Flask Tutorial

Tom Kelliher, CS 417

Tutorial Objectives:

1. Practice setting up and running a Flask application.
2. Gain familiarity with securing sensitive information (the Flask session key and database credentials, for example) from versioning systems (GitHub) and web servers (Apache).
3. Learn how to use the following when developing Flask applications:
   (a) Configuring an app.
   (b) Using decorators to associate static URLs and URLs with variable parts to Python functions in an app.
   (c) Using Jinja templates, including control structures and means for writing URLs so that they work correctly regardless of where in a web server's name space an app is installed.
   (d) Using sessions to save session state across HTTP requests.
   (e) Using forms to send request data to an app.
   (f) Writing simple HTML.

As you research tactics for getting Flask to bend to your will, you'll find lots of conflicting advice on how to start Flask and set up its environment. I strongly advise you to stick to what I've sketched-out in what follows. Specifically, refrain from creating a virtual environment. As a micro-framework, Flask doesn't contain functionality that is better provided by other Python modules. If you need other Python modules for your project, let me know and I'll install them globally for all to use. And, don't try running "`sudo pip3 install ...`". You don't have `sudo` privileges on phoenix. (Yes, we're using Python 3 with Flask.)

1 Preparing

The first step is to prepare your phoenix account for serving your web application from the Apache web server and to do so somewhat securely. While you're developing, you'll use Flask's built-in development web server. Once you've reached the production stage, you'll "deploy" your app on Apache. For safety and security, your Apache deployment is "sandboxed" — your application will only be accessible from Goucher's wired network or the GoucherWifi wireless network.

Login to phoenix and open a terminal 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.

# Run the following command:
```

```
umask

If it returns anything other than 022 or 0022, undo your umask changes by
running

umask 022

# AND placing this line at the end of the .bashrc file in your home
# directory

umask 022   # Set default permission of 'rx' for group and other.

# The following commands will ensure the security of your project files
# and sensitive information.

mkdir .secrets public_html
chmod go= public_html .secrets
setfacl -m u:www-data:rx public_html .secrets .
touch .secrets/cs417secrets.py
```

2 Running the pageCount App

Moving along, run these commands. The first thing you'll do is to get a copy of the WSGI starter code, putting it into the proper position. Run the following commands.

```
# Keep in mind:
#     public_html/wsgi will be the root directory of your web application
#     in production mode.

cd public_html

# Now run

unzip ~kelliher/Class/Cs417/cs417starter.zip

# Now run

cd wsgi

# Hidden files, aka dot files, files whose first character in the file name
# is a period, might not automatically appear when you try to open them.
# Check the documentation for your editor for instructions on showing these
# files.  This also tends to apply to directories...

# Run an editor (geany is a good choice) to open config.py in this
# directory and read the comments.  As instructed, run a command to
# generate a secret key value and then use that value to define SECRET_KEY
# in your cs417secrets.py file in the .secrets directory you created
```

```
# earlier.

# Now, open .flaskenv in your wsgi folder.  Replace the value for
# FLASK_RUN_PORT with a unique value in the range specified in this file.

# Make sure you've saved all the files you've modified.

# From the terminal, run the command

flask run

# This will run a development web server on the port you set earlier.
# Right-click on the URL that's output and open a web browser on the URL
# You should be able to interact with the web application.

# Here's what's going on in the background.  The flask command reads
# .flaskenv to get its environment.  One of the environment variables that
# it defines is FLASK_APP, which contains the name of the Python file that
# flask should run.  As given to you, this will cause flask to load
# pageCount.py  As you read pageCount.py's code, you'll notice that the
# app object is configured from config.py  The first thing done in
# config.py is inserting your .secrets directory at the beginning of
# Python's module search path.  config.py then imports everything defined
# in cs417secrets.py.  Then, the application logic is defined and decorated
# by URLs.  Finally, if the app is being run by Apache's WSGI module,
# application is defined.

# As instructed by 'flask run', type Ctrl-c to exit the development
# web server.

# In your editor, open pageCount.py and templates/pageCount.html and study
# them carefully.

# Now trying running your web app on apache.  Type a URL similar to this
# into a web browser running on phoenix:

# https://phoenix.goucher.edu/~kelliher/wsgi/pageCount.py/

# Replace my username with your username.  You should be able to interact
# with your web app using phoenix's Apache web server.

# As you develop other apps, remember to change the value of FLASK_APP
# in .flaskenv to let Flask know which app to run the development web
# server on.
```

3 Running the `form` App

```
# Edit .flaskenv and run form.py on the development server and Apache.
# Study the form.py and its template file carefully.
```

4 Developing Your Own App

See the "A Little List" app available from the course's phoenix site. Develop your own version of this. Here are some hints:

- The easiest way to wipe out session variables is to exit your web browser and restart it. It's wise to do so when you think you app is working, to prove that it's *really* working. Develop your app using the Flask's development server. Test your app using the development server and using phoenix's Apache web server.
- To see a web page's HTML source, type `Ctrl-u`. Study the source to start to reverse-engineer the app.
- I stored the list items as a list in the session variables. This was the only session variable I used. When I modified the list, I set `session.modified` to `True`. Here are some example Python statements which you'll find useful:

  ```
  session['list'] = []    # Create an empty list.
  session['list'].append(request.form['item'])    # Append an item to list.
  session.modified = True    # Used when an item is added or deleted.
  session['list'].pop(index)    # Remove an item from list
  ```

- I used Jinja's `for ... in ...` control structure to generate the HTML for each of the list items, coupled with loop.index0 to get the `<input>` `value`s associated with `id`. Here's what I used:

  ```
  {% for item in session['list'] %}
  <li> <input type="checkbox" name="id" value="{{ loop.index0 }}">
  {{ item }}
  {% endfor %}
  ```

  Using `id` to identify list items for deletion rather than the list item string is necessary because list items don't have to be unique.

- `id` is a repeated key in the `request.form` dict. Use `getlist`, a multidict method, to get all the `id` values, casting them to `int`. Here's how to collect multiple key/value pairs with the same key, in this case `id`, into a list, converting the `string`s to `int`s:

  ```
  indices = request.form.getlist('id', type=int)
  ```

- Use the `id` values to `pop` the appropriate items from the list. You have to be careful about the order in which you perform the `pop`s. For example, if the list of `id`s is `[2, 5]`, when you pop the item at position 2, then the item at position 5 is now actually at position 4. Think about how to order the `pop`s to avoid this re-indexing mess. I did so by sorting the list of `id`s in descending order.