Introduction to Flask

What is Flask

- Flask is a very lightweight and easy to use framework to make it quick and easy to create Web based applications using Python
- It is an alternative to the more complex and heavyweight Django framework
- Pinterest and LinkedIn use the Flask framework

Installation

- Install Python
  - 2.6 or higher
  - 3.2.x or higher
- Install Flask using pip
  - pip install flask

Create a Minimal App

- To create a simple Flask ‘hello world’ example

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'


if __name__ == '__main__':
    app.run()
```

- Now visit http://localhost:5000
  - It’s all up and running already!

Routing

- Flask applications map methods to routes in the browser using @app.route statements

```
@app.route('/')
def hello_world():
    return 'Hello World!'
```

- Multiple methods can be provided for different URL patterns

```
@app.route('/')
def index():
  return 'Index Page'
@app.route('/hello')
def hello():
  return 'Hello, World'
```

Parameters

- Aspects of the URL can easily be used as parameters to the methods

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id
```

- Type information can also be specified
  - Such as **int**:post_id

Parameter Types

- The following types are supported

string	accepts any text without a slash (the default)
int	accepts integers
float	like int but for floating point values
path	like the default but also accepts slashes
any	matches one of the items provided
uuid	accepts UUID strings

Handling HTTP Methods

- Methods can also be routed based on HTTP method

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

- All HTTP methods are supported
  - Get, Post, Put, Delete, Options, Head

Templating

- Flask incorporates the Jinja2 templating engine for HTML content generation
  - Templates have access to session, request, and other commonly used variables

```
@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
   return render_template('hello.html', name=name)
```

templates/hello.html

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
<h1>Hello {{ name }}!</h1>
{% else %}
<h1>Hello, World!</h1>
{% endif %}
```

Creating an Enterprise App

- We have seen the basics, so now lets see how to create a database aware enterprise application
  - We will use SQLite as the database
- The example is based on the official Flask tutorial
  - http://flask.pocoo.org/docs/0.12/tutorial/

Enterprise App Folders

- The folder structure required will be
  - EnterpriseApp/templates - for our page templates
  - EnterpriseApp/static - for static assets such as images/css and so on
  - EnterpriseApp/schema - for our database schema

Creating the Skeleton Application

- Within the EnterpriseApp project folder you can create an EnterpriseApp.py file

```
import os
import sqlite3
from flask import Flask, request, session, g, redirect, url_for, abort, render_template, flash

app = Flask(__name__)
app.config.from_object(__name__)  # load config from below in this file

# Here is the config that will be loaded by the above
app.config.update(dict(
    DATABASE=os.path.join(app.root_path, 'enterprise.db'),
    SECRET_KEY='development key',
    USERNAME='admin',
    PASSWORD='default'
))
# can use a config file instead with this name
app.config.from_envvar('ENTERPRISE_SETTINGS', silent=True)
```

Managing Database Connections

- Database connections can be created using a simple function

```
def connect_db():
    """Connects to the specific database."""
    rv = sqlite3.connect(app.config['DATABASE'])
    rv.row_factory = sqlite3.Row
    return rv
```

- To optimise performance the connection can be cached

```
def get_db():
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db
```

What is g?

```
def get_db():
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db
```

- This is for global variables where the value is only valid for one request
  - http://flask.pocoo.org/docs/0.12/api/#application-globals

Closing Connections

- Connections can be closed at the end using a decorated function

```
@app.teardown_appcontext
def close_db(error):
    if hasattr(g, 'sqlite_db'):
        g.sqlite_db.close()
```

Creating the Database

- The database can also be created using flask
- A function can be added with a decorator that enables it to be run using the flask CLI

```
def init_db():
    db = get_db()
    with app.open_resource(‘schema/schema.sql', mode='r')
as f:
        db.cursor().executescript(f.read())
    db.commit()

@app.cli.command('initdb')
def initdb_command():
    init_db()
    print('Initialized the database.')
```

Running the flask CLI

- To use the CLI, an environment variable can be set referring to your entry point python file

```
set FLASK_APP=EnterpriseApp.py
```

- Then the CLI can be invoked to call your new decorated function

```
flask initdb
```

Handling Views

- Below is a view function to return the list of items

```
@app.route('/')
def show_entries():
    db = get_db()
    cur = db.execute('select title, text from entries order by id desc')
    entries = cur.fetchall()
    return render_template('show_entries.html', entries=entries)
```

The Default Template

- Here is a template for the list of items that also contains a form to allow for the creation of new items

```
{% extends "layout.html" %}
{% block body %}
  {% if session.logged_in %}
    <form action="{{ url_for('add_entry') }}" method=post class=add-entry>
      <dl>
        <dt>Title:
        <dd><input type=text size=30 name=title>
        <dt>Text:
        <dd><textarea name=text rows=5 cols=40></textarea>
        <dd><input type=submit value=Share>
      </dl>
    </form>
  {% endif %}
  <ul class=entries>
  {% for entry in entries %}
    <li><h2>{{ entry.title }}</h2>{{ entry.text|safe }}
  {% else %}
    <li><em>No entries here so far</em>
  {% endfor %}
  </ul>
{% endblock %}
```

The ‘Base’ Layout

- Layouts can extend other layouts with the base layout containing common content

templates/layout.html

```
<!doctype html>
<title>Enterprise Example</title>
<link rel=stylesheet type=text/css href="{{ url_for('static',
filename='style.css') }}">
<div class=page>
  <h1>Enterprise Example</h1>
  <div class=metanav>
  {% if not session.logged_in %}
    <a href="{{ url_for('login') }}">log in</a>
  {% else %}
    <a href="{{ url_for('logout') }}">log out</a>
  {% endif %}
  </div>
  {% for message in get_flashed_messages() %}
    <div class=flash>{{ message }}</div>
  {% endfor %}
  {% block body %}{% endblock %}
</div>
```

Handling New Entries

- When the form is submitted, the following function will be triggered

```
@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('logged_in'):
        abort(401)
    db = get_db()
    db.execute('insert into entries (title, text) values (?, ?)',
               [request.form['title'], request.form['text']])
    db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('show_entries'))
```

- Flash messages can be shown in the templates

```
{% for message in get_flashed_messages() %}
    <div class=flash>{{ message }}</div>
  {% endfor %}
```

Logging In and Out

- Simple logging in and out can be handled by routing functions

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] !=
app.config['USERNAME']:
            error = 'Invalid username'
        elif request.form['password'] !=
app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('show_entries'))
    return render_template('login.html', error=error)
```

```
@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('show_entries'))
```

```
<h2>Login</h2>
<form action="{{ url_for('login') }}" method=post>
    <dl>
        <dt>Username:
        <dd><input type=text name=username>
        <dt>Password:
        <dd><input type=password name=password>
        <dd><input type=submit value=Login>
    </dl>
</form>
```

Styling

- A CSS can be placed in the static folder and then referenced from the layout.html page

static/style.css

```
body          { font-family: sans-serif; background: #eee; }
a, h1, h2      { color: #377ba8; }
h1, h2         { font-family: 'Georgia', serif; margin: 0; }
h1            { border-bottom: 2px solid #eee; }
h2            { font-size: 1.2em; }
```

templates/layout.html

```
<!doctype html>
<title>Enterprise Example</title>
<link rel=stylesheet type=text/css href="{{ url_for('static', filename='style.css') }}">
<div class=page>
..
```

REST API Creation

- Flask can also be used to create a REST API via an extension called **Flask-Restful**
- A minimal app is shown below

```
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True)
```

Installation

- The installation process is trivial by simply running
  - pip install flask-restful

Flask Restful Resources

- When using Flask Restful you can specify **resources**
- Resources can easily have HTTP methods applied to them

```
class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/<string:todo_id>')
```

Testing with the requests Library

- The REST API can then be tested with the requests library from Python

```
>>>from requests import put, get
>>>put('http://localhost:5000/todo1', data={'data': 'Remember the milk'}).json()
{u'todo1': u'Remember the milk'}
>>>get('http://localhost:5000/todo1').json()
{u'todo1': u'Remember the milk'}
```

Setting Response Codes and Headers

- It is also possible to set response codes and headers

```
class Todo1(Resource):
    def get(self):
        # Default to 200 OK
        return {'task': 'Hello world'}

class Todo2(Resource):
    def get(self):
        # Set the response code to 201
        return {'task': 'Hello world'}, 201

class Todo3(Resource):
    def get(self):
        # Set the response code to 201 and return custom headers
        return {'task': 'Hello world'}, 201, {Content-type': ‘application/json'}
```

Complete Example

- To see a complete CRUD type example, view FlaskRestfulFullCrud.py in the demos here:
  - http://flask-restful.readthedocs.io/en/latest/quickstart.html#full-example