Lab 4 : Using a Database as a Web Backend

CMSC 362
Spring 2017
Due Friday, 3/31/2017 by 11:59pm

Introduction

Databases are especially useful as the backend to powerful web applications. Most popular web sites, from Amazon to Yahoo, use some kind of database powered interface. A database administrator should be familiar with the ways in which a database can interact with these web technologies. In this lab, you will build a web interface for your "rants" database using a particularly powerful set of tools: Flask (a python microframework), Jinja (a web templating library), HTML5 (the latest hypertext markup standard), JQuery (a Javascript library for manipulating HTML elements), and Psycopg2 (a Python database interface).

This is a very complex project with a lot of parts. I have given you a lot of time to work on it, but you should start early and ask lots of questions if you get stuck.

Some resources you may find helpful are:

The Flask documentation: http://flask.pocoo.org/docs/quickstart/
The Flask tutorial: http://flask.pocoo.org/docs/tutorial/
The Jinja documentation: http://jinja.pocoo.org/
The JQuery documentation: http://learn.jquery.com/about-jquery/
The psycopg documentation: http://initd.org/psycopg/docs/

If you have never programmed in Python before, here are a few tips:

1. Python is interpreted. To run a python program, just type:

```
python program_name.py
```

2. Whitespace matters in Python. Instead of using curly-braces, Python uses indentation to indicate which statements belong to the same block.

3. Even though a tab and four or five spaces may both line things up on your screen, Python treats them differently. My solution is to always use TAB to indent instead of spaces, though in vim, it may be easier to set the "expandtab" option which converts all presses of the tab key into a number of spaces. Whichever solution you choose, be sure to be consistent so that you don't get syntax errors.

Step 0. Setting Up

Make a directory named Lab4. In that directory, create two additional directories: one named "static" and the other named "templates". The Jinja templating interface will use these to create the web pages that will implement our project. Then cd to the "static" directory and type:

```
	wget http://code.jquery.com/jquery-2.1.0.min.js
```

This downloads the most recent production release of the JQuery library into your static folder. We will use it later.

Now cd back to your Lab4 folder and use vim to create a file named "server.py" there. Most of your code will go in this file.

Step 1. A simple Flask server

Flask is a web micro-framework. That means that it is a very simple (but powerful) programming interface for creating dynamic web pages. Flask acts as a special kind of web server that maps URLs to Python functions rather than to static pages. Anything your function returns will be turned into a web page and displayed in the user's browser.

In your “server.py” file, type the following:

```
from flask import Flask, g

app = Flask(__name__)
app.debug = True            #Comment this out once your code is working

app.config.from_object(__name__)
app.config.update( dict(
       DATABASE=”lastname”,
       SECRET_KEY=”rant_key”,
       USERNAME=”lastname”,
       PASSWORD=”lastname”
))

@app.route(“/”)
def hello_world():
       return “Hello World”

if __name__ == “__main__”:
       app.run()
```

There are **two** underscore characters on each side of “__name__” and “__main__”. *Replace lastname with your actual database credentials on torvalds and change SECRET_KEY from “rant_key” to some random string (it doesn't matter what as long as it is different for each group).*

This imports several important objects from the flask library: the Flask server and the “g” object.

The “g” objects allows us to set global state for the flask server that can be shared between threads. We will use it later in the project. We then create a Flask server named “app”, and set some configuration variables.

The key piece of code is the part that says:

```
@app.route(“/”)
def hello_world():
       return “Hello World”
```

The at sign (@) is the Python syntax for a “function decoration”. Basically, it gives Python extra information about the function that follows (in this case, “hello_world”). The app.route decoration maps the URL “/” to the hello_world function.

Finally, we run the web server by calling app.run().

Save and quit. Then test your code by typing:

```
python server.py
```

You should see a message that a Flask server is starting up on port 5000. The “/” route is the default URL, so if you open a web browser (firefox?) to http://localhost:5000 and you should see a “Hello World” message show up.

Step 2. Additional Routes

Add a new route that maps “/goodbye” to a function named “goodbye_world” that returns the string “Goodbye World”. You can do this by typing:

```
@app.route(“/goodbye”)
def goodbye_world():
       return “Goodbye World”
```

Verify that it works properly in your browser, then add a route from "/welcome" to a function named "welcome_back" that returns the string "Welcome Back".

Test your work by browsing to http://localhost:5000/goodbye and http://localhost:5000/welcome (you should see the appropriate message in your browser when you visit each site).

Step 3. Jinja Templates

We now have a functional web server, but it doesn't do a whole lot. We could modify our functions so that instead of simple strings they printed entire HTML pages, but a better solution is to use templates.

A template is like a web page, but contains special placeholders that we can replace with new data when the user accesses the page. We can obtain the data for those placeholders from variables in our program, from files stored on disk, or from our database.

One of the most powerful templating libraries available for Python is Jinja2. Jinja has a really simple syntax, but provides incredibly powerful features for generating dynamic content. For instance, you can use "for loops" in your web page to display several slightly different copies of a block of HTML code or use "if statements" to only display a message if a certain condition is true.

Jinja works with Flask, but also with several other web frameworks (including Python pylons). There are also several other templating libraries that flask can use (such as mako and genshi). However, Jinja and Flask are developed together and they are very easy to use together.

Probably the simplest use of templates is to insert the contents of a variable into part of a web page.

Let's create an example template that lets you print pass a variable from Python to the templating engine. In your templates folder, create a file named "example.html". Add the following code to the file:

```
<!DOCTYPE html>
<html lang=en>

<head>
<meta charset=utf-8>
<title>Example Page</title>
</head>

<body>
<p>Your name is: {{ name }}.</p>
</body>

</html>
```

This is a very primitive HTML 5 document which contains a simple header (which sets the title and character set) and a one paragraph body. The only really unusual thing about it is the "{{ name }}" in the middle of the paragraph. This is the Jinja syntax for embedding a variable into the web page. When we tell Flask to render this template, Jinja will replace "{{ name }}" with the contents of the "name" variable from your python code (which we will add in a moment).

Save your work and quit. Then open your server.py file again and add "render_template" to the list of imports at the top of the file.

Then add a route named "/example/<username>" and associate it with the following function:

```
def template_example(username):
       if username=='superman':
              username = 'Clark Kent'
       return render_template("example.html", name=username)
```

Notice that this function takes a parameter. Flask allows you to embed variables in your route by enclosing them in angle brackets. If the user browses to http://localhost:5000/example/bob Flask will call template_example and pass "bob" in for username. If they browse to http://localhost:5000/example/sarah instead, Flask will pass "sarah" in for the username.

To render the template, we simply call the render_template function and return the result. Notice that the first parameter is the filename of the template relative to the templates folder. Jinja and Flask automatically add the templates folder to the path, so we just need the filename. To set the Jinja variable "name" to a value, we simply pass it in as a keyword parameter. In this case, we set it to username.

Test your work.

Step 4. Connecting to PostgreSQL

Flask and Jinja can be used with a wide variety of database systems. They are designed to work best with a package called SQL-Alchemy that acts as an intermediary between the database and Python. Alchemy is very powerful and basically coverts the rows of your database into Python objects (this is called an "object-relational mapper" or ORM).

Alchemy is overkill for what we need, so instead we'll use a database connector called Psycopg. Psycopg is a postgres-specific database interface for Python. However, it implements a more general API for Python database programming called "DB-2". This means that the same function calls you use to communicate with psycopg can be used with interfaces for mysql, sqlite, and other database systems.

The first step in retrieving data from the database is to open a connection to the postgres server.

Since we may need to do this from several different functions, we will create a global function which checks whether a connection has already been established and, if not, sets one up.

At the top of your program add:

```
import psycopg2, psycopg2.extras
```

Then add these two functions immediately after the "app.config.update" function call:

```
def connect_db():
       “””Connect to postgresql”””

       db=psycopg2.connect(database=app.config['DATABASE'],
user=app.config['USERNAME'], password=app.config['PASSWORD'],
host=”torvalds.cs.longwood.edu”, cursor_factory=psycopg2.extras.RealDictCursor)

       return db

def get_db():
       “””Ensure that a database connection is open.”””

       if not hasattr(g, 'db'):
              g.db = connect_db()

       return g.db.cursor()
```

The connect_db function establishes a new connection to your database on torvalds. Notice how we used the app.config object you set up earlier. This makes it easy to change the username or other parameters without modifying your code in several places.

The get_db function checks the "global state object" to see whether we're already connected to the database. If so, it just returns a database cursor. If not, it first opens a new connection using the connect_db function.

Step 5. Closing the Database

We want to close our database connection whenever the user closes the web page. Since we can't rely on them to click a "logout" button first, we need some way to detect that. Fortunately, Flask makes this easy by providing a teardown decorator we can use to identify a function that should be called whenever the web connection disappears.

Type:

```
@app.teardown_appcontext
def close_db(error):
       """Closes the database at the end of a request"""
       if hasattr(g, 'db'):
              g.db.close()
```

Save your work and test that you can still access the routes from the previous problem.

Step 6. Advanced Templates

Before we can retrieve data from the database, we should create a template that will let us display it. This is where Jinja can really shine. Create a new file named "rants.html" in your templates folder and type the following into it:

```
<!DOCTYPE html>
<html lang=en>
<head>
<meta charset=utf-8>
<title>Rants</title>
</head>
<body>
<p>{% for rant in posts %}
   {{ rant.username }}: {{ rant.rant }}<br/>
{% endfor %}</p>
</body>
</html>
```

Notice that to identify a Jinja programming construct like the for loop, we use {% and %} instead of double-curly braces. Also notice that we can easily intersperse Jinja variables and HTML tags like <br/>.

Jinja for loops are really straightforward. The syntax is just "for <variable> in <list>". Jinja will basically copy and paste all the HTML between the {% for <blah> %} and the {% endfor %}, making one copy for each member of list. In each copy, it will replace "variable" with the corresponding value in the list. If the items of the list are complex Python structures, like lists or dictionaries, we can access their fields using array notation or "dot" notation. In this case, each "rant" is one database row, and we access the columns by simply saying something like "{{ rant.username }}".

Cross-site scripting is a form of injection attack that is a major concern for web developers. Jinja provides a really neat form of protection against this. If you add the string |safe after a variable name, Jinja will automatically escape any dangerous characters in that variable's expansion.

Change "rant.username" and "rant.rant" to "rant.username|safe" and "rant.rant|safe". Then save your template and quit.

Step 7. Querying the Database

Now we are ready to pull data from the database into a template. Let's modify the default route to provide a list of our rants. Replace your hello_world function with a function named "list_rants" that contains the following code:

```
@app.route(“/”)
def list_rants():
        db = get_db()
        db.execute('SELECT post_id, username, rant, time_posted FROM rants')
        rows = db.fetchall()

        return render_template(“rants.html”, posts=rows)
```

See how easy that is?

Step 8. Featuring Comments

Your turn. Modify the rants.html template so that inside the for loop for each post there is an unordered list. To do this, add a <ul> and </ul> tag pair. Inside that list, add a nested loop which uses a variable named “comment” to iterate through “rant.comments” (which we will create in a minute) and print a list item (inside <li> and </li> tags) of the format:

```
        username: comment_text
```

(To get the username, you will need to use {{ comment.username }}. I will let you figure out how to get the comment_text, but it's similar).

Then modify list_rants() to build the rant.comments list. The way I did this looked something like this:

```
        for rant in rows:
                db.execute( “SELECT username, comment_text FROM comments  WHERE 
comments.post_id = '%s'” % (rant[“post_id”]) )
                rant[“comments”] = db.fetchall()
```

Be sure to test your code.

Step 9. Web Forms

This web site would be a lot more useful if we could actually add data to the database, too. Let's start by adding a web form to rants.html that will allow us to create new posts. Add the following just above the </body> tag:

```
<fieldset>
<legend>New Post</legend>
<form method=”POST” action=”new_post”>
<p>Username:<input type=”text” name=”username”></input></p>
<p>Rant:<br/><textarea width=”40” name=”text”></textarea></p>
<input type=”submit”></input>
</form>
</fieldset>
```

Now we need to add a route to process the information that comes in when they click submit. To do that, first add request, redirect, url_for, and escape to the list of imports from flask (right after render_template). Then create a new route that maps /new_post to a function named process_new_post:

```
@app.route('/new_post', methods=['POST'])
def process_new_post():
        username = request.form['username']
        text = request.form['text']

        db = get_db()
        db.execute(“INSERT INTO rants(username, rant) VALUES ('%s', '%s')” % 
(username, text))
        g.db.commit()
        return redirect(url_for('list_rants'))
```

This function reads the username and text of the new post from the web form. Then it connects to the database and does and "INSERT INTO". Finally, instead of displaying a string or web template, it reloads our main page. The url_for function takes the name of a python function as a string and returns the route for it. The redirect function tells flask to call that function and print its result just as if the user had visited it.

You should now be able to add new posts to the page.

Just as we had to protect the web page from HTML injection attacks, we need to protect the database from SQL injection attacks. To do this, we use the "escape" function. Change the username and text lines to read:

```
        username = escape(request.form['username'])
        text = escape(request.form['text'])
```

Flask will automatically escape any characters that are meaningful to PostgreSQL.

Now modify your rants.html file so that following each post, there is a web form that allows you to add a comment on that post. Set the method of the form to POST and the action to "new_comment". In order to identify which rant the comment goes with, you will need to use a hidden tag:

```
<input type="hidden" name="post_id" value="{{ rant.post_id }}"></input>
```

Notice that you can use Jinja variables inside a tag attribute!

Other than the hidden tag, your form will be very similar to the one we used before for creating new rants.

To handle the new comment, add a function named "process_new_comment" with a route of "/new_comment" that will accept the input from one of your comment forms and insert the corresponding information into the database.

Step 10. Making it prettier

Right now the web page is pretty cluttered with forms. JQuery is a Javascript library that allows us to easily manipulate the elements of a web page. Let's use it to hide the comment forms and only show them when the users places his mouse above an "Add Comment" line.

In rants.html, add the following tag to the <head> section of the document:

```
<script type="text/javascript" src="{{ url_for('static', filename='jquery-
2.1.0.min.js') }}">
</script>
```

Notice that we can use Flask's url_for inside a Jinja template, which is very handy.

Then add the following directly above the </body> tag of your file:

```
<script>
        $(".comment_form form").hide("fast");

        function show_form(event) {
                $( this ).find("form").show("slow");
        }

        function hide_form(event) {
                $( this ).find("form").hide("slow");
        }

        $(".comment_form").mouseover(show_form);
        $(".comment_form").mouseleave(hide_form);
</script>
```

This code uses JQuery to hide all the form elements for your comments. Then it defines a function that shows a single form and a function that hides a form. The last two lines map the mouseover and mouseleave events to those functions.

For this to work, you will need to change your nested for loop so that each of the comment forms is inside a div with class attribute set to “comment_form”. Inside that div, but outside the <form> and </form> block, add the words “Add Comment”. When the user hoves over those words, the form should appear. When he moves the mouse away, the form should disappear again.

The page is still ugly, but at least it's less cluttered now.

Step 11. Glitter

Extend the project in a creative and original way. Your glitter should illustrate a feature of Flask, Jinja, or Postgres that we haven't used yet in this class. A few ideas (of course, it's even better to come up with your own):

1. Allow users to delete comments.

2. Add authentication (i.e. logins) and sessions so that you don't have to type your username multiple times.

3. Allow users to upload a small “avatar” image to display next to their username (we do have an image field in the accounts table).

Step 12. Submitting

Make a tarball of your entire Lab4 folder and turn it in as usual through the submit web site at http://marmorstein.org/~robert/submit/