

## Using Python Flask

### Overview:

This document provides configuration details and some sample code that takes advantage of the Flask micro web framework. For this course, we will use Flask to develop secure Web applications using Python. Be aware additional functionality is available in other full stacks.

### Background:

According to the Flask web site, “Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions” (Flask, 2019). Frameworks are typically a bundle of libraries providing significant functionality. A microframework contains a smaller subset of functionality when compared to a full stack. Django is considered a full stack framework whereas Flask is a microframework.

Your readings for this week provided an overview of HTML with specific HTML code for creating lists and links. Since we will be using Flask, there are some configurations needed to be successful. In addition, we won't be using specific flask tag generations and modules such as flask-table. We will be using Flask templates allowing creating of HTML pages outside of the Python code then rendering the template from within Python.

### Installing Flask

To install Flask, open up your command prompt, and type:

```
pip install flask
```

You will receive an installation complete message upon success. As before, if you receive a message about updating pip, you should ignore this.

As shown in figure 1, you can list the modules currently installed by starting python and typing:

```
help('modules')
```

Note, your list of packages will look different than the ones below and is based on the previous and default module python installations.

```
C:\Users\jim>python
Python 3.8.3rc1 (tags/v3.8.3rc1:802eb67, Apr 29 2020, 21:39:14) [MSC v.1924 64 b...

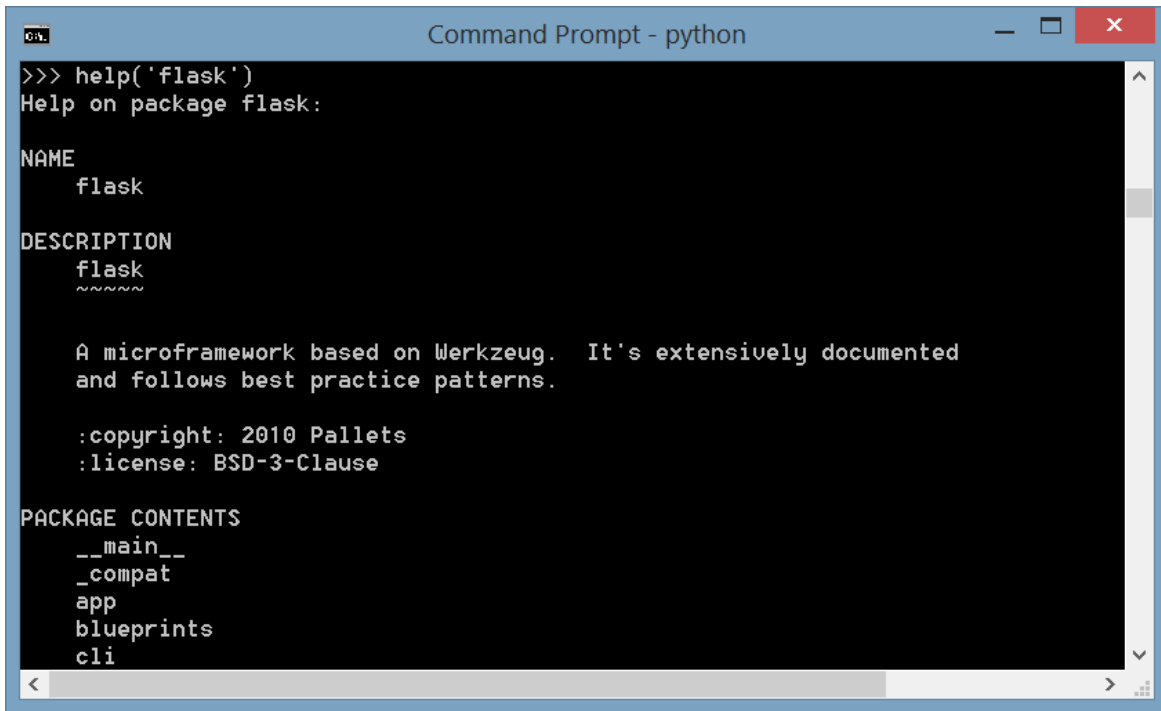
Type "help", "copyright", "credits" or "license" for more information.
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

Movies          asyncore        errno           pylab
PutMovie        atexit         faulthandler   pylint
__future__      audioop        filecmp        pyparsing
_abc            bandit         fileinput      pytz
_ast            base64         flask          queue
_asyncio        bdb            fnmatch        quopri
_bisect         binascii       formatter      random
_blake2         binhex         fractions      re
_bootlocale     bisect         ftplib         reprlib
_bz2            brain_argparse  functools      rlcompleter
_codecs         brain_attrs    gc             runpy
_codecs_cn      brain_boto3    genericpath    sched
_codecs_hk      brain_builtin_inference  getopt         secrets
_codecs_iso2022 brain_collections  getpass        select
```

Figure 1 Running `help('modules')` within Python

You can also list help on specific modules using `help('module_name')`. For example, `help('flask')` as shown in figure 2.



```
>>> help('flask')
Help on package flask:

NAME
    flask

DESCRIPTION
    flask
    ~~~~~

    A microframework based on Werkzeug. It's extensively documented
    and follows best practice patterns.

    :copyright: 2010 Pallets
    :license: BSD-3-Clause

PACKAGE CONTENTS
    __main__
    _compat
    app
    blueprints
    cli
```

Figure 2 Running `help('flask')` within Python

Once you have successfully installed the module, you can use it by properly importing it within your code and calling the available functions.

### Hello World Example:

After installing Flask, it makes sense to test a simple hello world program. To do this, create a folder for you to experiment with Flask applications. For example, you could create a folder structure such as `c:\courses\SDEV300\Flaskprojects\hello`. Then create a file named `hello_sdev.py` and place in the folder you just created. The contents of `hello_sdev.py` are listed below:

```
from flask import Flask

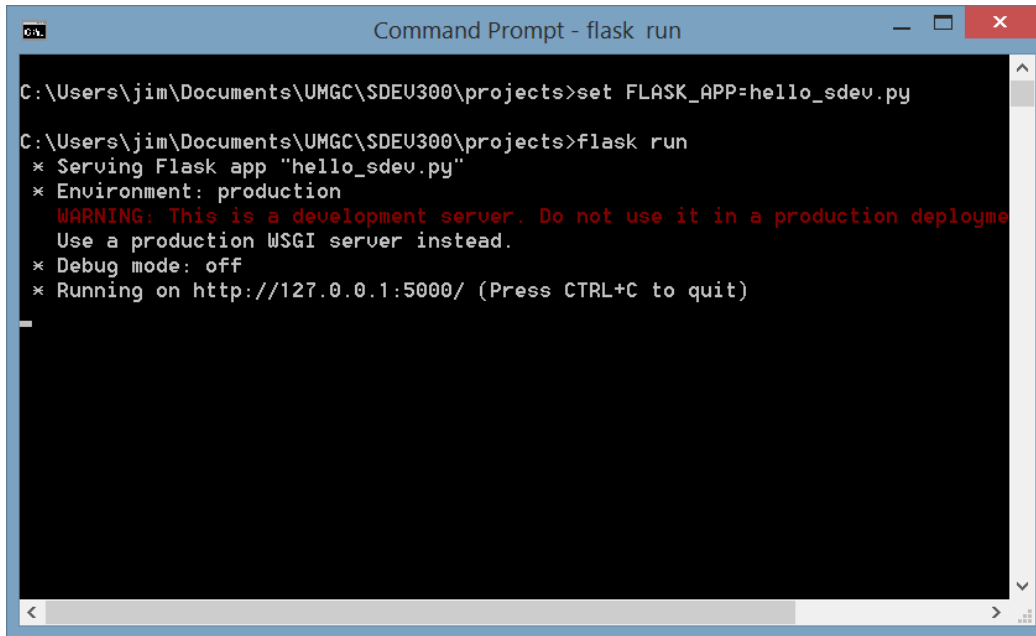
app = Flask(__name__)

@app.route('/')
def index():
    return show_hello()

# Can call functions as part of the returns
def show_hello():
    return 'Hello, UMGC SDEV Students!'
```

To run the code, you need to set an environmental variable aligned with the file we just created with the Flask app, and then run flask. The following commands entered from the command prompt, will satisfy these requirements resulting in an output similar to figure 3.

```
set FLASK_APP=hello_sdev.py  
flask run
```



```
Command Prompt - flask run  
C:\Users\jim\Documents\UMGC\SDEV300\projects>set FLASK_APP=hello_sdev.py  
C:\Users\jim\Documents\UMGC\SDEV300\projects>flask run  
* Serving Flask app "hello_sdev.py"  
* Environment: production  
  WARNING: This is a development server. Do not use it in a production deployment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Figure 3 Setting Flask Variable and Run Flask

To test the results, open your favorite browser and type the following URL:

```
localhost:5000
```

The results are shown in figure 4.

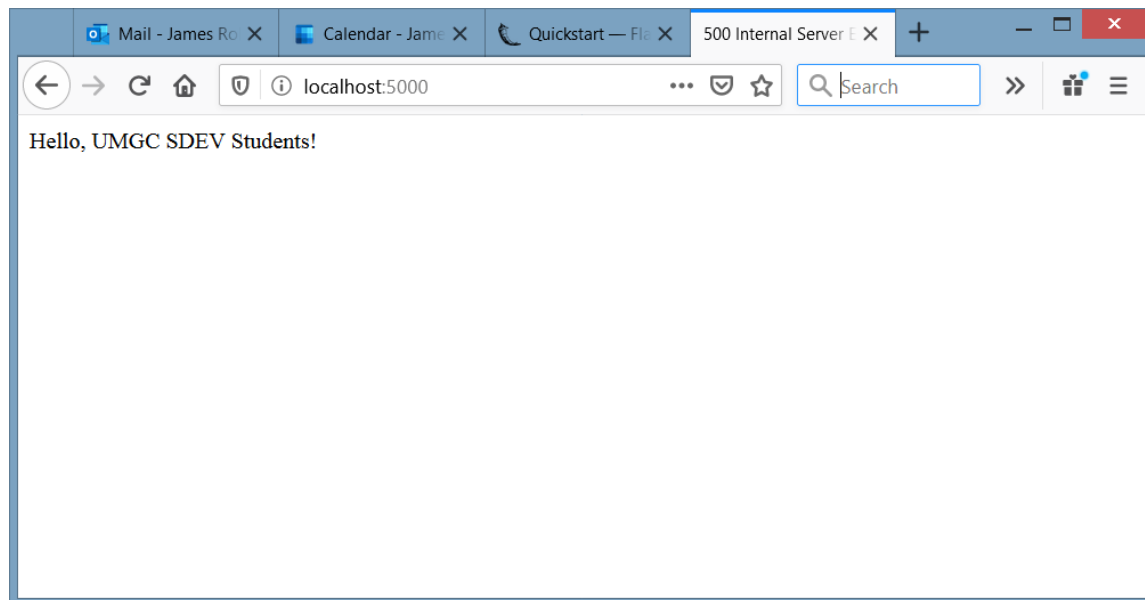


Figure 4 Testing the Hello Web Page

Flask and web application code will look a little different than some of the Python code you have created previously. Here are a few comments about the code and environment:

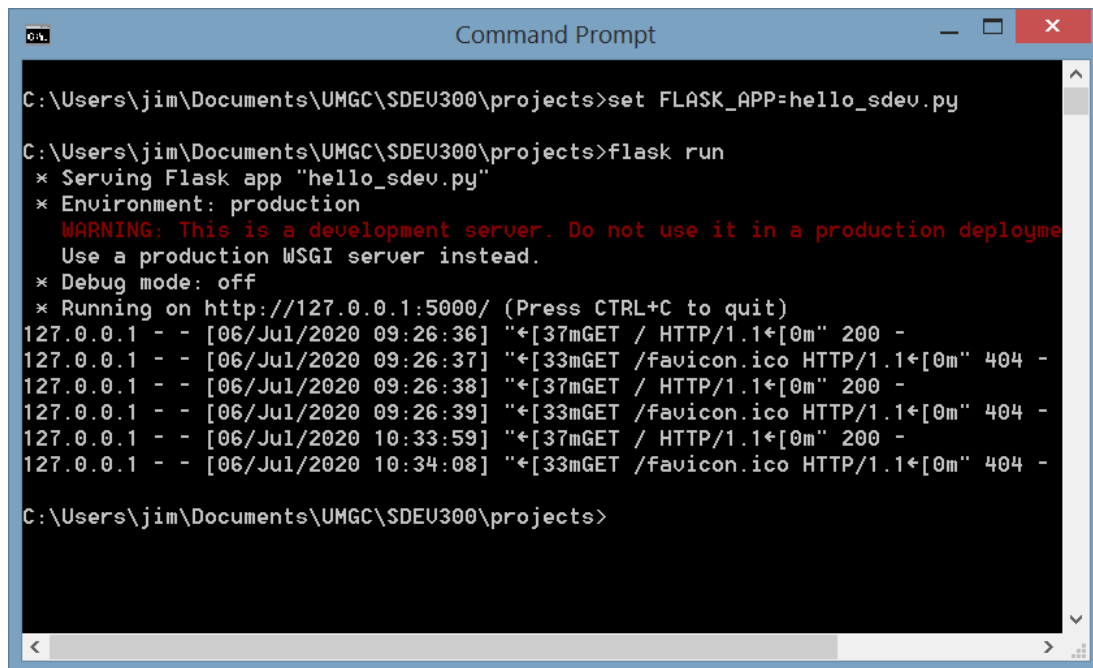
1. The Web server is running locally on your desktop. This is why you used localhost as the URL. Using 127.0.0.1:5000 will yield similar results. If you need to deploy the code to another server at some point, you would provide the URL of the production server.
2. The default port of Flask is 5000. Be sure to use [URL:port](#) to run the application. Keep in mind, the default URL port is port 80. This is where most HTTP web applications run. The secure version of HTTP is HTTPS and would run on port 443. For testing, running on port 5000 will suffice.
3. `app = Flask(__name__)` creates an instance of the class. This is an important object-oriented concept allowing us to call and use other methods and functions associated with Flask.
4. The `route()` decorator helps us align and route the URL to invoke the corresponding function. In this case, `'/'` is the default landing page. We could, and will soon, add more URL routes such as `/home`, `/about`, `/login` and others. Routes are virtual locations. Once we add more routes they are invoked by just adding the route to the URL. For example:

`localhost:5000/home`

### Stopping the Flask Web Application:

Assuming there are no errors in your code, the Flask app will run on your desktop until you quit the application. To quit the application just use CTRL+C. Sometimes you may need to enter CTRL+C a few times before the application will exit.

While running, some debug and information is available and displayed directly in the console where the Flask application was launched. As shown in figure 7, the information is updated each time a user accesses the URL. The command line prompt will return once the application has been stopped.



```
C:\Users\jim\Documents\UMGC\SDEV300\projects>set FLASK_APP=hello_sdev.py

C:\Users\jim\Documents\UMGC\SDEV300\projects>flask run
* Serving Flask app "hello_sdev.py"
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [06/Jul/2020 09:26:36] "[37mGET / HTTP/1.1[0m" 200 -
127.0.0.1 - - [06/Jul/2020 09:26:37] "[33mGET /favicon.ico HTTP/1.1[0m" 404 -
127.0.0.1 - - [06/Jul/2020 09:26:38] "[37mGET / HTTP/1.1[0m" 200 -
127.0.0.1 - - [06/Jul/2020 09:26:39] "[33mGET /favicon.ico HTTP/1.1[0m" 404 -
127.0.0.1 - - [06/Jul/2020 10:33:59] "[37mGET / HTTP/1.1[0m" 200 -
127.0.0.1 - - [06/Jul/2020 10:34:08] "[33mGET /favicon.ico HTTP/1.1[0m" 404 -

C:\Users\jim\Documents\UMGC\SDEV300\projects>
```

Figure 5 Stopping the Web Application

### Templates and Functions:

As you add more routes and functionality to the Web page, it makes sense to use Flask templates to separate the Python code from the HTML. A simple HTML template is shown below:

```
<!doctype html>

<title>Hello UMGC SDEV Student!</title>

{% if name %}

    <h1>Hello {{ name }}!</h1>

{% else %}

    <h1>Hello, World!</h1>

{% endif %}
```

This template has some logic in it as well as standard HTML tags. Notice the “if else” structure embedded with the `{% %}` braces.

All templates should be stored in a **“templates”** folder in the same folder as your main application Python file. A static folder is used for CSS files, Images and other static content. Figure 6 shows a representative folder structure with these folders in place. In this case, the Python application is named `hello.py`.

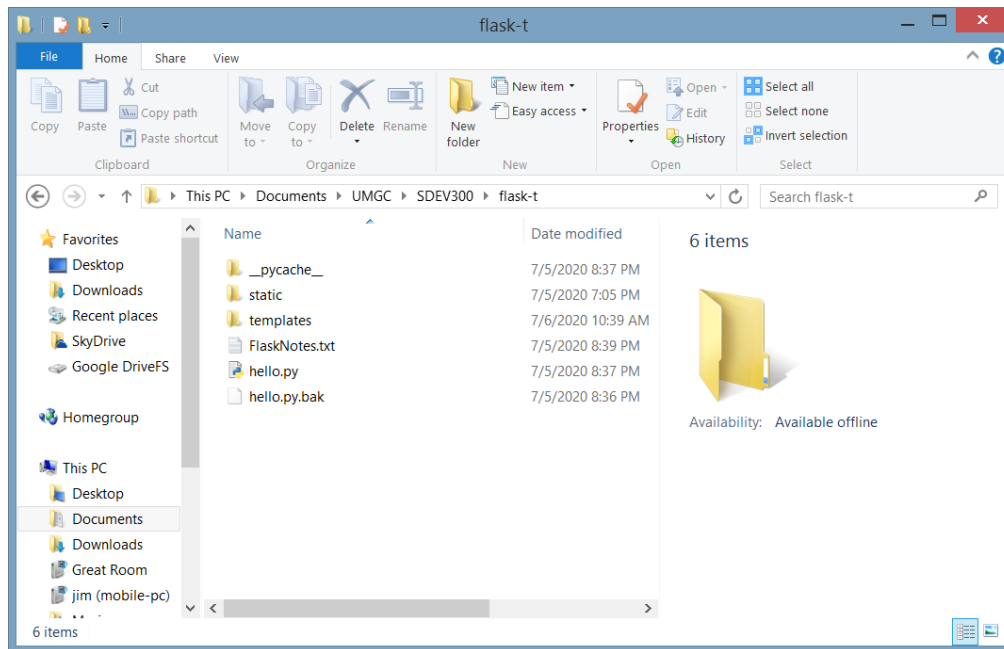


Figure 6 Flask folder structure

To call a template, you need to import the `render_templates` module and then embed the function as shown in the following code:

```
from flask import Flask
from flask import render_template

app = Flask(__name__)

@app.route('/')
def index():
    return show_hello()

# Can call functions as part of the returns
def show_hello():
    return 'Hello, UMGC SDEV Students!'

# Can provide multiple route versions
# And can render template - found in /template folder
@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

This Python file is similar to the hello world but uses the `render_template()` method to refer to the 'hello.html' file. The routes can be overloaded such that multiple URLs are possible. In this case `/hello/` as well as `hello/name` will invoke two different results. Figure 7 shows the `hello/name` option with the name of "jim" was added to the URL. Figure 8 shows the results if you just `hello/` without the name was entered.

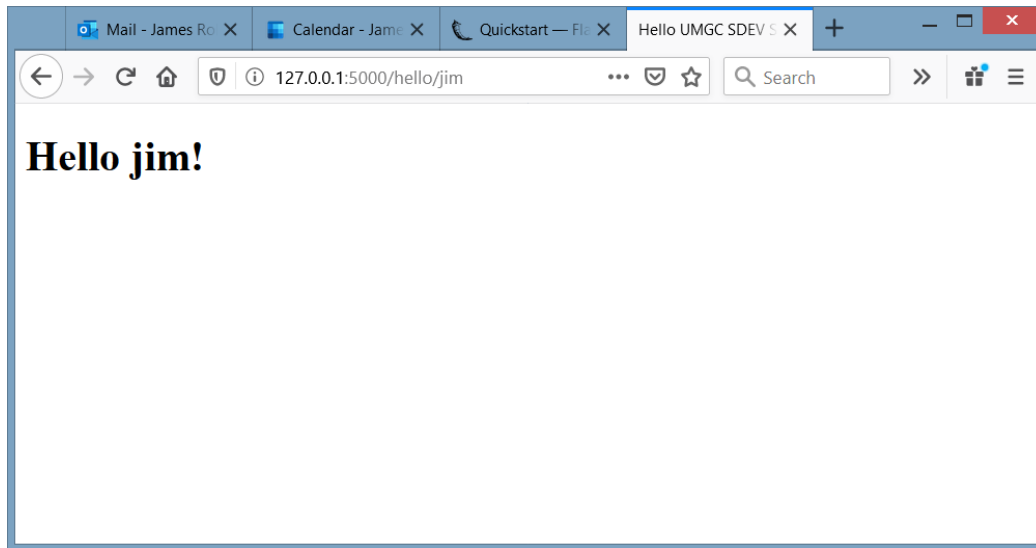


Figure 7 Using the Overloaded hello/name URL

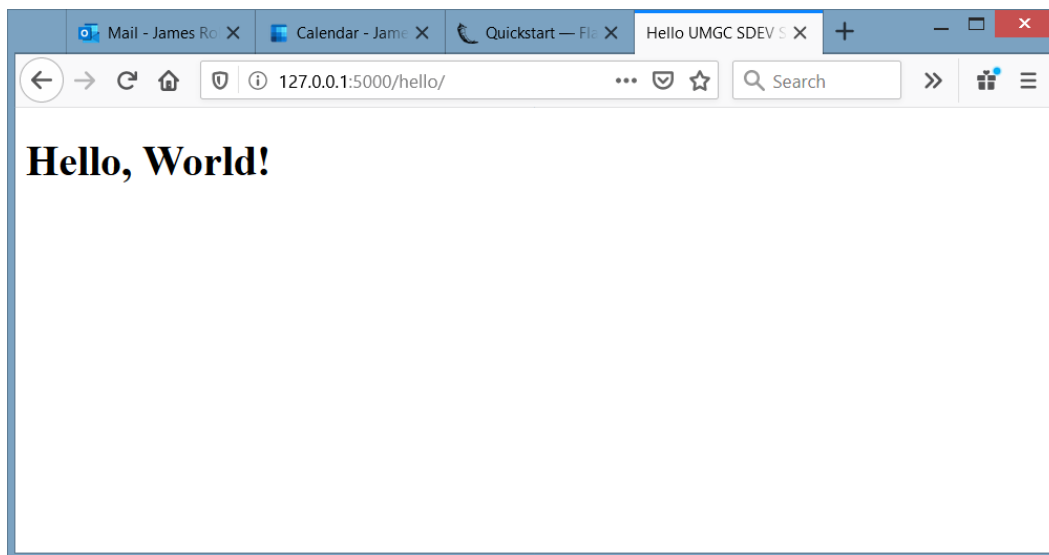


Figure 8 Calling the render\_template with hello.html

Continuing to add routes that use the `render_templates()` call, you can provide forms, tables and most any other HTML tag functionality you need to create an interesting web site. The following code is a snippet from an app that includes a user registration form:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        error = None
        if not username:
            error = "Username is required."
```



```

elif not password:
    error = "Password is required."
    flash(error)
    # TODO: Encrypt and Store credentials
    if error == None:
        return redirect(url_for("hello"))
return render_template('register.html')

```

There is much more functionality happening here including GET/POST methods, requests, error processing, and redirects. More complex functionality will be introduced and described over the next couple of weeks. However, note the concept and resulting functionality is similar. You have a specific route, in this case register, which by default will display the register.html form but if the form is submitted via POST method, the data will be processed and eventually encrypted and stored in a database system for authentication on future logins.

The representative register.html template is shown below:

```

<!doctype html>
<html>
<title>Register for the Application</title>
<h1>Register</h1>
<form method="post">
    <table>
        <tr>
            <td><label for="username">Username</label></td>
            <td><input name="username" id="username"
required></td>
        </tr>
        <tr>
            <td><label for="password">Password</label></td>
            <td><input type="password" name="password" id="password"
required></td>
        </tr>
        <tr>
            <td colspan='2'><input type="submit" value="Register"></td>
        </tr>
    </table>
</form>
</html>

```

Notice the template is a form embedded within a table using standard HTML5 tags and syntax.

Here are some summary notes and suggestions about this process and Flask code and project:

1. Use the @route decorators and HTML templates. This way you can test your HTML outside of the Flask application if needed.
2. Use the simple hello\_sdev.py examples as a starting point

3. Code a little and test a lot to avoid having to track multiple errors in your application.
4. Use the Flask, HTML and other references provided in this week's readings to expand from the simple examples I provided to much more complex web applications.