Ingeniería de Software: Manipulación de Web Forms y Carga de Archivos con Python y WSGI sbre Apache

Trabajar con formularios HTML desde Python y sobre todo, manejar la carga de archivos, es una de las tareas menos sencillas a la hora de crear aplicaciones Web sin utilizar Frameworks. Sin embargo, que no sea la más sencilla no significa que sea imposible.

Desde que Python comenzó a implementarse en el diseño de aplicaciones Web, *frameworks* como **Django** o **Web2Py** han parecido ser la única alternativa posible para desarrollar Software accesible mediante un navegador.

El auge de estos *frameworks* y por sobre todo, la gran publicidad que los programadores menos experimentados en la Ingeniería de Software basado en Web le han hecho a Django, lograron acercar a miles de de usuarios que sin conocimientos de programación en Python, podían desarrollar aplicaciones Web con tan solo aprender a usar el *framework*.

Sabemos que **la humanidad se mueve a base de abusos** de todo tipo y la Ingeniería de Software, no es ajena a ello. Tanto es así que **los usuarios que desarrollan aplicaciones Web utilizando Django, desconocen por completo el lenguaje** y carecen de nociones básicas de programación, a un punto tal que en mucho casos, se llega a creer imposible que puedan desarrollarse aplicaciones Web en Python sin el uso de *frameworks*.

Hace un tiempo publiqué un *paper* en **Debian Hackers**[1], sobre cómo crear un sitio Web en Python bajo Apache sin utilizar *frameworks*[2]. Este artículo pretende continuar esta idea, otorgando al lector las herramientas necesarias para manipular todo tipo de formularios Web incluyendo la carga de archivos al servidor. Toda esta información puede complementarse con el material[3] del **curso de Desarrollo de Aplicaciones Web con Python y MySQL** que dicté durante 2012 y 2013 en cursos.eugeniabahit.com

1 www.debianhackers.net

2 http://www.debianhackers.net/una-web-en-python-sobre-apache-sin-frameworks-y-en-solo-3-pasos

3 http://www.cursosdeprogramacionadistancia.com/static/pdf/material-sin-personalizar-python.pdf

Declarando el enctype correcto

En los formularios HTML, el atributo `enctype` se define cuando el método ha sido establecido como POST y su finalidad es la de indicar en qué forma serán codificados los datos al enviarse el formulario.

```
<form id='something' method='post' action='/some/file' enctype='multipart/form-data'>
</form>
```

El atributo `enctype` puede tener 3 **valores** diferentes:

`multipart/form-data`

Codificación:
ninguna (los caracteres son enviados tal cual están, sin codificación previa)

Uso:
recomendado para manipular la carga de archivos

Delimitador de campos:
aleatorio (se obtiene mediante la clave CONTENT-TYPE del diccionario environ)

```
delimitador = environ['CONTENT-TYPE'].replace('multipart/form-data; boundary=', '')
# retornará algo como: -----------------------------175537584916979984255548673 82
```

Formato recepción de datos:
Ejemplo para un formulario con dos campos de textos (nombre y edad)

```
-----------------------------175537584916979984255548673 82
Content-Disposition: form-data; name="nombre"

Juan Pérez
-----------------------------175537584916979984255548673 82
Content-Disposition: form-data; name="edad"

90
-----------------------------175537584916979984255548673 82--
```

`application/x-www-form-urlencoded`

Codificación:
ASCII Hexadecimal. Los espacios en blanco se reemplazan por el signo +

Uso:
es el valor por defecto de todo formulario. Se recomienda para todos los formularios que no requieran manipular la carga de archivos.

Delimitador de campos:
Signo &

Formato recepción de datos:
Ejemplo para un formulario con dos campos de textos (nombre y edad)

```
nombre=Juan+P%C3%A9rez&edad=90
```

Notar que la vocal “e” acentuada ha sido codificada como %C3% mientras que el espacio en blanco fue sustituido por el signo +

text/plain
Codificación:
ninguna (los caracteres son enviados tal cual están, sin codificación previa)

Uso:
desaconsejado

Delimitador de campos:
retorno de carro

Formato recepción de datos:
Ejemplo para un formulario con dos campos de textos (nombre y edad)

```
nombre=JuanPérez
edad=90
```

El uso del valor text/plain se desaconseja ya que al emplear el retorno de carro como delimitador, dificulta la obtención de datos en bloques de texto que incluyan el salto de línea.

Para el común de los formularios, no es necesario declarar el enctype. Declararlo como multipart/form-data si se trabajará con la carga de archivos.

Manipulando datos desde Python y WSGI

Cuando se reciben datos mediante POST, siempre se necesita tener acceso a dos factores: el nombre de los campos y su correspondiente valor. En Python, la forma lógica de asociar un nombre de campo a un valor, es utilizar un diccionario:

```
datos =  dict(campo1='valor del campo 1', campo2='valor del campo 2')
```

Lo que se debe lograr entonces, es crear un diccionario con dicha información convirtiendo los datos recibidos en un diccionario.

La forma de lograr esta conversión, depende directamente del modelo de codificación con el que la información haya sido enviada, es decir que depende del `enctype` que se haya declarado en el formulario.

Cuando se trabaja con WSGI los datos enviados por `POST` se almacenan en la clave `wsgi.input` del diccionario `environ` (que WSGI entrega a la función `application`) y se obtienen leyendo dicho elemento con el método `read()` tal como se muestra a cotinuación:

```
datos = environ['wsgi.input'].read()
Retornatará algo como: campo1=valor1&campo2=valor2
```

APPLICATION/X-WWW-FORM-URLENCODED

Recordemos que éste, es el valor por defecto de todo formulario. Si no se ha asignado un `enctype`, los datos serán codificados siguiendo este formato.

Como el delimitador de campos es el signo &, podemos obtener una lista donde cada elemento sea un par `clave=valor`

```
datos = environ['wsgi.input'].read().split('&')
```

Luego, en cada elemento de la lista, el signo = separa al nombre del campo de su valor correspondiente con lo que ya tenemos todos los elementos necesarios para armar el diccionario:

```
datos = environ['wsgi.input'].read().split('&')

POST = {}

for par in datos:
    campo, valor = par.split('=')
    POST[campo] = valor
```

Recordemos que los datos son codificados en formato ASCII Hexadecimal y los espacios en blanco, sustituidos por el signo +. Podemos volver los valores a su estado puro, utilizando la función `unquote` del módulo `urllib2` y reemplazando el signo + por un espacio en blanco:

```
from urllib2 import unquote


datos = environ['wsgi.input'].read().split('&')
```

```
POST = {}

for par in datos:
    campo, valor = par.split('=')
    POST[campo] = unquote(valor).replace('+', ' ')
```

De esta forma, habremos obtenido un diccionario como el siguiente:

```
{
    'nombre': 'Juán Pérez',
    'edad': '65',
    'nacionalidad': 'uruguaya'
}
```

Al cual podremos acceder utilizando el nombre de los campos como nombre de clave:

```
nombre = POST['nombre'] if 'nombre' in POST else ''
edad = POST['edad'] if 'edad' in POST else ''
nacionalidad = POST['nacionalidad'] if 'nacionalidad' in POST else ''
```

RECOGER GRUPOS DE DATOS

En cualquier formulario Web, es frecuente tener grupos de opciones bajo un mismo nombre. Es el caso de los campos de tipo checkbox y los de tipo select múltiple:

```
<input type='checkbox' name='grupo_a' value='1' checked>Opción 1<br>
<input type='checkbox' name='grupo_a' value='2'>Opción 2<br>
<input type='checkbox' name='grupo_a' value='3' checked>Opción 3<br>

<select name='combo' size='3' multiple>
    <option value='1'>Opción 1</option>
    <option value='2'>Opción 1</option>
    <option value='3'>Opción 1</option>
    <option value='4'>Opción 1</option>
    <option value='5'>Opción 1</option>
</select>
```

Cuando éste es el caso, **los valores elegidos no son agrupados al momento de enviarse**. Por el contrario, el nombre del campo se repetirá tantas veces como opciones se hayan elegido.

Lo que se pretende entonces es, capturar las opciones elegidas dentro de una lista asignada a la misma clave del diccionario.

Un ejemplo de lo que se quiere obtener, sería como el siguiente:

```
{
    'grupo_a': [1, 3],
    'combo': [2, 4, 5]
}
```

Para lograrlo, si simplemente asignáramos el valor a la clave, el último valor sobre escribiría a los anteriores. Entonces, antes de asignar un valor a una clave, debemos verificar que la clave no exista.

```
POST = {}

for par in datos:
    campo, valor = par.split('=')

    if not campo in POST:
        POST[campo] = unquote(valor).replace('+', ' ')
```

Si la clave existe, podemos encontrar dos posibilidades:

1. Que su valor sea un único dato (el dato fue almacenado cuando `if not campo in POST` fue verdadero)
2. Que su valor sea una lista (cuando `if not campo in POST` fuese falso)

Esto habrá que verificarlo y:

1. En el primer caso, habrá que convertir al valor de dicha clave en una lista y agregarle el valor que ya tenía asignado más el nuevo valor;
2. y en el segundo, agregar directamente el nuevo valor.

Completaremos el código agrupándolo ya mismo en una función y haremos que esta retorne el diccionario con los datos para que pueda ser llamada desde cualquier función que necesite obtener datos enviados desde un formulario:

```
from urllib2 import unquote


def get_simple_form_data(environ):
    datos = environ['wsgi.input'].read().split('&')
    POST = {}

    for par in datos:
        campo, valor = par.split('=')
        valor = unquote(valor).replace('+', ' ')

        if not campo in POST:
            POST[campo] = valor
        else:
            if not isinstance(POST[campo], list):  # aún no es una lista
                POST[campo] = [POST[campo], valor]
            else:  # ya es una lista
                POST[campo].append(valor)
```

```
    return POST
```

Luego, podríamos llamarla desde cualquier otra función:

```
def enviar_form(environ):
    POST = get_simple_form_data(environ)
    # ...
```

MULTIPART/FORM-DATA

Manipular datos en este caso, es mucho más complejo ya que parte de esos datos, es la carga de archivos.
Para poder gestionar archivos cargados desde un formulario, se necesita recurrir a dos módulos adicionales: **tempfile** (para trabajar con archivos temporales) y **cgi** (para facilitar el acceso a todos los datos y obtener toda la información adicional sobre los mismos). De cada módulo, haremos uso de una sola clase: `TemporaryFile` y `FieldStorage` respectivamente.

```
from cgi import FieldStorage
from tempfile import TemporaryFile
```

La clase **FieldStorage** sirve para almacenar una secuencia de campos leyendo, justamente, los datos enviados desde un formulario codificado como `multipart/form-data`. Esta clase genera un diccionario como el que creamos anteriormente de forma manual, pero con una diferencia muy importante: cada elemento de este diccionario contiene información relevante como el tipo MIME del dato e incluye el contenido de archivos incluso cuando éstos sean binarios.
Pero esta clase requiere indefectiblemente que se le indique un puntero y no puede utilizarse `wsgi.input` para ser apuntado como archivo. Es necesario entonces, crear un archivo temporal para poder trabajar con `FieldStorage`, la clase que nos permitirá manejar la carga de archivos.

Lo primero que debemos hacer entonces, es grabar el contenido de `wsgi.input` en un archivo temporal ya que no nos será posible trabajar directamente con `wsgi.input` ya que solo podríamos acceder al nombre del archivo. Grabándolo en un archivo temporal, podremos utilizarlo como puntero desde `FileStorage` para crear el archivo en el servidor. Pero vamos de a poco. Comencemos por grabar el contenido de `wsgi.input` en un archivo temporal:

```
# Crea un archivo temporal y lo abre (por defecto) en modo w+r
archivo_temporal = TemporaryFile()

# Escribimos el contenido de wsgi.input en el archivo temporal
archivo_temporal.write(environ['wsgi.input'].read())

# Movemos al cursor al inicio ya que necesitaremos leer este archivo desde el comienzo
archivo_temporal.seek(0)
```

Una vez creado el archivo temporal, podemos inicializar un objeto `FieldStorage` indicando como puntero al archivo temporal que acabamos de crear:

```
datos = FieldStorage(fp=archivo_temporal)
```

Por defecto, `FieldStorage` obtiene el diccionario `environ` desde el módulo os, pero como estamos trabajando con WSGI, sabemos que el diccionario `environ` es modificado por WSGI, así que se lo pasaremos para que pueda obtener la información correcta:

```
datos = FieldStorage(fp=archivo_temporal, environ=environ)
```

Ahora, la variable `datos` se ha convertido en un diccionario sobre el que podremos iterar para obtener los pares de clave y sus valores correspondientes.

El nombre del campo (o clave para nuestro diccionario), lo obtendremos iterando entonces, sobre la variable datos, tratándola como si fuese un diccionario común y corriente:

```
for clave in datos:
    # ...
```

Antes de obtener el valor de un campo, necesitaremos saber si se trata o no de un archivo, porque de ser así, no solo necesitaríamos el contenido del archivo, sino otros datos como el tipo MIME y el nombre del archivo. Para saber si se trata o no de un archivo, debemos verificar si la propiedad `filename` es `None`:

```
for clave in datos:
    if datos[clave].filename is None:
        # NO ES un archivo
```

Si no se tratase de un archivo podríamos obtener el valor de este campo mediante la propiedad `value`:

```
for clave in datos:
    if datos[clave].filename is None:
        valor = datos[clave].value
```

Si en cambio se tratara de un archivo, quisiéramos que el valor fuese otro diccionario con los siguientes datos: nombre del archivo, tipo MIME y contenido (para poder crearlo posteriormente en el servidor). Para ello, necesitaremos acceder a las propiedades `filename`, `type` y `value` respectivamente:

```
for clave in datos:
    if datos[clave].filename is None:
        valor = datos[clave].value
    else:
        valor = dict(
            filename=datos[clave].filename,
            filetype=datos[clave].type,
            content=datos[clave].value
        )
```

Finalmente, solo resta utilizar las claves y valores para crear nuestro propio diccionario igual que hicimos anteriormente:

```
POST = {}

for clave in datos:
    if datos[clave].filename is None:
        valor = datos[clave].value
    else:
        valor = dict(
            filename=datos[clave].filename,
            filetype=datos[clave].type,
            content=datos[clave].value
        )

    POST[clave] = valor
```

RECOGER GRUPOS DE DATOS

Nuevamente nos podemos encontrar con que tenemos grupos de datos que pertenecen a un mismo nombre. En este caso, el tratamiento difiere del que hemos hecho antes ya que `FieldStorage` nos permite acceder a los grupos de datos como una lista, mediante el método `getlist()`. Lo que haremos entonces es verificar si se trata o no de un grupo de datos, llamando a este método y de ser así, el valor ya no será el obtenido mediante la propiedad `value`, sino el obtenido mediante este método. Al igual que en el caso anterior, vamos a aprovechar a colocar todo en una función que retorne el diccionario POST, para que pueda ser llamada desde cualquier otra que la necesite:

```
from cgi import FieldStorage
from tempfile import TemporaryFile


def get_multipart_form_data(environ):
    archivo_temporal = TemporaryFile()
    archivo_temporal.write(environ['wsgi.input'].read())
    archivo_temporal.seek(0)
    datos = FieldStorage(fp=archivo_temporal, environ=environ)

    POST = {}

    for clave in datos:
        if datos[clave].filename is None:
            lista = datos.getlist(clave)
            valor = datos[clave].value if len(lista) < 2 else lista
        else:
```

```
            valor = dict(
                filename=datos[clave].filename,
                filetype=datos[clave].type,
                content=datos[clave].value
            )

        POST[clave] = valor

    return POST
```

Como podemos ver, valor será igual a lo obtenido mediante la propiedad `value` siempre y cuando la lista tenga menos de dos elementos (es decir, uno o ninguno). De lo contrario, el valor será la lista de datos.

RECOGER GRUPOS DE ARCHIVOS

En el ejemplo anterior, se contempla que los grupos de datos (campos con el mismo nombre y distintos valores) no serán de tipo *file*. Sin embargo, puede suceder que un mismo formulario, tenga más de un campo de tipo *file* con el mismo nombre y aquí, la metodología de implementar `getlist()` para reconocer «grupos de información» ya no será viable. Por consiguiente, la primera condición a evaluar ya no sería preguntar si `datos[clave].filename` es nulo sino, si `datos[claves]` es una lista.

Para que la función no se haga repetitiva y engorrosa, crearemos una función adicional, encargada de retornar los valores como diccionario o *string*, según se trate de un campo de tipo file o no y luego, la llamaremos desde `get_multipart_form_data()`.

```
from cgi import FieldStorage
from tempfile import TemporaryFile


def get_multipart_form_data(environ):
    archivo_temporal = TemporaryFile()
    archivo_temporal.write(environ['wsgi.input'].read())
    archivo_temporal.seek(0)
    datos = FieldStorage(fp=archivo_temporal, environ=environ)

    POST = {}

    for clave in datos:
        if isinstance(datos[clave], list):
            POST[clave] = []
            for elemento in datos[clave]:
                POST[clave].append(traer_valor(elemento))
        else:
            POST[clave] = traer_valor(datos[clave])

    return POST


def traer_valor(dato):
    valor = dict(filetype=dato.type, filename=dato.filename, content=dato.value)
    return dato.value if dato.filename is None else valor
```

Un decorador que decida solo

Hemos logrado abarcar todas las posibilidades en cuanto a manipulación de formularios respecta desde Python con WSGI. Creamos dos funciones que contemplan todas las posibilidades que existen y podemos con ellas, crear un decorador para que mediante un simple llamado, envuelva a cada función que reciba datos desde un formulario.

Si te perdiste la edición anterior sobre wrappers y decoradores en Python, te recomiendo descargues The Original Hacker Nº4 y leas el artículo de página 30.

Para descargar The Original Hacker Nº4 ingresa en http://library.originalhacker.org/biblioteca/revista/ver/20

Crearemos un decorador genérico que identificando el `enctype` del formulario, se encargue de llamar a una u otra función desde el `wrapper`.

No necesitamos modificar las funciones que ya creamos aunque recomiendo hacerlas privadas y utilizar un módulo `.py` para las funciones y el decorador.

```
from cgi import FieldStorage
from tempfile import TemporaryFile
from urllib2 import unquote


def get_post_data(funcion):

    def wrapper(environ):
        _POST = {}

        try:
            if not environ['CONTENT_TYPE'].startswith('multipart/form-data'):
                _POST = __get_simple_form_data(environ)
            else:
                _POST = __get_multipart_form_data(environ)
        except:
            pass

        return funcion(_POST)

    return wrapper


def __get_simple_form_data(environ):
    datos = environ['wsgi.input'].read().split('&')

    POST = {}

    for par in datos:
        campo, valor = par.split('=')
        valor = unquote(valor).replace('+', ' ')

        if not campo in POST:
            POST[campo] = valor
        else:
            if not isinstance(POST[campo], list):
```

```
                POST[campo] = [POST[campo], valor]
            else:
                POST[campo].append(valor)

    return POST


def __get_multipart_formdata(environ):
    POST = {}

    archivo_temporal = TemporaryFile()
    archivo_temporal.write(environ['wsgi.input'].read())
    archivo_temporal.seek(0)
    datos = FieldStorage(fp=archivo_temporal, environ=environ)

    for clave in datos:
        campo = datos[clave]
        if isinstance(campo, list):
            POST[clave] = []
            for elemento in campo:
                POST[clave].append(__get_value(elemento))
        else:
            POST[clave] = __get_value(campo)

    return POST


def __get_value(campo):
    archivo = dict(filetype=campo.type, filename=campo.filename, content=campo.value)
    return campo.value if campo.filename is None else archivo
```

Luego, simplemente, deberás decorar a cualquier función que reciba datos desde un formulario, independientemente del `enctype` que se le haya declarado:

```
@get_post_data
def enviar_form(POST):
    pass
```

Carga de archivos desde formularios y almacenamiento en el servidor

Ya tenemos todo listo para poder manipular los archivos. Lo único que nos resta es algo tan simple como seguir unos cortos pasos.

Crear un directorio privado con permisos de escritura:

```
mkdir private_dir && chmod 777 -R private_dir
```

Grabar el contenido del archivo en uno nuevo dentro el directorio privado:

```
@get_post_data
def enviar_form(environ):
    carpeta = '/ruta/a/directorio/privado/'
    ruta_archivo = carpeta + POST['archivo']['filename']

    with open(ruta_archivo, 'w') as archivo:
        archivo.write(POST['archivo']['content'])
```

Por supuesto que si se debe validar el tipo MIME del archivo, se lo hará mediante `POST['archivo']['filetype']` antes del bloque `with` y solo se ejecutará éste si el tipo MIME es el esperado:

```
@get_post_data
def enviar_form(environ):
    carpeta = '/ruta/a/directorio/privado/'
    ruta_archivo = carpeta + POST['archivo']['filename']
    mime = POST['archivo']['filetype']
    if mime == 'application/pdf':
        with open(ruta_archivo, 'w') as archivo:
            archivo.write(POST['archivo']['content'])
        return 'Archivo guardado'
    else:
        return 'Solo se permiten archivos PDF'
```