G54CCS Labs, Exercise 3

Last week's lab extended the first exercise to generate a more dynamic web page. In this exercise you will develop a new application that performs some simple arithmetic operations using data provided by the user via the browser. As your programmes are becoming more complex, you will also see one way to structure them to help you better track their operation.

Structure

Commonly, GAE applications split into three parts, given in three separate files:

- `urls.py` containing the mapping between the application's URLs and the classes that handle them;
- `views.py` containing the classes with the code that actually handles the URLS; and
- `models.py` containing code connected with storing data in App Engine — we do not need this now, but will investigate it in the next exercise.

Splitting the code up like this makes it easier to track your code as the application becomes more complex. It is, of course, possible to further subdivide your code among more files if it continues to grow.

The `app.yaml` file is also slightly changed in this case, as we no longer have an `app.py` file:

```
application: your-appname-here
version: 1
runtime: python
api_version: 1

builtins:
- datastore_admin: on
- appstats: on

handlers:
- url: /static
  static_dir: static

- url: /favicon.ico
  static_files: static/favicon.ico
  upload: static/favicon.ico
```

```
- url: /robots.txt
  static_files: static/robots.txt
  upload: static/robots.txt

- url: /.*
  script: urls.py
```

urls.py

This begins with the usual boilerplate:

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

Followed by:

```
import views

urls = [
    (r'/calc-parameters', views.CalcParameters),
    (r'/calc-url/(?P<op>add|subtract)/(?P<x>\d+)/(?P<y>\d+)/?', views.CalcUrl),
    ]
```

This first imports the contents of `views.py` to get access to the classes which will handle the URLs specified here.

It then specifies a list of pairs, or *2-tuples*, of (URL, handler-class). The first URL, `/calc-parameters` is handled by the `CalcParameters` class, and the second more complex URL, `/calc-url/...`, is handled by the `CalcUrl` class.

For the first URL, parameters are passed as HTTP URL parameters. These are specified by the user in the URL bar in the browser. The set of parameters is separated from the constant part of the URL by a `?`. Each separate parameter is separated from the others by a `&`. For example, `http://localhost:8080/calc-parameters?x=1&y=10&op=subtract`

...invoked the URL `http://localhost:8080/calc-parameters` with three parameters, `x=1`, `y=2`, and `op=subtract`.

Each URL is actually specified as a *regular expression* — these are a common way to specify string *patterns* rather than simple constant strings. In the case of the second URL above, we specify a constant part (`/calc-url/`) followed by three variable parts (`(?P<...>...)`) and finally an optional trailing slash (`/?`).

The variable parts are used to pass parameters, `(?P<op>...)` will pass a parameter labelled `op` to the handler; `(?P<x>...)` will pass a parameter labelled `x`,

and so on. In each case, the `...` is another regular expression that determines how much of the URL will be passed in as the labelled parameter. In the first case, the `...` is replaced by `add|subtract` which means the parameter `op` will have the value `add` or `subtract` depending which appears in the URL. If none appears, the parameter `op` will have a special value called `None` in Python (often called `null` in other programming languages).

In the second and third cases, the `...` is replaced by`\d+`. This can be broken down as `\d` meaning a decimal digit (i.e., 0–9) and `+` meaning "one or more of the previous character". Altogether then, both these mean that two variables `x` and `y` will be passed in, both containing a non-zero string of decimal digits.

The file then finishes in the usual manner, with the lines of code that actually run your application in response to an incoming request:

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

Note that this is because the `app.yaml` file points all incoming URLs to this file (`urls.py`) for handling.

views.py

This is now the file that contains the code which is used to actually deal with the incoming request. It begins in the usual fashion, although only *one* GAE import is required now since the other is now in `urls.py`:

```
from google.appengine.ext import webapp

import logging

HTML = """\
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>A Basic Calculator</title>
    <meta charset='utf-8'>
  </head>

  <body>
    %s
  </body>
</html>"""
```

We then have the code for the two handler classes referred to in `urls.py`. First, the class that handles invocation with parameters arriving as URL parameters, separated by `?...&`:

```
class CalcParameters(webapp.RequestHandler):
    def get(self):
        logging.info("request: %s" % (self.request,))

        x = int(self.request.get('x'))
        y = int(self.request.get('y'))
        op = self.request.get("op")

        logging.info("x:%d y:%d op:%s" % (x, y, op,))

        if op.lower() == "add":
            r = "PARAMETERS: %d + %d = %d" % (x, y, x+y)
        elif op.lower() == "subtract":
            r = "PARAMETERS: %d - %d = %d" % (x, y, x-y)
        else:
            r = "PARAMETERS: unknown operation %s" % (op,)

        self.response.out.write(HTML % r)
```

You should be able to follow most of this code with reference to the previous exercise. There are four new uses of Python:

- We must extract the parameters from the incoming request, referred to as `self.request`; this is done using its builtin `get()` method, which ensures that we get either the value given by the user, or `None` if the user gave no value.
- We want our parameters to be integers but they are passed in as strings, so we use the builtin`int()` function to convert them from strings to integers.
- We want to allow the user to specify the operation using any case (lower or upper), so we take what they give us, and convert it to all lower case by invoking the string's builting method `lower()`.
- We then decide which arithmetic operation is requested by testing with an `if...elif...else` clause. This tests the condition after the `if` or `elif` ("else-if") and, if true, executes the following block of code. If false, the following block is skipped and the next condition is evaluated, or if the `else` is reached, that block of code is executed anyway. In this case if the `op` parameter, once made lower case, is `add` then we add the `x` and `y` parameters together and construct the string we wish to embed in the page. Similarly in the case where `op` is `subtract`.

Using this method, you can easily handle parameters passed in by the user, in any order. The drawback of this approach is that, because the parameters are not embedded in the URL, search engines will not be able to index the resulting page — for some applications, this is important.

An alternative approach is to embed the parameters directly in the URL, as in the second class:

```
class CalcUrl(webapp.RequestHandler):
    def get(self, op, x, y):
        logging.info("request: %s" % (self.request,))

        x = int(x)
        y = int(y)

        logging.info("x:%d y:%d op:%s" % (x, y, op,))

        if op.lower() == "add":
            r = "URL: %d + %d = %d" % (x, y, x+y)
        elif op.lower() == "subtract":
            r = "URL: %d - %d = %d" % (x, y, x-y)
        else:
            r = "URL: unknown operation %s" % (op,)

        self.response.out.write(HTML % r)
```

This code is identical to the `CalcParameters` code except that the parameters are now **not** extracted from the request. Instead, the GAE libraries have already used the URL regular expressions provided to do this, and so the parameters are explicitly passed in as strings to the GET handler, `get()`.

Ex.1. Change the URL specifications to enable <http://localhost:8080/calc-parameters/ to work.

Ex.2. Add, to both methods, the ability to multiply and divide. Recalling the use of `float()` in the previous exercise, pay particular attention to the type of the result when dividing.

Ex.3. Extend both methods to support an optional third parameter with corresponding second operation. That is, to allow the user to compute things like `1+2-3` or `4+5*10`. Think about how to make the usual rules of arithmetic association works, i.e., how to ensure `4+5*10` is parsed as `4 + (5*10)` **not** `(4+5) * 10`.