Tema 14: Gestión de usuarios

Contenido

1 Introducción

Google Application Engine (GAE) es un *framework* suministrado por *Google*, para Python (entre otros lenguajes de programación),que proporciona una forma sencilla de crear aplicaciones web. Además, también permite acceder de forma sencilla a la infraestructura de *Google*.

Uno de las funcionalidades típicas en aplicaciones web que realmente son útiles es el de la gestión de usuarios. La aplicación web se comporta de una manera u otra según un usuario u otro hayan hecho *login*.

2 "demousers" en GAE

La aplicación más sencilla es aquella que es capaz de soportar que el usuario haga *login*, y cuando lo haya hecho, le salude y le permita hacer *logout*.

2.1 El archivo app.yaml

El archivo app.yaml no sufre ningún cambio, pues el manejo de usuarios ya está incluido en el *framework* GAE, por lo que no es necesario incluir ninguna librería o realizar otra acción.

```
application: demousers
version: 1
runtime: python27
api_version: 1
threadsafe: yes

handlers:
- url: /
  static_files: index.html
  upload: index\.html

- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: .*
  script: main.app

libraries:
- name: webapp2
  version: "2.5.2"
- name: jinja2
  version: "2.6"
```

2.2 El archivo main.py

Dada una aplicación creada desde cero, sólo es necesario modificar el archivo *main.py* tal y como se explica a continuación.

```
from google.appengine.api import users
import webapp2

class MyHandler(webapp2.RequestHandler):
    def get(self):
        user = users.get_current_user()
        if user:
            greeting = str.format(
                                    "Welcome, {0}! (<a href=\"{1}\">sign out</a>)",
                        user.nickname(),
                        users.create_logout_url('/'))
        else:
            greeting = str.format(
                                    "<a href=\"{0}\">Sign in or register</a>.",
                        users.create_login_url('/'))

        self.response.out.write(
                        str.format( "<html><body>{0}</body></html>", greeting ) )


app = webapp2.WSGIApplication([
    ('/', MyHandler),
], debug=True)
```

La clave en el código superior es la llamada a *get_current_user()*. Esto permite saber si la aplicación está funcionando con un usuario que ha accedido a ella o no. Así, si *user* (el objeto obtenido) es **None**, entonces ningún usuario se ha dado de alta. En caso de que exista un usuario, entonces podemos obtener, mediante su método *nickname()*, el nombre de login de dicho usuario. Los métodos *create_login_url(s)* y *create_logout_url(s)*, crean una URL desde la que el usuario puede acceder o salir de la aplicación. Estos dos métodos aceptan un *path* al que volver tras haber autenticado (o descartado) al usuario. En el caso de esta aplicación de ejemplo tan sencilla, se vuelve siempre a la misma funcionalidad, la raíz.

3 La clase User

Esta clase no permite acceder a mucha información, pero son interesantes los métodos en la tabla inferior. Para obtener un nombre visualizable con el que referirse al usuario, lo mejor es usar *nickname()*. Para, por ejemplo, almacenar un usuario como parte de un objeto en el *datastore*, lo más adecuado es *user_id()*.

No es recomendable guardar el objeto **User** en el datastore, pues incluye mucha más información de la necesaria. Por ejemplo, el *email* también se guarda en el objeto **User**, además de *user_id*, mientras que el primero puede cambiar, y el segundo nunca cambia. Tampoco se debe hacer derivar nuestra propia clase **User** o **Usuario** de la clase **User** de GAE, ya que es parte de la API. Así, lo

más adecuado es ceñirse siempre al *user_id*.

Método	Explicación
nickname()	Devuelve el nick de un usuario. Se trata de la parte anterior a '@' en una dirección de correo electrónico de Google.
user_id()	Devuelve la identificación única del usuario.
email()	Devuelve el *email* del usuario.
create_login_url(s)	Crea una *url* que llevará el control de la aplicación a la identificación del usuario. A continuación, se moverá a la URL pasada entre paréntesis.
create_logout_url(s)	Crea una *url* que llevará el control de la aplicación a la desconexión del usuario. A continuación, se moverá a la URL pasada entre paréntesis.

En cuanto al módulo *users*, tiene algunos métodos que son interesantes.

Método	Explicación
get_current_user()	Devuelve al usuario actual, como un objeto de la clase **User**.
is_current_user_admin()	Devuelve **True** si el usuario actual es administrador.

Para establecer los usuarios administradores, se utiliza la consola de administración de la aplicación (ver referencias).

4 Configuración mediante el archivo app.yaml

El archivo *app.yaml* permite configurar al detalle la aplicación. Entre otras cosas, también permite establecer el acceso con *login* como necesario para algunas páginas, lo que permite evitarse el trabajo repetitivo de comprobar en el método *get()* que existe un usuario autenticado en la aplicación.

```
handlers:
- url: /profile/.*
  script: user_profile.app
  login: required
  auth_fail_action: redirect
```

En la porción de código del archivo *app.yaml* que aparece más arriba, se usa la propiedad *login* para todos accesos dentro de la rama *profile* de la aplicación. Para la propiedad *login*, los valores posibles son *optional* (por defecto), *required* (que comprueba que el usuario se haya autenticado, y en caso contrario, hace lo que diga la propiedad *auth_fail_action*.

La propiedad *auth_fail_action* puede tener los siguientes valores: *redirect* (por defecto, que lleva al usuario a una página de autenticación), o *unauthorized*, que retorna un error 401 con un mensaje de error.

Finalmente, es posible establecer también la configuración *admin* en la propiedad *login*. Esto no solo exige que el usuario esté autenticado, sino que sea uno de los usuarios administradores de la misma. Los usuarios administradores son los que pueden acceder a la consola de la aplicación (ver referencias más abajo), por lo que es allí a donde dirigirse para añadir a los usuarios con capacidades administrativas (más allá del creador de la aplicación).

5 “Salúdame” con control de usuarios

De nuevo retomamos el ejemplo de la simple aplicación que saluda, añadiéndole control de usuarios. Es decir, el usuario no introducirá su nombre, sino que se dará de alta en la aplicación, y esta recuperará el nombre de la información de usuario. Guardará los datos del usuario, si no han sido guardados ya, y permitirá que el usuario salga de la misma.

```
application: saludame
...
handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: .*
  script: main.app

libraries:
- name: webapp2
  version: "2.5.2"
- name: jinja2
  version: "2.6"
```

El archivo *app.yaml* es el mismo de siempre para aplicaciones que emplean Jinja2, tal y como se puede observar más arriba. No es posible confiar en un HTML estático el inicio de la aplicación, puesto que es necesario que el usuario realice un *login*. El archivo de plantilla *index.html* se muestra a continuación.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Login users for salutation</title>
</head>
<body>
    <b>Please <a href="{{user_login}}">login</a>.</b>
</body>
</html>
```

El código que se encarga del manejo de la aplicación, *main.py* (se muestra a continuación), tiene dos partes funcionales: "/", que permite que el usuario se registre en la aplicación; y "/list", que, una vez que el usuario se ha registrado, lista los usuarios a los que ha saludado previamente.

```
import webapp2
import os
import time
from google.appengine.ext import ndb
from google.appengine.api import users
import jinja2

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.dirname(__file__)),
    extensions=["jinja2.ext.autoescape"],
    autoescape=True)

class User(ndb.Model):
    id_user = ndb.StringProperty(required=True)
    name = ndb.StringProperty(required=True)


class MainHandler(webapp2.RequestHandler):
    def get(self):
        user = users.get_current_user()

        if user:
            self.redirect("/list")
        else:
            labels = {
                "user_login": users.create_login_url("/")
            }

            template = JINJA_ENVIRONMENT.get_template("index.html")
            self.response.write(template.render(labels))
```

```
class ListHandler(webapp2.RequestHandler):
    def get(self):
        user = users.get_current_user()

        if user == None:
            self.redirect("/")
        else:
            # Look for the user's information
            user_id = user.user_id()
            name_info = user.nickname()
            stored_user = User.query(User.id_user == user_id)
            if stored_user.count() == 0:
                # Store the information
                img = User(id_user = user_id, name = name_info)
                img.put()
                time.sleep(1)

            people = User.query().order(User.name)
            labels = {
                "people": people,
                "user_logout": users.create_logout_url("/")
            }

            template = JINJA_ENVIRONMENT.get_template("answer.html")
            self.response.write(template.render(labels))

app = webapp2.WSGIApplication([
    ('/', MainHandler),
    ('/list', ListHandler)
], debug=True)
```

Tal y como se aprecia al final del código, de la parte funcional raíz se encarga **MainHandler**, mientras que de la parte funcional "/list" se encarga **ListHandler**. Ambos *handlers* solamente definen el método *get()*, puesto que no hay manejo de formularios en este ejemplo.

En los casos en los que a) En **MainHandler** el usuario ya está dado de alta en la aplicación, y b) en **ListHandler** el usuario no está dado de alta, no se puede continuar. Es por eso que se utiliza el método *self.redirect(path)*, que permite que la aplicación se redirija a un *path* determinado. Nótese que, hasta que el método no termina, la redirección no se produce.

Finalmente, la plantilla *answer.html* se utiliza para mostrar los resultados.

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Users results</title>
</head>
<body>
    {% if people.count() > 0 %}
    <ul>
    {% for person in people %}
        <li>Hello, {{person.name}}</li>
    {% endfor %}
    </ul>
    {% else %}
    <b>No people found.</b>
    {% endif %}
    <br><b>You can <a href="{{user_logout}}">logout</a> now.</b>
</body>
</html>
```

Se trata de una plantilla que ya conocemos, en la que se listan los usuarios que se han registrado en el sistema, y se da la opción de salir de la aplicación.

6 Referencias

- Tutorial de Python 3: http://docs.python.org/3/tutorial/ (accedido en 03/2015)
- Documentación de GAE (accedido en 03/2016): https://cloud.google.com/appengine/docs
- Documentación sobre manejo de usuarios en GAE (accedido en 04/2016): https://cloud.google.com/appengine/docs/python/users/
- Referencia de la clase **User** (accedido en 04/2016): https://developers.google.com/appengine/docs/python/users/userclass
- Como configurar la aplicación a través de *app.yaml* (accedido en 04/2016): https://cloud.google.com/appengine/docs/python/config/appconfig
- Consola de administración (accedido en 04/2016): https://console.cloud.google.com/appengine/

7 Ejercicios

1. *Recuerda el siguiente ejercicio:* Crea una aplicación que permita generar facturas (en HTML). La aplicación pedirá los datos del emisor de la factura (CIF, nombre, dirección, población, provincia, código postal, país, persona de contacto, email, y teléfono), y los datos del cliente (con los mismos datos que los del emisor de la factura). A continuación, permitirá introducir una línea de detalle: concepto, precio, por unidad, unidades, importe bruto, porcentaje de IVA a aplicar e importe total. Valida los datos de entrada de tal forma que no haya campos vacíos en la factura. La fecha de la factura será la fecha del momento de creación de la misma. *Añade la funcionalidad de que cada usuario pueda crear sus propias facturas.*
2. Crea una aplicación que permita guardar un directorio telefónico que registre nombre, e.mail, teléfono, y la foto del usuario. Cada usuario tendrá su propio directorio telefónico, y podrá anadir, modificar y eliminar usuarios.
3. Un perito especializado en accidentes automovilísticos necesita una aplicación que le permita gestionar su trabajo con cada siniestro. Así, la aplicación permitirá que cada perito lleve sus casos de manera diferenciada. Permitirá añadir y modificar los siniestros que perita, de forma que guarda de cada caso el nombre, NIF, teléfono, fecha, compañía de seguros y una foto del siniestro que deberá reescalar a 200x200. El perito podrá acceder a un listado de todos los siniestros, además de poder seleccionar una fecha determinada y listar los peritajes de ese día.