Tema 13: Almacenamiento (Datastore)

Contenido

1 Introducción

En cualquier aplicación es necesario guardar datos de manera que sea posible recuperarlos, y operar sobre ellos, más tarde. La solución que proporciona GAE a este problema es una base de datos orientada a objetos. En ella, se pueden guardar objetos que más tarde se pueden recuperar en su totalidad o según cumplan alguna condición.

2 "Salúdame" en GAE, con almacenamiento

En las siguientes secciones se volverá a retomar el ejemplo del saludo para ejemplificar cómo se usa el almacenamiento, de manera que se guarden los usuarios que ya han saludado antes.

2.1 El archivo app.yaml

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

```
application: saludame
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*

Este archivo proporciona la lógica de negocio de la aplicación. Deberemos crear una nueva clase, que servirá como modelo para los registros de la base de datos.

```
from google.appengine.ext import ndb

class Salute(ndb.Model):
        name = ndb.StringProperty(required = True)
        time = ndb.DateTimeProperty(auto_now_add = True)
```

La nueva clase **Salute** deriva de **ndb.Model,** lo cual le proporciona mucha funcionalidad relacionada con el almacenamiento y recuperación de objetos **Salute**, como se verá proximamente. A partir de este modelo, el motor GAE es capaz de soportar una infraestructura total de almacenamiento y recuperación. Nótese que los futuros campos de cada registro se crean como si fueran atributos de clase, a los que se iguala un objeto **StringProperty**, **IntegerProperty**, **FloatProperty**, **DateTimeProperty**... según el tipo de dato que vayan a soportar. Los constructores de estas propiedades soportan varios parámetros, algunos de los cuales se muestran en el ejemplo.

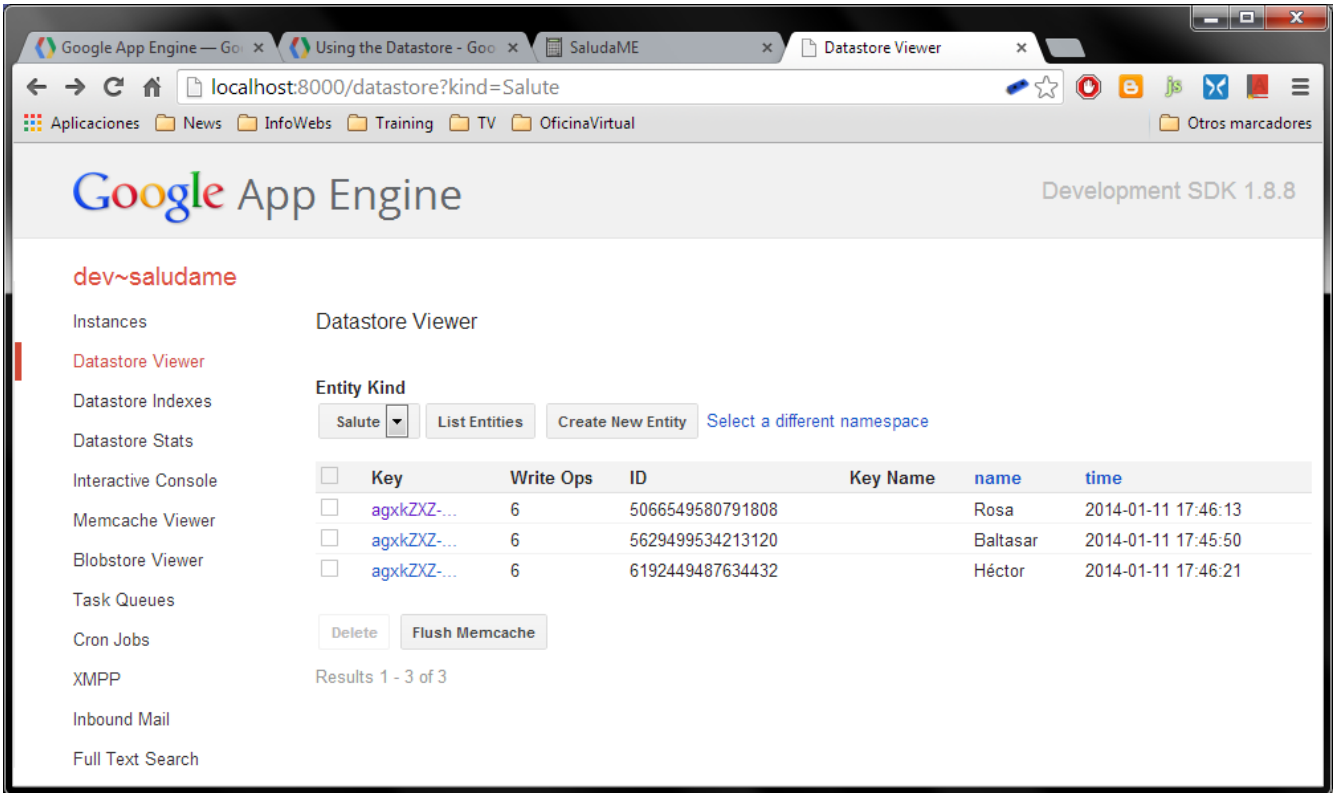
2.3 Guardando registros en la base de datos

Para guardar nuevos registros, el proceso no podría ser más fácil: se crea un nuevo objeto de la clase **Salute**, y se llama a su método *put()*.

```
# Store the answer
salute = Salute(name = self.name);
salute.put();
```

Una vez que se hayan almacenado unos cuantos objetos en la base de datos, es posible visualizarlos mediante el *Datastore Viewer* del gestor de Google. Como siempre, la consola de desarrollo está disponible, una vez arrancada cualquier aplicación, en http://localhost:8000.

La consola tiene una sección, accesible desde la lista de opciones de la izquierda, llamada Visor del almacenamiento de datos (*Datastore Viewer*). Desde esta opción, se pueden manejar los objetos almacenados. Se puede observar como cada uno tiene una clave distinta, y también cómo el visor solamente mostrará aquellos objetos pertenecientes a una determinada clase. Desde este visor, realmente se pueden incluso crear nuevos elementos, o eliminar algunos o todos de los objetos guardados.



2.4 Recuperando registros de la base de datos

Para recuperar registros, es necesario utilizar el método *query()*, que soporta varias funcionalidades para filtrar los registros. Sin parámetros, devuelve todos los registros de esa clase.

```
salutations = Salute.query();                          # recupera todos
salutations = Salute.query(Salute.name == “a”);        # recupera aquellos cuyo
                                                       # nombre sea 'a'
```

Es posible especificar ordenaciones, de una forma también muy sencilla:

```
salutations = Salute.query().order(Salute.time);
```

De esta forma, se obtienen todos los saludos, y se ordenan por el campo *time*. Si en el lugar del parámetro se hubiera indicado *-Salute.Time*, la ordenación sería descendente.

El archivo main.py queda entonces como sigue:

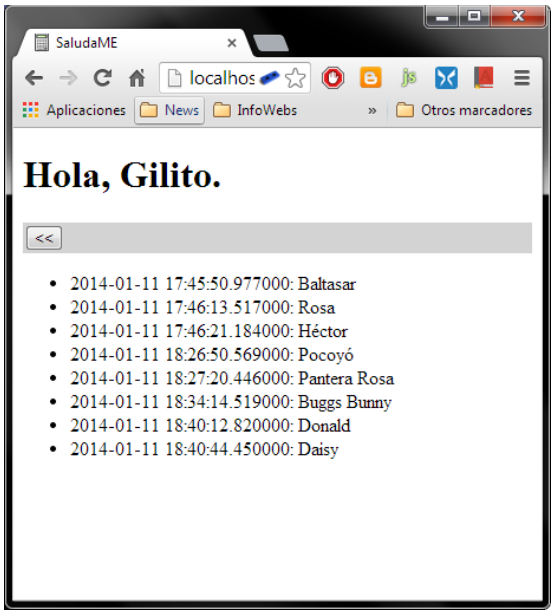
```
import os
import jinja2
import webapp2

from google.appengine.ext import ndb

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

class Salute(ndb.Model):
      name = ndb.StringProperty(required = True)
      time = ndb.DateTimeProperty(auto_now_add = True)

class SaluteHandler(webapp2.RequestHandler):
      def get_input(self):
            self.name = self.request.get("name", "pobrecito hablador")

      def post(self):
            self.get_input()

            # Get all previous answers
            salutations = Salute.query().order(Salute.time);

            # Store the answer
            salute = Salute(name = self.name);
            salute.put();

            # Prepare the answer
            template_values = {
                  'name': self.name,
                  'salutations': salutations,
            }

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

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

Hay varias diferencias: las que se han desgranado en cuanto a manejo de la base de datos, y además, un nuevo campo para la plantilla de respuesta. Esta modificación se realizará en la siguiente sección.

2.5 El archivo answer.*html*

Es necesario modificar el archivo *answer.html*, puesto que ahí, además del saludo propiamente dicho, se visualizará la lista de usuarios que han saludado. Para ello, se utiliza la característica de listas de JINJA2, que, como veremos, es muy potente. A continuación, el archivo *answer.html*:

```
<!DOCTYPE html>
<html>
<head>
    <title>SaludaME</title>
</head>
<body>
    <p>
    <h1>Hola, {{name|capitalize}}.</h1>
    <form id="frmResult" style="background-color:lightgray">
        <input type="button" onClick="javascript:history.go(-1);" value="<<">
    </form>
    </p>
    <p>
        {% if salutations.count() > 0 %}
        <ul>
            {% for salute in salutations %}
                <li>{{salute.time}}: {{salute.name}}</li>
            {% endfor %}
        </ul>
        {% endif %}
    </p>
</body>
</html>
```



La parte resaltada en negrita es lo realmente importante. Se crea una lista mediante un bucle for, que mostrará cada elemento en una entrada nueva. Para este tipo de bucles, JINJA2 soporta una notación parecida a la de los campos normales, que se marca con {% como apertura y %} como cierre. La sintaxis del *for* es igual a la que se puede soportar en Python, aunque eso sí, termina con una marca **endfor**. De esta forma, dado que el valor que recibe a través de *salutations* es una lista de objetos de la clase **Salute**, es posible acceder a sus campos *name* y *time*, con lo que se crea la lista de una manera muy sencilla.

3 Almacenando imágenes

Google Application Engine permite el manejo de imágenes en su *framework*. Para ello, sin embargo, es necesario tener instalado PIL (*Python Image Library*, la instalación varía de un sistema operativo a otro), de manera que sea posible probar la aplicación en el simulador local. Además, será necesario indicar la librería en el archivo YAML.

Lo explicado aquí sirve para todo tipo de archivos, pero se hace hincapié en las imágenes porque es necesario realizar un tratamiento más completo, debido a que normalmente se deseará mostrarlas más tarde.

3.1 "Salúdame", con imágenes en GAE

La aplicación más sencilla es aquella que es capaz de soportar que el usuario introduzca su nombre y su foto o avatar, y cuando lo haya hecho, le mostrará los datos, así como los datos del resto de personas que le precedieron.

3.1.1 El archivo app.yaml

El archivo app.yaml no sufre ningún cambio, pues el manejo de imágenes ya está incluido en el *framework* GAE, por lo que no es necesario incluir ninguna librería (aparte de PIL) o realizar otra acción. Como siempre, … indica que faltan líneas de uso común.

```
application: saludame
...

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"
- name: PIL
  version: latest
```

3.1.2 El archivo index.html

Esta aplicación arranca directamente con un formulario estático, sin más preámbulos. Este formulario viene detallado a continuación.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Image upload Demo</title>
</head>
<body>
    <form   method="post"
            action="/list"
            enctype="multipart/form-data">
    Name: <input type="text" name="edName" />
    Upload file: <input type="file" name="fileImage" />
    <br><input type="submit" value="send"/>
    </form>
</body>
</html>
```

Del formulario anterior es importante destacar la necesidad de declarar la cláusula *enctype="multipart/form-data"*, de manera que sea capaz de enviar imágenes o en general, archivos binarios. Como siempre, las cláusulas *name*, permiten identificar los campos del formulario del lado del servidor.

3.1.3 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.

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

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

class Image(ndb.Model):
    name = ndb.StringProperty(required=True)
    image_bin = ndb.BlobProperty(required=True)
```

```
class MainHandler(webapp2.RequestHandler):
    def post(self):
        # Store the added image
        image_file = self.request.get("fileImage", None)
        name_info = self.request.get("edName", "")

        if (name_info != ""
        and image_file != None):
            img = Image(name = name_info)
            img.image_bin = images.resize(image_file, 64, 64)
            img.put()
            time.sleep(1)

        # Retrieve images
        #     there is the need to encode the image data as base64:
        #     person.image_bin.encode("base64").
        #     It's done here in the JINJA2's template
        people = Image.query().order(Image.name)
        labels = {
            "people": people
        }

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

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

Si los datos están completos, se guarda el nuevo objeto en el **datastore**. Este objeto tiene la peculiaridad de que contiene un dato que es una sucesión de bytes. Así, para poder guardar datos de este tipo, es necesario especificar un **BlobProperty**. GAE permite guardar registros de hasta un límite de 1MB en un *datastore* (en el ejemplo, esto incluye el nombre, así que serán imágenes ligeramente más pequeñas). Como se reescala la imagen antes de guardarla para que ocupe 64x64 pixeles, es complicado que se alcance este tamaño, aunque sin embargo sí es posible cuando guardamos imágenes sin reescalar, como panorámicas, etc. En ese caso, el *datastore* no será la mejor opción, debiendo decantarnos por un *blobstore* o el almacenamiento en la nuble de Google.

Es interesante tener en cuenta que la transformación *resize(img, w, h)* también cambia el formato de la imagen a *png* por defecto.

3.1.4 El archivo answer.html

Se utiliza una template de JINJA2 para mostrar los resultados. Es necesario tener en cuenta que es obligatorio codificar los datos en BASE64, por lo que la template tendrá que ocuparse de eso también.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Image results</title>
</head>
<body>
    {% if people.count() > 0 %}
    <ul>
    {% for person in people %}
        <li><img src="{{person.image_bin.encode("base64")}}">
        Hello, {{person.name}}</li>
    {% endfor %}
    </ul>
    {% else %}
    <b>No people found.</b>
    {% endif %}
</body>
</html>
```

Para poder visualizar imágenes directamente a partir de un bloque de bytes, es necesario utilizar la cláusula *src* de las imágenes de una manera un tanto especial. Se indica el formato de imagen deseado, la codificación, y a continuación, separados por una coma, los bytes que componen la imagen en sí: *src=" xxxx"*, en donde "xxx" es la sucesión de bytes.

3.2 El módulo images

Este módulo contiene no solo la clase **Image**, que permite muchas modificaciones de imágenes, sino varias funciones que pueden ser llamadas directamente, como por ejemplo las siguientes.

Método	Explicación
resize(img, h, w, output_encoding=PNG)	Cambia una imagen para que sus nuevas dimensiones sean *w* (ancho) x *h* (alto). Por defecto, la devuelve como PNG.
rotate(img, d, *output_encoding=PNG)*	Rota una imagen d grados. Por defecto, la devuelve como PNG.
crop(img, left_x, top_y, right_x, bottom_y, output_encoding=PNG)	Recorta una imagen dados los parámetros *x* e *y* izquierda-superior, derecha-inferior. Por defecto, la devuelve como PNG.

4 Profundizando en el Datastore

En las siguientes secciones se profundiza sobre las posibilidades que ofrece **ndb**, el Datastore de GAE. Es importante tener en cuenta que, según el uso que se le dé a los datos (las *queries* que realicemos), GAE se encargará por si mismo en crear los índices necesarios (de ahí el archivo *app.index* autogenerado), de manera que no es necesario que lo hagamos nosotros por nuestra cuenta.

4.1 Posibles campos a utilizar en una entidad

Como ya se ha visto, para crear una nueva entidad, es necesario crear una nueva clase derivada de **ndb.Model**,como en el ejemplo siguiente.

```
class Persona(ndb.Model):
      nombre = ndb.StringProperty()
      edad = ndb.IntegerProperty()
```

Existe una limitacióm importante, y es que no es posible crear un campo llamado *key*. Esto es debido a que GAE tiene que crear un identificador único para cada objeto, y lo guarda precisamente en un atributo llamado *key*.

Un resumen de los tipos que realmente soporta GAE se describen a continuación. Nótese que más o menos los tipos se corresponden con los datos que se pueden utilizar en Python, pero no exactamente. Por ejemplo, un número en Python automáticamente puede superar las capacidades de la máquina (por ejemplo, 32 o 64 bits), mientras que **IntegerProperty** está limitado a 64bits.

Tipo de campo	Descripción
`IntegerProperty`	Número entero de 64bits.
`FloatProperty`	Número real de doble precisión.
`BooleanProperty`	Booleano.
`StringProperty`	Texto *unicode*, hasta 1500 bytes, indexado.
`TextProperty`	Texto ilimitado, no indexado.
`GeoPtProperty`	Tiene los atributos *lat* y *lon*. Se pueden asignar en el momento de la creación empleando dos flotantes, uno por cada coordenada.
`DateTimeProperty`	Equivalente a la clase **datetime** del módulo estándar **datetime**.
`DateProperty`	Equivalente a la clase **date** del módulo estándar **datetime**.
`TimeProperty`	Equivalente a la clase **time** del módulo estándar **datetime**.

Tipo de campo	Descripción
`PickleProperty`	Un objeto de Python cualquiera, sin indexación.
`ComputedProperty`	Un campo que es calculado a partir de otros campos.
`BlobProperty`	El campo guarda un objeto cualquiera, como una secuencia de bytes.

4.1.1 Manejo de campos relacionados con la fecha/hora

En GAE, las fechas no se almacenan según el huso horario local, sino siguiendo UTC. Esto es relevante si la aplicación, por poner un ejemplo, almacena horas obtenidas a través de *datetime.now()* para guardar una cita.

En el momento de crear un campo relacionado con DateTimeProperty y sus relacionadas, es posible especificar los siguientes parámetros:

- *auto_now_add*: Se asigna la fecha/hora actual cuando se crea el objeto en el *datastore*.
- *auto_now*: Se asigna la fecha/hora actual al crear un nuevo objeto y cada vez que se modifique.

4.1.2 Campos calculados (Computed)

Los campos calculados guardan una pequeña función que hace que el valor del campo se actualice cuando se hace un *put()* al almacenamiento de la entidad. Estos campos son útiles si se van a pedir mediante una *query* directamente, en otro caso (simplemente se usa en el programa), es mucho mejor crear una función en Python que realice la tarea.

```
class Persona(ndb.Model):
      nombre = ndb.StringProperty()
      edad = ndb.IntegerProperty()
      telefonos = ndb.IntegerProperty(repeated = True)
      felefono_ppal = ndb.ComputerProperty(
                                    lambda self: self.telefonos[0]
                                          if len(self.telefonos) > 0 else None)

p1 = Persona(“Baltasar”, 18, [988368891, 988387000])
p1.put()
```

Así, en el momento de guardar la nueva entidad, el *telefono_ppal* pasaría a tener el valor 988368891, pues es el primero de la lista, y el primero de la lista se considera el teléfono principal.

4.1.3 Campos que admiten repeticiones

En general, estos campos son listas de un tipo determinado, que es el principal en el que el campo es declarado. Por ejemplo, una persona podría tener varios números de teléfono. Cada persona, de hecho podría tener un número variable de teléfonos.

```
class Persona(ndb.Model):
      nombre = ndb.StringProperty()
      edad = ndb.IntegerProperty()
      telefonos = ndb.IntegerProperty(repeated = True)

p1 = Persona(“Baltasar”, 18, [988368891, 988387000])
p1.put()
```

Cuando se cambia el campo *telefonos*, es posible modificar la lista como tal, o cambiar la lista por una totalmente nueva. El orden se mantiene.

4.1.4 Parámetros de creación de nuevos campos

Es posible modificar el comportamiento de los campos que conforman la entidad. De hecho, ya hemos visto anteriormente que se puede marcar un campo como requerido (*required*), pero existen más posibilidades.

Parámetro	Tipo	Por defecto	Descripción
`indexed`	`boolean`	`true`	Los objetos están o pueden estar indexados por ese campo. Si no están indexados, el almacenamiento es más rápido, aunque no se puede hacer una *query* sobre datos en ellos.
`repeated`	`boolean`	`false`	El campo es en realidad una lista de valores.
`required`	`boolean`	`true`	El campo es requerido, debe proporcionarse un valor al crear el objeto.
`choices`	`list`	`None`	Lista de posibles valores para el campo, siempre del tipo del campo.
`validator`	`lambda`	`None`	Una función validadora para el campo. Debe retornar el nuevo valor, posiblemente modificado. Puede retornar **None** si la modificación no se admite. Se invoca con el objeto y el valor a guardar.

4.2 Consultas a la base de datos (Queries)

Se puede realizar una consulta a la base de datos mediante el método *query()*, como se ha visto en el ejemplo anterior. Normalmente, el método conlleva una condición sobre un campo que los objetos deben cumplir. Además, este método retorna un objeto que no es una lista, sino un objeto de la clase **ndb.Query**.

De esta clase, los métodos más importantes son *count()*, que devuelve los elementos recuperados, *filter()*, que permite establecer un filtro o filtros adicionales sobre los elementos a recuperar, y *order()*, que los recupera en un orden determinado.

En cualquier caso, el método *query()* admite varios parámetros:

- *kind = "Persona"*. El nombre de la clase de la entidad a recuperar.
- *keys_only = True*: Solo recupera las claves de los objetos, no los objetos en sí.
- *projection = ["nombre", "edad"]*: Solo recupera los campos especificados.
- *offset = 10*: Se salta los primeros 10 resultados.
- *limit = 10*: Establece un límite de 10 resultados.

En el momento de crear la query, ya se pueden establecer además, varios filtros.

```
personas_mayores_edad = Persona.query(Persona.edad >= 18)
personas_edad_trabajar = Persona.query(Persona.edad >= 18, Persona.edad < 65)
```

También se puede emplear el método filter() en lugar de especificar una lista de filtros.

```
personas = Persona.query()
personas_no_edad_trabajar = Persona.query(
                                        ndb.OR(
                                              Persona.edad < 18,
                                              Persona.edad >= 65))
personas_edad_trabajar = personas_mayores_edad.query(Persona.edad < 65)
```

Es posible buscar por valores que puedan aparecer en un campo. Por ejemplo:

```
personas = Persona.query(Persona.edad.IN([18, 19, 20]))
```

O combinar las búsquedas mediante relaciones OR, en lugar de solo AND:

```
personas_no_edad_trabajar = Persona.query(
                                        ndb.OR(
                                              Persona.edad < 18,
                                              Persona.edad >= 65))
```

Un método similar a *ndb.OR()*, *ndb.AND()* existe de manera que se puedan combinar las búsquedas sin límite.

El método *order()* permite establecer un orden. Cuando el campo por el que se ordena, se precede de un signo menos ('-'), entonces la ordenación es inversa.

```
personas_mayores_edad = Persona.query(Persona.edad >= 18).order(-Persona.edad)
```

Se pueden establecer ordenaciones múltiples:

```
personas_ordenadas = Persona.query().order(-Persona.edad).order(Persona.nombre)
```

Las *queries* devuelven objetos que pueden ser fácilmente recorridos mediante un bucle *for*.

```
personas_ordenadas = Persona.query().order(-Persona.edad).order(Persona.nombre)

for persona in personas_ordenadas:
      print(persona.nombre)
```

El objeto **Query** devcuelto se encarga, de manera *lazy*, es decir, solo recupera los objetos que realmente necesita, mediante iteradores inherentes al bucle *for*. Si se desea obtener todos los objetos que son representados la *query*, entonces se puede utilizar el método *fetch()*, que devuelve una lista, y opcionalmente permite especificar el máximo número de elementos a recuperar.

```
personas_ordenadas = Persona.query().order(-Persona.edad).order(Persona.nombre)

for persona in personas_ordenadas.fetch():
      print(persona.nombre)
```

El método *fech()* puede ser peligroso si el volumen de datos a recuperar es muy elevado. Si esto se prevee, es posible utilizar *fetch_async()*. Este método devuelve un objeto **Future** en lugar del valor en sí. El método **Future**.*done()* devuelve verdadero o falso según el resultado ya esté listo o no. **Future**.*get_result()* devuelve el resultado, bloqueándose y esperando si es necesario.

5 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 del Datastore de GAE (accedido en 04/2016): https://developers.google.com/appengine/docs/python/gettingstartedpython27/usingdatastore https://developers.google.com/appengine/docs/python/ndb/properties
- *Queries* en el Datastore de GAE (accedido en 04/2016): https://cloud.google.com/appengine/docs/python/ndb/queryclass https://cloud.google.com/appengine/docs/python/ndb/queries
- Documentación sobre manejo de imágenes en GAE (accedido en 04/2016): https://cloud.google.com/appengine/docs/python/images/usingimages
- Referencia de la clase **Image** (accedido en 04/2016): https://cloud.google.com/appengine/docs/python/refdocs/google.appengine.api.images#google.appengine.api.images.Image
- Documentación sobre *blobstore*: https://cloud.google.com/appengine/docs/python/blobstore/

6 Ejercicios

1. Crea una aplicación que permita guardar un directorio telefónico que registre nombre, e.mail, teléfono, y la foto del usuario. Se podrán añadir, modificar y eliminar usuarios.
2. *Recuerda el siguiente ejercicio, y añade la funcionalidad de guardar las facturas:* 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. No puede haber campos vacíos en la factura. La fecha de la factura será la fecha del momento de creación de la misma.
3. Escribe una aplicación sobre peritajes. 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.