G54CCS Labs, Exercise 7

This exercise demonstrates making use of a third-party service, in this case the Twitter search service. Specifically we will examine:

- Maintaining synchronization state;
- Encoding data for the web;
- Compliant fetching data from a third-party service; and
- GAE Cron jobs and Task queues.

This exercise builds particularly on the storage exercise (#4), and to some extent on the URL parameter exercise (#3). It is quite a bit more complex than those you have seen so far, so do not be surprised if you need to work through it fairly slowly and carefully.

Background

JSON Encoding

We will encode the data we fetch from Twitter as *JSON*. This is a relatively simple textual encoding of potentially binary data. This will all be handled by a Python library that comes with GAE — for more information see `http://json.org/`.

Extra Python Syntax

We will use two new pieces of Python syntax. The first is an abbreviation. Instead of writing

```
if condition:
    result = value
else:
    result = othervalue
```

we can write

```
result = value if condition else othervalue
```

The second is also an abbreviation, albeit a little more complex: called a *list comprehension*, it is a compact way of constructing one list from another. As some of the data we get back from Twitter arrives in lists, this will be quite convenient. Thus, instead of

```
result = []
for item in list:
    if condition: result.append(item)
```

we can write

```
result = [ item for item in list if condition ]
```

Synchronization State

Since things can go wrong when invoking a third-party service, for a whole range of reasons that are nothing to do with us (e.g., network errors), and since a user may try to fetch data from us while we're still fetching data from Twitter, we need to record the *synchronization state* of our program. We will use three states for this:

- *unsynchronized*, we are yet to start fetching data from Twitter;
- *inprogress*, we are in the middle of fetching data from Twitter; and
- *synchronized*, we have fetched all the data that Twitter offered us at this time.

Structure

The overall structure of the program is as usual: `urls.py` contains the mappings from URL to handler class; `views.py` contains the handler classes themselves; `models.py` contains the mapping from the application data model to the GAE storage service.

In outline, the app will invoke the Twitter search API URL to fetch tweets that contain a particular hashtag (`#cloud`). Twitter limit the number of responses that may be returned to a single request, instead returning a `next_page` field with the results. Thus we must repeatedly invoke the `next_page` URL we are given until there are no more results. As GAE limits the length of time you can spend processing a single invocation of a URL (typically to 30s), each repeated invocation of the `next_page` URL is via the *Taskqueue*. We schedule periodic fetching of new matching tweets using the GAE *Cron* facility: this causes GAE to periodically invoke a specified URL — in our case, every 2 hours. The Cron configuration is stored in a new file, `cron.yaml`.

cron.yaml

This is quite simple: each stanza contains an (optional) human readable *description*, the *URL* to be invoked, and the *schedule* for the URL to be invoked:

```
cron:
- description: twitter poller
  url: /cron
  schedule: every 2 hours
```

urls.py

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

import views

urls = [
    (r'/$', views.Root),
    (r'/cron/?$', views.Cron),
    (r'/sync(?:/(?P<cmd>start|stop))?/?$', views.Sync),
    (r'/tweets/?$', views.Tweets),
    ]

application = webapp.WSGIApplication(urls, debug=True)
def main(): run_wsgi_app(application)
if __name__ == "__main__": main()
```

This should now be fairly familiar in structure. We have four handler classes (`views.Root`, `views.Cron`, `views.Sync`, `views.Tweets`), and the usual application boilerplate. The `/sync` URL has a second element, `/start` or `/stop`, indicating whether synchronization should start or stop.

models.py

We begin with some relatively uninteresting boilerplate — you should be able to work out what all of this does now!

```
import sys, logging, datetime, time
log = logging.info
err = logging.exception

from django.utils import simplejson as json
from google.appengine.ext import db

def datetime_as_float(dt):
    '''Convert a datetime.datetime into a microsecond-precision float.'''
    return time.mktime(dt.timetuple())+(dt.microsecond/1e6)
```

Next we have a class we use to enumerate possible states of our program: unsynchronized, in progress, synchronized:

```
class SYNC_STATUS:
    unsynchronized = 'UNSYNCHRONIZED'
    inprogress = 'INPROGRESS'
    synchronized = 'SYNCHRONIZED'
```

Then we have the class that actually records which state we're in and, if synchronized, when we last became so. The class also contains two helper methods to encode the state itself: as a Python dictionary (`todict`) and as a JSON string (`tojson`):

```
class SyncStatus(db.Model):
    status = db.StringProperty(
        default=SYNC_STATUS.unsynchronized, required=True)
    last_sync = db.DateTimeProperty()

    def todict(self):
        last_sync = datetime_as_float(self.last_sync) if self.last_sync else None
        return { 'status': self.status,
                 'last_sync': last_sync,
                 }

    def tojson(self):
        return json.dumps(self.todict(), indent=2)

    def put(self):
        if self.status == SYNC_STATUS.synchronized:
            self.last_sync = datetime.datetime.now()
        super(SyncStatus, self).put()
```

Finally we have the class we use to represent an individual tweet, both in `raw` form and after being JSON encoded (as text):

```
class Tweet(db.Model):
    raw = db.TextProperty(required=True)
    txt = db.TextProperty(required=True)
```

views.py

```
import logging, urllib, urlparse
log = logging.info

from google.appengine.ext import webapp
from google.appengine.api import urlfetch, users
from google.appengine.api.labs import taskqueue
from django.utils import simplejson as json

import models

COUNT = 80
HASHTAG = '#cloud'

class Root(webapp.RequestHandler):
    def get(self):
        tweets = [
            json.loads(t.raw) for t in models.Tweet.all().order('__key__') ]

        self.response.headers['Content-Type'] = 'application/json'
        self.response.out.write(json.dumps(tweets))

class Cron(webapp.RequestHandler):
    def get(self):
        taskqueue.add(url="/sync/start", method="POST")

class Sync(webapp.RequestHandler):
    def get(self, cmd):
        ss = models.SyncStatus.get_by_key_name("twitter-status")
        self.response.headers['Content-Type'] = 'application/json'
        self.response.out.write(ss.tojson())

    def post(self, cmd):
        if cmd == "start":
            s = models.SyncStatus.get_or_insert("twitter-status")
            if s.status != models.SYNC_STATUS.inprogress:
                taskqueue.add(url="/tweets/", method="GET")
                s.status = models.SYNC_STATUS.inprogress

            s.put()

        elif cmd == "stop":
            s = models.SyncStatus.get()
            if s.status == models.SYNC_STATUS.inprogress:
                s.status = models.SYNC_STATUS.unsynchronized
                s.put()

        self.response.headers['Content-Type'] = 'application/json'
        self.response.out.write(s.tojson())

class Tweets(webapp.RequestHandler):
    def get(self):
        log("request:%s" % (self.request,))
        ps = { 'q': HASHTAG, 'rpp': COUNT, }
        max_id = self.request.GET.get("max_id")
        if max_id: ps['max_id'] = max_id

        page = self.request.GET.get("page")
        if page: ps['page'] = page

        url = "http://search.twitter.com/search.json?%s" % (
            urllib.urlencode(ps),)
        log("url:%s" % url)
        res = urlfetch.fetch(url)
        log("res:%s\nhdr:%s" % (res.content, res.headers))
        js = json.loads(res.content)

        if 'error' in js: ## retry after specified time
            retry = int(res.headers.get('retry-after', '300'))
            rurl = "%s?%s" % (self.request.path, self.request.query_string)
            log("retry: url:%s countdown:%d" % (rurl, retry+2,))
            taskqueue.add(url=rurl, countdown=retry+2, method="GET")

        else:
            if 'results' in js:
                for tw in js['results']:
                    t = models.Tweet.get_or_insert(
                        tw['id_str'], raw=json.dumps(tw), txt=tw['text'])
                    t.put()

            if 'next_page' in js:
                next_page = "%s%s" % (self.request.path_url, js['next_page'],)
                urlo = urlparse.urlsplit(next_page)
                next_page_rel = urlparse.urlunsplit(
                    ("", "", urlo.path, urlo.query, urlo.fragment))
                taskqueue.add(url=next_page_rel, method="GET")

            else:
                s = models.SyncStatus.get_by_key_name("twitter-status")
                s.status = models.SYNC_STATUS.synchronized
                s.put()
```

```
self.response.headers['Content-Type'] = 'application/json'
self.response.out.write(json.dumps(js, indent=2))
```