G54CCS Labs, Exercise 4

In the last exercise we extended our code to be more structured, and to handle incoming parameters from the user. In this exercise we will introduce three new concepts:

- Handling HTTP `POST`, as well as `GET`;
- Use of *page templates* to create more complete websites; and
- Management of state using the GAE storage backend.

Structure

The overall structure of the program is the same as before: `urls.py` contains the mappings from URL to handler class; `views.py` contains the handler classes themselves. GAE provides two basic storage backends: an unstructured *binary blob store*, and a structured *bigtable store*. We will not use the former here — it is typically for storing large, unstructured items of data such as media files, pictures, documents, and so on.

The bigtable store is a little like a SQL database — it stores *tables* of data, where each *column* has a particular type (integer, string, etc), and each *row* is a particular *record*. You can express constraints on columns to ensure that, for example, all rows contain a value in a given column or that a row's value in a particular column is always updated every time the row is stored. It does have some restrictions over a full SQL database however — most notably that the query syntax is a little more restricted, and the way that data can be indexed is much more restrictive. To compensate, you do not need to worry about backing up your database, or about scaling it up in size (although if your application goes live then once you get past the free limit, you would need to start paying).

`urls.py`

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

import views

urls = [
    (r'/value', views.Value),
    (r'/incr', views.Incr),
    ]

application = webapp.WSGIApplication(urls, debug=True)
```

```
def main(): run_wsgi_app(application)
if __name__ == "__main__": main()
```

This should now be fairly familiar in structure. As before, it simply defines two urls, `/value` and `/incr` with their associated handlers.

Before we proceed to `views.py` where the two handler classes, `Value` and `Incr`, are defined we will first look at `models.py`.

models.py

```
from google.appengine.ext import db

class Counter(db.Model):
    ctime = db.DateTimeProperty(auto_now_add=True)
    atime = db.DateTimeProperty(auto_now=True)
    value = db.IntegerProperty(default=0, required=True)
```

The first line is boilerplate, importing the relevant GAE library.

The remainder is the interesting part. This creates a class, `Counter`, which is used to map data objects in Python to a table in GAE storage. (The technical term for this is an *object relational mapping*, or ORM.) This file defines only one such mapping, `Counter`, which we will use to store a simple integer counter.

The counter in question is perhaps over-engineered, containing three separate fields which will each map to a different column in the corresponding storage table. Taking each in turn,

- `ctime` contains a timestamp indicating when the row was first *created* ("creation time").
- `atime` contains a different timestamp that indicates the last time the row was *accessed* ("access time").
- `value` contains the actual value of the counter, stored as an integer.

The type of each field, and so the corresponding column, is given as a *property*. In this case either `DateTimeProperty` for the timestamps, or an `IntegerProperty` for the counter value.

Each of these different properties makes different constraints available for specific instances as parameters to the property. In the case of this class we use the following:

- `ctime` sets the `auto_now_add` property to `True`. This ensures that when an instance of the `Counter` class is first stored ("added") by GAE, the value of this field will be set to the time that happens.

- `atime` uses a similar property, `auto_now`, but this sets the value of the field to the current time *every* time the instance of `Counter` is stored (not just the first).
- `value` uses two properties: `default` (unsurprisingly) sets the default value when no value is specified in the program; and `required` specifies whether the field is allowed to be left unspecified; setting it to `True` as in this case means that this field *cannot* be left blank.

The net result of this is that we have defined a table structure within which we can store values of the type `Counter`. GAE handles the rest of the storage details for us: which disk in which machine, how many copies should be stored, ensuring backups are on separate disks, indexing, providing a query infrastructure to retrieve values, and so on.

The remainder of the code is, as usual, in `views.py`.

views.py

```
import logging, time, os.path

from google.appengine.ext import webapp
from google.appengine.ext.webapp import template

import models
```

The standard `views.py` boilerplate, with three additions: we will need a function from the standard `os.path` module; we need access to the `templates` library module; and we also need access to the storage model we defined in `models.py`.

Next we define a simple helper function which constructs a string representation of the current time and date:

```
def now():
    return time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime())
```

Followed by another that will prepend the directory containing our HTML page template:

```
def t(p):
    return os.path.join("templates", p)
```

Now we come to the two handler classes. First, `Value`:

```
class Value(webapp.RequestHandler):
    def get(self):
        counter = models.Counter.all().get()
```

As usual this class defines a single method `get()` which will be invoked when the user's browser access the relevant URL with an HTTP `GET`. It continues by retrieving the counter using the `all()` method to specify that we want all suitable values, and *chaining* the `get()` method to actually issue the request to the GAE storage system. The result is a variable, `counter`, which contains the first matching value if one exists, or `None` otherwise.

We continue:

```
        context = { 'now': now(),
                    'counter': counter,
                    }
        if counter: counter.put()

        self.response.out.write(template.render(t("page.html"), context))
```

This now constructs a dictionary, `context` which contains the current time, and the value recovered from the storage backend (if any). If there is already a value in the storage backend (i.e., `counter` is *not* `None`), then we store it back. This should cause the `atime` field to be updated. Finally, we invoke the template system to construct the page we wish to return.

The template system works as follows: you *render* a predefined HTML template by passing in a dictionary which contains the values you want to put into your page. In this case our template is stored in a subdirectory of your application, `templates`, and we have only one template, `page.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Counting Is Fun!</title>
    <meta charset='utf-8'>
  </head>

  <body>
    <p>Time now is {{ now }}.</p>
    {% if counter %}
    <p>The latest counter value is {{ counter.value }}.</p>
    <p>Previous access was at {{ counter.atime }}.</p>
    <p>It was created at {{ counter.ctime }}.</p>
    {% endif %}
```

```
    <div style="float:left">
      <form action="/incr" method="post" accept-charset="utf-8">
        <input id="incr_button" type="submit" value="Increment"/>
      </form>
    </div>

    <div style="float:left">
      <form action="/value" method="get" accept-charset="utf-8">
        <input id="read_button" type="submit" value="Read"/>
      </form>
    </div>

  </body>
</html>
```

Do not worry too much about the details of this — for those who are interested, more information can be found by reading about HTML and CSS, particularly HTML5. In short, it begins with some boilerplate which specifies the page title; followed by three main blocks. The first is the main page contents which contains some text, e.g., `Time now is ...`, and the values of some variables (`now`, `counter.value`, etc) predicated on whether `counter` exists in the dictionary we gave to the `render` function (`{% if counter %} ... {% else %} ...`.

The next two blocks declare two buttons for the page which will invoke the relevant URLs (`/incr` and `/value`), before we finally finish the page with two more lines of boilerplate.

Finally, we have the other handler class, `Incr`:

```
class Incr(webapp.RequestHandler):
    def post(self):
        counter = models.Counter.all().get()
```

This starts very similarly to the `Value` class, with one important difference: by defining a `post()` method, this class will respond to `POST`s to `/incr` while the `Value` class responds to `GET`s to `/value`.

It continues:

```
        if not counter: counter = models.Counter()
        else:
            counter.value += 1

        context = { 'now': now(),
                    'counter': counter,
                    }
```

```
        counter.put()

        self.response.out.write(template.render(t("page.html"), context))
```

Here we *create* an instance of `Counter` if none exists; if one does already exist, we simply increment its value by 1. As before, we then create the dictionary to use for rendering the template, we store the counter (whether the new one, or an old one with an incremented value), and then finally, we render and return the template page.

Exercises

Ex.1. View the admin console for your running application at `http://localhost:8080/_ah/admin`. How many rows are in your table? What values do they contain? Why?

Ex.2. Lookup the documentation for the Python `time` module, and the `strftime` function in particular. Modify the `now()` function to record the day of the week in addition to the year/month/day/hours/minutes/seconds.

Ex.3. The URL `/value` is not very descriptive of invoking it does. Change it from `/value` to `/read`, and update the button label.

Ex.4. Extend this application to allow the user to increment by a value they specify as a parameter, using one of the methods from the previous exercise.