Project Tracker - Flask

In this segment, we'll take our Project Tracker database online, turning it into a web application using Flask.

Step One

Setup

Remember that the first step for any new project is to set up your environment. This exercise is no exeception. Here's a short reminder of the process:

1. Create and activate a new, empty virtual environment.
2. Use ***pip*** to install Flask, Flask-SQLAlchemy, and psycopg2.
3. `pip freeze > requirements.txt` to make a permanent record of the exact versions of things ***pip*** installed, so you can always make another virtual environment just like this one.
4. Add your ***env/*** directory to ***.gitignore*** so that you don't accidentally check it in.
5. `git add` your ***.gitignore*** and ***requirements.txt*** file; you'll want these things in your git repository.

An example from the terminal:

```
$ virtualenv env
New python executable in env/bin/python
Installing setuptools, pip...done.
$ source env/bin/activate
(env) $ pip install flask flask-sqlalchemy
Downloading/unpacking flask flask-sqlalchemy
...
Successfully installed flask Werkzeug Jinja2 itsdangerous markupsafe
Cleaning up...
(env) $ pip install psycopg2
Collecting psycopg2
...
Installing collected packages: psycopg2
Successfully installed psycopg2-2.6.1

(env) $ pip freeze > requirements.txt
(env) $ subl .gitignore
(env) $ cat .gitignore
env/
(env) $ git init
(env) $ git add .gitignore requirements.txt
```

If you want a more in-depth review of ***pip*** and ***virtualenv***, here's a full tutorial <http://www.dabapps.com/blog/introduction-to-pip-and-virtualenv-python/>.

Next, you'll need a database. You should have dumped the database you created in the ***project-tracker-py*** exercise into a file called ***hackbright.sql*** and stored that on your GitHub. If you didn't get that far, use the one provided in this folder from when you ran ***hbget***.

Regardless of which version of ***hackbright.sql*** you use, recreate your database by running the following commands:

```
(env) $ dropdb hackbright
(env) $ createdb hackbright
(env) $ psql hackbright < hackbright.sql
(env) $
```

Changing Our Project Tracker to a Web App

In our previous project tracker script, we used the ***print*** statement to output text to the screen. Now, we want that output to go the user's browser.

We'll need to change our script to *return* the relevant information (student info, grade info, etc), and create Flask routes to get those pieces of information. (We'll leave the existing print statements in, as they're good debugging information in the console window—once everything is working, you could take them out.)

In this directory is a copy of ***hackbright.py***, the completed version of the Project Tracker from the ***project-tracker-py*** exercise. If you already have a version you wrote and it's functional, you can substitute yours for this one.

In the original version of this program, you have a function like:

```
def get_student_by_github(github):
    """Given a github account name, print information about the
    matching student."""

    QUERY = """
        SELECT first_name, last_name, github
        FROM Students
        WHERE github = :github
        """
    db_cursor = db.session.execute(QUERY, {'github': github})
    row = db_cursor.fetchone()
    print "Student: %s %s\nGithub account: %s" % (row[0], row[1],
```

We'll change this now, so that in addition to printing the output, the function returns it:

```
def get_student_by_github(github):
    """Given a github account name, print information about the
    matching student."""

    QUERY = """
        SELECT first_name, last_name, github
        FROM Students
        WHERE github = :github
        """
    db_cursor = db.session.execute(QUERY, {'github': github})
    row = db_cursor.fetchone()
    print "Student: %s %s\nGithub account: %s" % (row[0], row[1],
                                                  row[2])
    return row
```

This function now returns a tuple of ***(first name, last name, github)***. (If you're using the version of ***hackbright.py*** provided for this exercise, we've added the ***return*** for you.)

The ***project-tracker-flask*** directory also includes the skeleton of a web application, ***hackbright-web.py***.

In it is a function called ***get_student***, which is routed to the resource ***/student***:

```
@app.route("/student")
def get_student():
    """Show information about a student."""

    github = "jhacks"
    first, last, github = hackbright.get_student_by_github(github)
    return "%s is the GitHub account for %s %s" % (github, first, last)
```

Since we're going to use ***get_student*** and other function right out of our ***hackbright.py*** file, we'll need to import our existing code at the top:

```
import hackbright
```

You can test this by starting your Flask server:

```
(env) $ python hackbright-web.py
* Running on http://127.0.0.1:5000/ <http://127.0.0.1:5000/> (Press CTRL+C to quit)
* Restarting with stat
```

Visit ***http://127.0.0.1:5000/student*** and you should see information about the student.

Request Arguments

We need to make this app more better-er. At the very least, we need a way for it to display information for a student besides the one that's hardcoded. To do that, we have to collect input from the person using the browser. We'll be using a GET request.

Remember, the arguments for a GET request are a set of key/value pairs that the user can send to the web server via the URL. An example URL with GET request arguments would look like this:

```
http://127.0.0.1:5000/?key1=val1&key2=val2
```

A question mark after a URL tells the server that the remainder of the line is not part of the URL. It indicates that anything that follows is a set of key/value pairs in the form of key=val, with each pair separated by an ampersand. In Flask, the set of pairs gets transformed into a dictionary which is an attribute on a 'Request Object'. For GET requests, this dictionary is called ***args***. If you were to examine ***request.args*** from inside the handler that responds to the above URL, the dictionary would be the following:

```
{'key1': 'val1', 'key2': 'val2'}
```

We'll use this to collect the student's GitHub username from the user. The ***request.args*** variable acts like a dictionary. To be safe, just in case they don't enter anything, we'll use the familiar dictionary ***.get()*** method (no relation to ***GET*** request). Modify your ***/student*** route as follows:

```
@app.route("/student")
def get_student():
    """Show information about a student."""

    github = request.args.get('github', 'jhacks')
    first, last, github = hackbright.get_student_by_github(github)
    return "%s is the GitHub account for %s %s" % (github, first, last)
```

Now try accessing your application with the following URL, changing the GitHub account as appropriate to view other students:

> *http://127.0.0.1:5000/student?github=jhacks <http://127.0.0.1:5000/student?github=jhacks>*

If you don't pass a GitHub account name in the ***GET*** request, it will fall back to using ***jhacks***.

Using Jinja

Let's make this better by using a Jinja template so we can easily return prettier HTML.

First, add a ***templates/*** directory to your project.

Then, add a file called ***student_info.html*** to the ***templates/*** directory. In that file, write the following:

```
<!doctype html>
<html>
<head>
    <title>Student Info</title>
</head>
<body>
<h1>{{ first }} {{ last }}</h1>

<p>GitHub Account: {{ github }}</p>
</body>
</html>
```

Save your file, then we'll make changes in ***hackbright-web.py***.

We still need to call ***get_student_by_github*** to get the row. Then we need to feed the data into the template and fill in the pieces. Once we've filled in the template, we can return the string of the filled template and let the browser render it.

Update your ***get_student*** function to look like this:

```
def get_student():
    """Show information about a student."""

    github = request.args.get('github', 'jhacks')
    first, last, github = hackbright.get_student_by_github(github)
    html = render_template("student_info.html",
                           first=first,
                           last=last,
```

```
                               github=github)
    return html
```

Behold your majesty at http://127.0.0.1:5000/student?github=jhacks <http://127.0.0.1:5000/student?github=jhacks>

User Input Revisited

We do sort of have a webapp. It's an application where a user can enter some input and receive a response backed by a database. The last remaining issue here is that our input collection mechanism is pretty ugly. We can't ask our users to enter information via the URL every time. Heck, we can't even ask our users to remember what our URL is half the time. Users are pretty dumb.

The standard way to ask for data is to display a form. The user fills in the form and submits it, and our webapp takes that input and processes it as before.

Building forms (at this stage) is essentially a static process. The page displaying the form won't change dynamically with some other input, so we simply need to make an HTML page. Create a file in your ***templates/*** directory called ***student_search.html***, and put the following in it.

```
<!doctype html>
<html>
<head>
    <title>Project Tracker</title>
</head>

<body>

<form action="/student">
    <label>Enter GitHub username
        <input type="text" name="github">
    </label>
    <input type="submit" value="Display Student">
</form>

</body>
</html>
```

If you tried to view this file in our browser, perhaps by going to http://127.0.0.1:5000/student_search.html <http://127.0.0.1:5000/student_search.html>, you would get a 404 (***Not Found*** error). Remember that Flask only calls view functions based on the route provided to them with the ***@app.route()*** decorator. It's not sufficient to have a template; we have to have a handler that will display the template, too.

In ***hackbright-web.py***, add the following:

```
@app.route("/student-search")
def get_student_form():
    """Show form for searching for a student."""

    return render_template("student_search.html")
```

This handler is simpler than the other one we wrote. Whenever a user browses to http://127.0.0.1:5000/student-search <http://127.0.0.1:5000/student-search>, it will simply render the template ***student_search.html***. Load this in your browser now and examine it.

The meat of this file is the line:

```
<form action="/student">
```

Recall that the action tells the browser where to send the form values when the user clicks submit. Here, it sends the form data to the first handler that we built earlier. That handler expects the student's GitHub username in the form of a key/value pair. The value is obviously whatever the user types into the text field, but where is the key?

Remember, the key for a value input by a user is stored on the HTML element as the name attribute. Whew. That was a mouthful. Just look at the following line:

```
<input type="text" name="github">
```

This HTML tag displays a text field for the user to type in. The ***name*** attribute on the tag specifies the key that will be used for the value when the form is submitted. If that's unclear, try entering a value then submitting the form. Inspect the URL request arguments on the next page. Go back, change the name attribute, submit again, and inspect the URL again and see how they've changed.

If you were being observant, you would have noticed that the first time you submitted, everything **worked**. You now have a webapp–a proper one even. It's a little bit duct-taped together, but it's still complete. It displays a form to a user, collects data, processes the submitted form, then displays the result.

Furthermore, now we have a context we can use to talk about *all* webapps. Every webapp is a series of page-pairs. One for displaying a form, and one for processing it. Sometimes multiple forms show up on the same page, and sometimes two different forms end up getting processed in the same way, but the principle still stands. If you think of webapps as a series of forms, you can decompose and reconstruct pretty much any webapp in existence. Yes, even Facebook.

Step Two

Well, our app works, but it's pretty brittle. If you leave the form empty and submit, it breaks. If you enter a user who's not in the database, it breaks. Basically, if you look at it funny, it breaks. That's okay, because Flask gives us a bunch of tools to make our apps significantly more robust. We're not going to worry about them here; right now we're going to focus on the general layout of our app and the integration of HTML and Python.

Here are your tasks:

Student Creation Form

Make an HTML form for creating a new student. Make a pair of routes to both display the form and to handle the form results. This should add a new student to the database.

> **Note: POST Requests**
>
> Remember that ***GET*** requests are good for forms that don't have "side effects" but that forms with side effects should use ***POST*** requests.
>
> To do this, you'll need to change the form to use `method="POST"`.
>
> You'll also need to do two things in your ***hackbright-web.py***:
>
> - On the function declaration for the student-adding method, you'll need to specify that this route should accept ***POST*** requests. For security reasons, Flask requires that you say this specifically, rather than assuming that all routes handle ***POST*** requests. You can do this with a method defined like:
>
> ```
> @app.route("/student-add", methods=['POST'])
> def student_add():
>     """Add a student."""
> ```
>
> - Instead of looking for URL variables with ***request.args***, you'll need to use ***request.form***.

The method that handles the form results should return a template that acknowledges the user was added. Have it provide a link (via an `<a>` tag) to the information page about the student.

STOP and ask for a code review.

Grade Listing for Student

On the page provided by the ***/student*** route, instead of just showing the first name, last name, and github, also show a listing of all of the projects a student has completed, along with their grade for that project.

To do this, you'll need use your ***hackbright.py*** code, which includes a function that returns a **list of tuples**, where each tuple contains the project title and grade.

You'll need to edit the ***student_info.html*** template to loop over the list of titles/grades and display them. Do that in a bulleted list (use HTML `<ul>` and `<li>` for this).

STOP and ask for a code review.

Project Listing

Make a page that lists information about a project (using a project title given in a ***GET*** request). Route this to ***/project***.

This should list the title, description, and maximum grade of a project.

Change the student information template so that, when it lists a project title, this should become a link to this project information page.

STOP and ask for a code review.

Intertwining

On the project information page, list all of the students who have completed that project and the grade they received.

This is similar to the task in the “Grade Listing for Student” section. You’ll need to find the function in ***hackbright.py*** that returns a list of tuples.

You’ll need to edit the ***project_info.html*** (or whatever you called it) template to loop over the list of students/grades and display them. Do that in a bulleted list.

And, when you list the student name, make that a link to their student information page, so that it’s easy to jump to the information about the student.

STOP and ask for a code review.

Homepage

Make a homepage that has two lists – one for all students, and one for all projects. Both should be bulleted lists and both should be made up of live links to the full student and project pages.

STOP and ask for a code review.

Remember to use ***pg_dump hackbright > hackbright.sql*** and check ***hackbright.sql*** into git along with your other files when you are done.

If you have extra time, you may want to look at the Further Study <further-study.html>.