anota.titulo<
   /li>
09       #end for
10     </ul>
11     <form method="post"
   action="/nueva">
12       <p>
13         <input type="text"
   name="titulo" />
14         <input type="submit"
   value="Nueva anotacion" />
15       </p>
16     </form>
17   </body>
18 </html>
```

RECURSOS

[1] Sitio de Aaron Swartz: *http://www.aaronsw.com/*

[2] Descargas de web.py: *http://webpy.org/web.py*

[3] Plantillas Chetah: *http://cheetahtemplate.org/*

[4] Psycopgl, el adaptador Python para PostgreSQL: *http://initd.org/projects/psycopg1*

[5] Sitio web de PostgreSQL: *http://www.postgresql.org*