web.py Tutorial

Tom Kelliher, CS 317

1 Acknowledgment

This tutorial is the tutorial from the web.py web site, with a few revisions for our local environment.

2 Starting

So you know Python and want to make a website. web.py provides the code to make that easy. Let's get started.

3 URL Handling

The most important part of any website is its URL structure. Your URLs aren't just the thing that your visitors see and email to their friends, they also provide a mental model of how your website works. On popular sites like del.icio.us, the URLs are even part of the user interface. web.py makes it easy to make great URLs.

Login to phoenix and open a shell window. In the following, anything from a `#` character on is a comment. You don't need to type comments! Run the following commands:

```
cd   # Make sure you're in your home directory.

umask   # This should print 022, 0022, or a similar octal number.

# The following two commands will ensure that only the web server
# has access to your project files.  This is a good thing, because your
# databases's password will be stored in here in cleartext.

chmod go= public_html
setfacl -m u:apache:rx public_html

# IMPORTANT:
# wsgi will be the root directory of your web application.
# Save your code.py into this directory.  Create your templates
# directory in this directory.

mkdir public_html/wsgi
```

To get started with your web.py application, open up a new text file (let's call it code.py) and type:

```
import web
```

This imports the web.py module.

Now we need to tell web.py our URL structure. Let's start out with something simple:

```
urls = (
  '/', 'index'
)
```

The first part is a regular expressions that matches a URL, like `/`, `/help/faq`, `/item/(\d+)`, etc. (i.e. `\d+` would match a sequence of digits). The parentheses say to capture that piece of the matched data for use later on. The second part is the name of a class to send the request to, like `index`, `view`, `welcomes.hello` (which gets the `hello` class of the `welcomes` module), or `get_\1`. `\1` is replaced by the first capture of your regular expression; any remaining captures get passed to your function.

This line says we want the URL `/` (i.e. the front page) to be handled by the class named `index`.

4 GET and POST: The Difference

Now we need to write the index class. While most people don't notice it just browsing around, your browser uses a language known as HTTP for communicating with the World Wide Web. The details aren't important, but the basic idea is that Web visitors ask web servers to perform certain functions (like GET or POST) on URLs (like `/` or `/foo?f=1`).

GET is the one we're all familiar with, the one used to request the text of a web page. When you type harvard.edu into your web browser, it literally asks the Harvard web server to GET /. The second-most famous, POST, is often used when submitting certain kinds of forms, like a request to purchase something. You use POST whenever the act of submitting a request does something (like charge your credit card and process an order). This is key, because GET URLs can be passed around and indexed by search engines, which you definitely want for most of your pages but definitely don't want for things like processing orders (imagine if Google tried to buy everything on your site!).

In our web.py code, we make the distinction between the two clear:

```
class index:
    def GET(self):
        return "Hello, world!"
```

This GET function will now get called by web.py anytime someone makes a GET request for /.

Now we need to create an application specifying the urls and a way to tell web.py to start serving web pages:

```
if __name__ == "__main__":
    app = web.application(urls, globals())
    app.run()
```

First we tell web.py to create an application with the URLs we listed above, looking up the classes in the global namespace of this file. And finally we make sure that web.py serves the application we created above.

Now notice that although I've been talking a lot here, we only really have five or so lines of code. That's all you need to make a complete web.py application.

For easier access, here's how your code should look like:

```
import web

urls = (
    '/', 'index'
)

class index:
    def GET(self):
        return "Hello, world!"

if __name__ == "__main__":
    app = web.application(urls, globals())
    app.run()
```

5 Start the Server

If you go to your command line and type:

```
python code.py 4200
```

The running program will print:

```
http://0.0.0.0:4200/
```

You now have your web.py application running a real web server on your computer. Visit that URL using a browser running on the same machine as your python program and you should see "Hello, world!" (The 4200 is the port number. In-use port numbers must be unique and be greater than 1024.)

6 Templating

Writing HTML from inside Python can get cumbersome; it's much more fun to write Python from inside HTML. Luckily, web.py makes that pretty easy.

Let's make a new directory for our templates (we'll call it `templates`). Inside, make a new file whose name ends with html (we'll call it `index.html`). Now, inside, you can just write normal HTML:

```
<em>Hello</em>, world!
```

Or you can use web.py's templating language to add code to your HTML:

```
$def with (name)

$if name:
    I just wanted to say <em>hello</em> to $name.
$else:
    <em>Hello</em>, world!
```

As you can see, the templates look a lot like Python files except for the `def with` statement at the top (saying what the template gets called with) and the `$`s placed in front of any code. Currently, template.py requires the `$def` statement to be the first line of the file. Also, note that web.py automatically escapes any variables used here, so that if for some reason name is set to a value containing some HTML, it will get properly escaped and appear as plain text. If you want to turn this off, write `$:name` instead of `$name`.

Now go back to code.py. Under the first line, add:

```
render = web.template.render('/home/username/public_html/wsgi/templates/')
```

(Replace `username` with your username.) This tells web.py to look for templates in your templates directory. Then change `index.GET` to:

```
name = 'Bob'
return render.index(name)
```

('index' is the name of the template and 'name' is the argument passed to it)

Visit your site and it should say hello to Bob.

But let's say we want to let people enter their own name in. Replace the two lines we added above with:

```
i = web.input(name=None)
return render.index(i.name)
```

Visit `/` and it should say hello to the world. Visit `/?name=Joe` and it should say hello to Joe.

Of course, having that `?` in the URL is kind of ugly. Instead, change your URL line at the top to:

```
'/(.*)', 'index'
```

and change the definition of index.GET to:

```
def GET(self, name):
    return render.index(name)
```

and delete the line setting name. Now visit `/Joe` and it should say hello to Joe.

If you wish to learn more about web.py templates, visit the templetor page.

7 Forms

The form module of web.py allows the ability to generate html forms, get user input, and validate it before processing it or adding it to a database. If you want to learn more about using the module forms web.py, see the Documentation or direct link to Form Library

8 Databasing

First you need to create a database object.

```
db = web.database(dbn='postgres', user='username', pw='password', db='dbname')
```

(Replace `username` and `password` with the appropriate values. Replace `dbname` with your database name, which is just your username.)

That's all you need to do – web.py will automatically handle connecting and disconnecting from the database.

Using your database engines admin interface in a shell window, create a simple table in your database:

```
# Start the admin inteface.  Respond with your database account password.
# Initially, your password is the same as your username.
psql

# Change your password.  Replace 'kelliher' with your username and
# select a secure password.

alter user kelliher with password 'CurlyQ42';

# SQL keyword capitalization is optional.

CREATE TABLE todo (
  id serial primary key,
  title text,
  created timestamp default now(),
  done boolean default 'f'    );
```

And an initial row:

```
INSERT INTO todo (title) VALUES ('Learn web.py');

\d   # Print the objects in your database.

\d todo # Print the todo table's schema.

select * from todo;   # Print the contents of the todo table.

\q   # Exit psql
```

Return to editing code.py and change `index.GET` to the following, replacing the entire function:

```
def GET(self):
    todos = db.select('todo')
    return render.index(todos)
```

and change back the URL handler to take just `/` as in:

```
'/', 'index',
```

Edit and replace the entire contents of index.html so that it reads:

```
$def with (todos)
<ul>
$for todo in todos:
    <li id="t$todo.id">$todo.title</li>
</ul>
```

Visit your site again and you should see your one todo item: "Learn web.py". Congratulations! You've made a full application that reads from the database. Now let's let it write to the database as well.

At the end of index.html, add:

```
<form method="post" action="add">
<p><input type="text" name="title" /> <input type="submit" value="Add" /></p>
</form>
```

And change your URLs list to read:

```
'/', 'index',
'/add', 'add'
```

(You've got to be very careful about those commas. If you omit them, Python adds the strings together and sees '/index/addadd' instead of your list of URLs!)

Now add another class:

```
class add:
    def POST(self):
        i = web.input()
        n = db.insert('todo', title=i.title)
        raise web.seeother('/')
```

(Notice how we're using POST for this?)

web.input gives you access to any variables the user submitted through a form.

Note: In order to access data from multiple identically-named items, in a list format (e.g.: a series of check-boxes all with the attribute `name="name"`) use:

```
post_data=web.input(name=[])
```

db.insert inserts values into the database table todo and gives you back the ID of the new row. seeother redirects users to that URL.

Some quick additional notes: db.update works just like db.insert except instead of returning the ID it takes it (or a string WHERE clause) after the table name.

web.input, db.query, and other functions in web.py return "Storage objects", which are just like dictionaries except you can do `d.foo` in addition to `d['foo']`. This really cleans up some code.

You can find the full details on these and all the web.py functions in the documentation.

9 Developing

web.py also has a few tools to help us with debugging. When running with the built-in webserver, it starts the application in debug mode. In debug mode any changes to code and templates are automatically reloaded and error messages will have more helpful information.

The debug is not enabled when the application is run in a real webserver. If you want to disable the debug mode, you can do so by adding the following line before creating your application/templates.

```
web.config.debug = False
```

10 Running Your Application on Phoenix's Web Server

Change the end of code.py to read:

```
if __name__ == "__main__":
    app = web.application(urls, globals())
    app.run()
else:
    app = web.application(urls, globals(),autoreload=False)
    application = app.wsgifunc()
```

Now, using a browser running on any computer inside Goucher's network, visit the URL `http://phoenix.goucher.edu/~username/wsgi/code.py/` and you should see your web app running. (As before, replace `username` with your username.)

If you edit a template file and notice that the web server doesn't appear to be using the new version of the file, run Linux's `touch` command on your Python code file to tell the web server to read the new versions of all your application files:

```
touch code.py
```

11 What Next?

This ends the tutorial for now. Take a look at the cookbook and the code examples for lots more cool stuff you can do with web.py. Also don't forget about the api reference