CS 145 Introduction to Databases

Stanford University, Fall 2015

Getting Started with web.py and Jinja2

Before starting Part 3 of the AuctionBase project, your first task is to become familiar with the web.py framework and the Jinja2 templating engine. You'll be using web.py for request handling and database querying, while Jinja2 will be used for generating HTML responses. Before jumping in, we recommend at least skimming through the Web.py tutorial, the Web.py API documentation, and the Jinja2 Template Designer Documentation, to familiarize yourself with the tools you'll be using, particularly if you've never used web.py or Jinja2 before. If these still aren't helpful, you can find additional materials in the Web.py code samples and the Web.py cookbook.

Examining the Sample Code

The provided sample code should serve as a guide for the overall code structure, as well as for implementing key components of Part 3 of the project, such as querying the database, adding additional URLs, and correctly handling parameters from the user. You should feel free to build off this code to complete the assignment.

The first thing to examine is the `getTime` method in `sqlitedb.py`. This will give you an example of how to execute a simple command on a SQLite database in web.py and process the results. In `auctionbase.py`, the `GET` method for the `curr_time` class also shows how to call this method from your main application.

Next, take a look at `getItemById` method, also in `sqlitedb.py`. Here, you'll notice that the method takes in a variable, `item_id`, to specify which item to look for. We pass both the `query_string` and a Python dictionary that includes `item_id` to the method `query`, which will automatically take care of splicing in the value of `item_id` into the `query_string`. This works because the key in our dictionary, `'itemID'`, matches the variable `$itemID` in the `query_string`.

So, as you extend `sqlitedb.py` to support different types of queries, it is important that you follow this same template – otherwise, web.py will not be able to correctly generate the correct query statements for you! In particular, if you wish to support different input variables in your queries, you should always place these variables in a Python dictionary that corresponds to your query string.

> **Warning**: Do not directly concatenate or insert different variables into the query string using standard Python string operations! This will introduce a critical security loophole into your website!

You'll notice that, for every URL, there is a mapping from the URL to a given Python class in `auctionbase.py`. (This mapping is explicitly shown in the `urls` variable, which is located just above the `curr_time` class in the sample code.) Any time you add an additional URL to your web application, you need to provide a mapping in the `urls` variable from the URL to a Python class in `auctionbase.py`. Be sure to update the `urls` variable to include this mapping – otherwise, your application will not work.

You'll also notice that every Python class either has a `GET` method and/or a `POST` method included as part of the class definition. For a URL to be valid in web.py, you need to implement at least one of these methods for a given class. Inside your `GET/POST` method, you have access to the parameters passed with the URL to your application; a simple call to `web.input()` will provide for you a Python dictionary of those parameters. (See the `POST` method in `select_time` for an example.) Lastly, you must invoke `render_template` at the end of the method and return its value. (See the examples in `curr_time` and `select_time`.)

The `render_template` method takes in two arguments: the name of the template file to be rendered, and a dictionary of key-value pairs that will be passed to the template as variables. For example, the `GET` method in `curr_time` returns `render_template('curr_time.html', time = current_time)`. This means that, if a user navigates to `/cgi-bin/auctionbase.py/currtime`, the `curr_time.html` template will be rendered and returned to the browser. All of your template files will be located in the `templates/` directory; you should find four sample templates already provided for you: `add_bid.html`, `curr_time.html`, `select_time.html`, and `app_base.html`. Examine `curr_time.html` to familiarize yourself with these templates. Although these template files have `.html` file extensions, these are **not** HTML files. Instead, they are Jinja2 templates, which contain a mixture of HTML and Jinja2, a templating language that is patterned after Python. (Note: do not confuse the two – although Jinja2 looks like Python, it is **not** Python.)

Jinja2 gives you the ability to compose your HTML in a clean but dynamic fashion, using various control structures (such as `for` loops and `if` statements) as well as variables. In `curr_time.html`, we see the use of variables in the fifth line:

```
Current time is: {{ time }}
```

The double-braces on either side of `time` denote that Jinja2 will treat `time` as a Python variable that needs to be evaluated; the value of that variable will then be inserted into the generated HTML. So, for example, if the variable `time` was set to the value "9:00 PM", then Jinja2 would transform our line of code into "Current time is: 9:00 PM".

Where does the time variable come from? It comes from our invocation of `render_template` in `auctionbase.py`. We passed in `time = current_time` as the second argument, which means that the `time` variable was set to the value of `current_time`, the variable we created in `auctionbase.py`. By doing this, we effectively passed the current time in our database into our `curr_time.html` template, so that it can be displayed as HTML in the browser. (Note: you can pass in any number of variables to `render_template`; you simply need to delimit each key-value pair with a comma, e.g. `time = current_time, user = ''Joe'', number_of_users = 1`, etc.)

Jinja2 offers a great deal of additional functionality, most of which you will not need for this project. However, here are some features that are likely to come in handy:

- `for` loops and `if` statements: Pay close attention to the `for-else` example (the second-to-last example in the "For" section) as well as the table of special variables that are made available inside `for` loops.
- String concatenation: Jinja2 does not support string concatenation with the '+' character – instead, use the '~' operator. This operator will also convert non-string variables (such as `floats` or `ints`) into strings before concatenation.
- Variable definition: Checking to see whether or not a variable is defined in your template will almost certainly be useful when addressing various edge cases in your application.

Lastly, you will need to understand the concept of base and child templates in Jinja2. You can see two lines at the top of `curr_time.html`

```
{% extends "app_base.html" %}
{% block content %}
```

and one line at the bottom

```
{% endblock %}
```

that aren't directly related to the HTML output of the template. These three lines are necessary because they specify that `curr_time.html` is actually a child template of `app_base.html`, which is a base template. Essentially, `curr_time.html` is responsible for generating only part of the HTML response for the `/currtime` URL – the `app_base.html` template also generates part of the HTML response, too. If you look at the contents of `app_base.html`, you'll see that it's mostly a "skeleton" template – it contains boilerplate HTML, along with a simple navigation bar that appears at the top of the webpage.

Using `app_base.html` as our base template allows us to reuse it in powerful ways. For example, if we open `select_time.html`, you'll notice that it, too, contains the same three lines at the top and bottom – it also inherits from `app_base.html`. Now, if we navigate to the `/selecttime` URL from our browser, the same navigation bar should also appear at the top of the page.

This base-child template model allows us to modularize our templates so that we can reuse common components throughout our web application. While it's not required that you adhere to this design pattern, we do recommend that you continue using it as you add additional templates for any new URLs you include in your application. Specifically, you should always include

```
{% extends "app_base.html" %}
{% block content %}
```

at the very beginning of your template, as well as

```
{% endblock %}
```

at the very end. (For more information on template inheritance in Jinja2, see the documentation here.)

Performing transactions in web.py

Here is an example of how you might structure your code in `auctionbase.py` to use database transactions:

```
t = sqlitedb.transaction()
try:
    sqlitedb.query('[FIRST QUERY STATEMENT]')
    sqlitedb.query('[SECOND QUERY STATEMENT]')
except Exception as e:
    t.rollback()
    print str(e)
else:
    t.commit()
```

The first line is responsible for initiating the transaction. Then, we begin our queries on the database, but we do so within a `try/except` block. That way, if our query violates a constraint in the database, we can catch the resulting error thrown by web.py and handle it appropriately. Here, we call `t.rollback()` to abort the transaction, and then call `print str(e)` to print the error message generated by our SQLite database (which most likely explains which constraint we violated) to the console. If no errors occur, we enter into our `else` branch and commit our transaction.

Debugging web.py locally

If you start to come across errors in your application – and you undoubtedly will – it may be helpful to run your application locally, rather than through Stanford's Personal CGI service. You can do this by executing the following from your shell:

```
python auctionbase.py [OPTIONAL: port_number]
```

This command will start the web application and loop indefinitely, printing out a thorough debug log of the activities your application is performing, including all SQL queries it executes. Any additional print statements you add to your code will also appear here, enabling you to diagnose any problems that may arise.

There is, however, an important caveat: your application will only be accessible from the local machine. For example, if you have logged in via `ssh` to the `corn24` machine, then you will only be able to access the web application from a

browser that also runs on corn24 as well. More importantly, you will **not** be able to access the web application from your own browser. So, to access the application, you must open Firefox from the shell, by executing:

```
firefox &
```

(**Note**: X-forwarding must be enabled for this to work – otherwise, you will not be able to launch Firefox from the command-line. Since, you're running `python auctionbase.py` in your shell already, you may need to open a new window and `ssh` into the same `corn` machine.)

Once the Firefox browser window appears, you can then access the application by typing `localhost:8080/currtime` (or any other URL that's part of your application, e.g. `localhost:8080/selecttime`) into the browser's URL address bar. Notice that `auctionbase.py` is no longer part of the URL as it was with CGI – when testing the server on a local machine, `auctionbase.py` is **not** needed in the URL.

Now, whenever you visit a page via `localhost`, you'll get a steady stream of debugging information in your shell. Once you're done debugging, you can quit the local web application by typing `Ctrl-C`.

One last important point: as is indicated above, you can pass an optional `port_number` argument when you execute `python auctionbase.py`. This argument specifies which port the webserver will run on. (By default, the port is 8080.) This can be helpful because, occasionally, the default 8080 port is already occupied by another process. (Usually, it will be another CS145 student who's working on this project and is logged into the same `corn` machine as you!) If that's the case, you'll most likely receive a "`socket.error: No socket could be created`" error message; you can easily fix this by explicitly providing a port number when you execute the command, like so:

```
python auctionbase.py 8081
```

Debugging web.py via cgi-bin

While debugging your web application locally can be very helpful, you may also encounter problems that are only exposed when running your application on Stanford's Personal CGI service. These bugs won't necessarily show up when you run your application locally, though, so how can you debug these problems?

Fortunately, Stanford IT provides a nice way to debug your application while it's executing from your `cgi-bin` directory. Navigate to your `web.py` directory (inside of `cgi-bin`) and execute the following command:

```
touch .suexecd
```

This will create a hidden file called `.suexecd` that exists inside your `web.py` directory. (To see the file, you need to execute `ls -a` instead of `ls`.) Now, when you visit your AuctionBase website via the `cgi-bin` URL, you'll now be able to see server log information printed at the very top of the webpage. (Any print statements you've included in your application will also appear.) If there's a bug in your application and an exception is thrown, you'll also be able to see a stack trace for that exception printed at the very top of the page.

Once you've finished debugging, you can simply remove the `.suexecd` file (using `rm .suexecd`), and the server logs will no longer be printed. Your AuctionBase website will now be rendered as it was before.

Debugging web.py with print statements

We certainly encourage to use print statements to debug your web.py application. However, it's important to note that any print statements you include in your application **will interfere with the rendering of your webpages**. In particular, the web.py server will include your print statements as part of the HTML response it sends to the browser – this will often mean that a particular URL request that triggers a print statement (or set of print statements) **will not work**. (Oddly enough, it **will** work if you've executed `touch .suexecd` to debug your application, as was explained above. This is important to remember as well!)

Most importantly, be sure to remove all print statements from your web.py application before you submit! Please don't forget to do this!

Configuring Your Database and Views

Let's go ahead and make the first change to the starter code: configuring your SQLite database to work with your web application. In particular, let's modify the `sqlitedb.py` file so that it uses the correct filename of you SQLite database. For instance, if your SQLite database is named `auctions.db`, you should update the existing code at the top of the file to be:

```
db = web.database(dbn='sqlite', db='auctions.db')
```

Be careful: If `auctions.db` does not exist in your `cgi-bin` directory, web.py will automatically create a new empty SQLite database called `auctions.db` for you!

For a first sample of how a page works in web.py, visit the following URL in your browser:

```
http://www.stanford.edu/~[your SUNet ID]/cgi-bin/auctionbase.py/currtime
```

At this URL, you should see the current time as reflected in your `CurrentTime` table. If this doesn't work, you may need to edit `sqlitedb.py` so that it uses your `CurrentTime` table (named `Time` in the sample code) and the correct attribute within your `CurrentTime` table (named `currenttime` in the sample code). At this point, the `/cgi-bin/auctionbase.py/currtime` URL should be working! ☺