Python Web sin frameworks: sobre las sesiones y el acceso restringido

— *Eugenia Bahit agradece a **Hugo** (@huguidugui) por la **revisión ortográfica** de este artículo*

Beaker es un middleware para WSGI que permite, entre otras características, efectuar un manejo de sesiones avanzado en aplicaciones Web creadas con Python. Su implementación es verdaderamente sencilla y se logra en pocos pasos.

Poco a poco, la ingeniería de Software tras haber ganado terreno en el ámbito Web, se está volcando más por Python y cada vez son más los programadores que le pierden el miedo a lo desconocido y se animan a hacer verdadera ingeniería creando aplicaciones sin recurrir a *Django* o a otros *frameworks* de similares características.

En el desarrollo de aplicaciones Web siempre ha sido la restricción de acceso y el manejo de usuarios, un gran hito para los recién llegados. En lenguajes como PHP existe no solo un soporte nativo para el manejo de **«sesiones de usuario»** sino que además, abunda la bibliografía al respecto. Pero en lenguajes como Python, la documentación pareciera reflejar que el manejo de sesiones de usuarios solo sería posible empleando *frameworks*. Y esto no es así.

Pero para profundizar aún más, es necesario primero, comprender de qué se habla en realidad cuando hablamos de «sesiones». Y para ello, recomiendo leer el artículo **«Sesiones ¿qué son realmente y en qué consisten? (desmitificando el concepto de sesiones)»** publicado en la edición Nº6 de The Original Hacker. Una versión del artículo en PDF puede obtenerse desde la siguiente URL: http://library.originalhacker.org/search/sesiones

Sesiones en Python

Como bien comencé diciendo, Python no tiene un soporte «nativo» para la implementación de sesiones y el manejo de *cookies* suele ser algo engorroso con WSGI. Esto, convierte al trabajo de desarrollar un manejador de sesiones en una tarea larga y compleja de emprender, aunque no imposible. Aquí, abarcaremos el manejo de sesiones empleando la librería **Beaker** (http://beaker.readthedocs.org/en/latest/).

Primeros pasos

Se utilice Beaker, cualquier otra librería o se desarrolle una propia, las herramientas con las cuáles debe contarse son:

Una base de datos con al menos un usuario de prueba en una tabla de usuarios

```
-- Reemplazar foobar por el nombre de la base de datos a utilizar
USE foobar;

-- Estructura base de la tabla usuarios
CREATE TABLE usuarios (
    usuario_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(25) NOT NULL,
    password VARCHAR(32) NOT NULL
) ENGINE=InnoDB;

-- Usuario de prueba
INSERT INTO usuarios
            (username, password)
     VALUES ('prueba', 'c893bad68927b457dbed39460e6afd62');
```

c893bad68927b457dbed39460e6afd62 es el ***hash* MD5** de la clave **prueba**.

Luego, necesitaremos un **formulario de inicio de sesión**:

```
<form method='POST' action='/usuarios/validar'>
  Usuario: <input type='text' name='u' id='username'><br>
  Clave: <input type='password' name='p' id='password'>
  <button type='submit'>Ingresar</button>
</form>
```

Por consiguiente, un módulo de usuarios (**`usuarios.py`**) con un **recurso público encargado de mostrar el formulario**:

```
def ingresar():
    with open('/ruta/a/form_login.html', 'r') as archivo:
        return archivo.read()
```

Una **función validar** (en el módulo de usuarios) encargada de verificar que los datos ingresados en el formulario anterior, coincidan con el de un usuario en la base de datos y en caso afirmativo, invoque al iniciador de sesión o de lo contrario, muestre nuevamente el formulario:

```
from hashlib import hashlib

from core.postdata import POSTDataHandler, POSTDataCleaner
from core.dblayer import run_query
from core.sessions import iniciar  # TODO
```

```
@POSTDataHandler
def validar(_POST):  # application deberá enviarle environ como parámetro
    usuario = POSTDataCleaner().sanitize_string(_POST['u'])
    clave = hashlib.md5(_POST['p']).hexdigest()

    sql = """SELECT usuario_id FROM usuarios
             WHERE username = '%s'
               AND password = '%s'""" % (usuario, clave)
    resultados = run_query(sql)

    return iniciar() if len(resultados) == 1 else ingresar()
```

El archivo de sesiones (**sessions.py**) deberá hospedarse a nivel del *core* y contar con al menos las funciones para iniciar una sesión, destruirla y verificar una sesión activa (completaremos las funciones más adelante):

```
def iniciar():
    pass


def destruir():
    pass


def verificar():
    pass
```

Implementando Beaker

Ante todo, para poder utilizar Beaker será necesario instalarlo desde PyPi:

```
sudo pip install beaker
```

Luego, serán necesarios algunos cambios en nuestro controlador y en la función `application`. El primer paso será **importar el *middleware***:

```
from beaker.middleware import SessionMiddleware
```

A continuación, **la función `application` cambiará de nombre** (puede elegirse cualquier nombre) ya que deberá ser envuelta por el *middleware* y éste, ser quien responda ante la llamada de WSGI:

```
def ~~application~~(environ, start_response):
def start(environ, start_response):
```

Finalmente, **el *middleware* envolverá a la ex función `application`** y éste será quien actúe ante la invocación de WSGI:

```
# Configuraciones necesarias para el Middleware
beaker_dic = {
    'session.type': 'file',
    'session.cookie_expires': True,
    'session.auto': True,
    'session.data_dir': '/tmp/sessions',
}


application = SessionMiddleware(start, beaker_dic)
```

El archivo de sesiones

Ahora completaremos las funciones definidas en el archivo de sesiones. Las **variables de sesión** generadas por *Beaker*, se encontrarán disponibles en la **clave `beaker.session` del diccionario `environ`**. La función de inicialización de sesión se encargará de crear variables de sesión que nos ayuden a identificar si un usuario se encuentra *«logueado»* en nuestro sistema. Crearemos 2 variables de sesión: `logged` (*booleana*) y `username` (*string*). Para lograrlo, será necesario contar con el diccionario `environ` por lo cual, primero, modificaremos la función `validar()` de `usuarios.py` (para que nos pase este parámetro):

```
@POSTDataHandler
def validar(_POST, environ):
    usuario = POSTDataCleaner().sanitize_string(_POST['u'])
    #...
    return iniciar(environ, usuario) if len(resultados) == 1 else ingresar()
```

Y ahora sí, completaremos la **función `iniciar`** creando las nuevas variables de sesión:

```
def iniciar(environ, username):
    environ['beaker.session']['logged'] = True
    environ['beaker.session']['username'] =  username
```

Para **destruir una sesión** bastará con invocar al método `delete()` -creado por Beaker-:

```
def destruir(environ):
    environ['beaker.session'].delete()
```

Y para finalizar este archivo, **la función que verifica la sesión** será la «frutilla del postre» :)
Esta función se encargará de buscar las variables de sesión dentro de `beaker.session` en `environ`. De encontrarla nada malo podría pasar pero de no encontrarla, deberá destruir «las eventuales variables de sesión residuales que puedan quedar» e invocar a la función de solicitud de datos del módulo de usuarios. Por esto motivo, la **función verificar** será un decorador que envuelva a aquellas funciones cuyo acceso se

encuentre restringido. Veamos cómo lo logrará:

```
from usuarios import ingresar


def verificar(funcion_restringida):

    def wrapper(*args, **kwargs):
        # Comienzo diciendo que el usuario no está logueado hasta comprobarlo
        usuario_logueado = False

        # Obtengo el diccionario environ (suponiendo es el primer argumento)
        environ = args[0]

        # Verifico si las variables de sesión están disponibles
        existe_var_logged = 'logged' in environ['beaker.session']
        existe_var_uname = 'username' in environ['beaker.session']
        # Si están disponibles, modifico el valor de usuario_logueado con el de logged
        if existe_var_logged and existe_var_uname:
            # obtengo el valor de la variable logged
            usuario_logueado = environ['beaker.session']['logged']

        # Si el usuario está logueado, devuelvo la función decorada
        if usuario_logueado:
            return funcion_restringida(*args, **kwargs)
        # Sino, destruyo residuales y devuelvo la función para loguearse
        else:
            destruir()
            return ingresar()

    return wrapper
```

Switcheando solicitudes en el controlador

Una vez preparados los módulos de usuarios y sesiones y adaptado el controlador, habrá que *switchear* las URL teniendo en cuenta la siguiente tabla de referencia:

Acción/Funcionalidad	URL	Función a invocar (parámetros)
Ingresar al sistema (muestra formulario de *logueo*)	`/usuarios/ingresar`	`usuarios.ingresar()`
Verificar los datos del *loguin* (*action* del formulario anterior)	`/usuarios/validar`	`usuarios.validar(environ, environ)`
Finalizar una sesión	`/usuarios/salir`	`sessions.destruir(environ)`

Restringir el acceso

Para restringir el acceso a una función determinada solo bastará con decorarla con la función verificar del módulo de sesiones:

```
from core.sessions import verificar

@verificar
def funcion_con_acceso_restringido():
    pass
```