Web Services with Flask

Due Thursday, February 26, 2020, 5 PM

In this Lab we explore the construction of a web service with Flask (http://flask.pocoo.org), a microframework for Python. We chose Flask because it's a prominent framework for Python, it's simple, and it's emblematic of frameworks for in other languages.

Learn Flask and you'll find NodeJS (Javascript), Rails or Sinatra (Ruby), or Beego (Golang) immediately familiar. That's because the concepts behind all web frameworks are essentially the same, although details vary.

A Todo List App

In this lab, we'll create the backend of a very simple app: a todo manager.

The app will consist of the following entities:

- `users`
- `tasks`

This app lets individual users manage their tasks. That is, users will log in to the app and be able to list their tasks, create or delete tasks, and edit tasks or update the status of their tasks. Users will only be able to see their own tasks, not of other users.

Concepts and Plan

The accompanying lecture covers the main concepts: web services, their relevance to data science, HTTP, REST APIs, routing, and so on. We'll assume you are familiar with the basics already.

The plan with the todo app is to work incrementally: to begin on the outside and proceed inward. Here 'outside' is the API. We'll begin by 'stubbing' out one. That is, we'll create one resource (users) and define the operations we might want to perform on these users. Over the next couple of weeks we'll add tasks and create a persistence layer with a database. By the time you are done, you will have a complete app that can be deployed to the cloud and accessed from command-line clients, and web, mobile or desktop apps.

Users

We'll track the following information about `users` :

```
{
    id: 'unique ID',
    first_name: 'first name',
    last_name: 'last name',
    email: 'email address',
}
```

- GET `/users` : list all users
- GET `/users/{:id}` : retrieve a user detail by ID
- POST `/users` : create a user
- PATCH `/users/{:id}` : update a user indexed by ID
- DELETE:

Tasks

This resource manages courses. It will have the following attributes:

```
{
    id: 'unique ID of this task',
    user_id: 'id of user who owns this task'
    name: 'task name',
    status: 'done or not',
}
```

- GET `/tasks?user_id` : list all tasks for this user
- GET `/tasks/{:id}/?user_id={:user_id}: retrieve a task` by id `for a given` user_id`
- POST `/tasks` : create a task. Details of the course (`name` , `user_id` , `status` , etc) will be passed as a JSON object in the POST body)

Lab/Homework: A Flask API for a User Resource

For this lab, we'll have to leave the confines of the Jupyter notebook to run the app server. The API will be implemented with the Flask framework. Source code for a skeleton starter project is here (https://www.dropbox.com/s/qnaw0hh73huqk3z/tasks-api-stub.zip?dl=0). Download and unzip the file. This will create a folder called `tasks-api-server` .

`cd` into the folder and start up flask like so:

```
$ export FLASK_APP=api.py
$ export FLASK_ENV=development
$ flask run
```

By default this will start the server on port `5000` .

You can try accessing the server by navigating to `http://localhost:5000/users` on your browser.

I have taken the liberty of stubbing out the operations to list, create, read, and update `users` in. Your job in this exercise is understand the structure of a flask app and fill in the implementation of each stubbed operation. Note that in this iteration of the app, the `user` resource isn't persisted yet. Instead, you will store updates to `users` in a Python array. This means that all changes will be lost whenever the server is taken down. We'll be extending this app with a database layer in the coming weeks.

To simplify your task, I've also created a set of unit tests below to ensure that your API adheres to some basic requirements. Study the the unit test code closely. It shows basic usage of `requests` , a Python library for HTTP clients. I highly recommend your looking at the documentation here (http://docs.python-requests.org/en/master/).

Imports

In [44]:
```python
import unittest
import requests
import json

# The base URL for all HTTP requests
BASE = 'http://localhost:5000/users'

# set Content-Type to application/json for all HTTP requests
headers={'Content-Type': 'application/json'}
```

Problem Set

60 Points Total

You'll be implementing the following operations on a `/users` resource:

METHOD	Description
`GET /users`	List all users
`POST /users`	Create a User
`GET /users/:id`	Retrieve a user by `id`
`PUT/PATCH /users/:id`	Update a user with given `id`
`POST /users/:id/deactivate`	Deactivate a user

Note that we won't support `DELETE` on this resource. We'll want to prevent users from being deleted. This is because they are typically retained for historical purposes. For this reason, you implement a `/deactivate` operation instead.

Problem 1: List Users

10 Points

Modify `api.py` to retrieve the collection of users. Essentially, you will return the contents of the `USERS` as JSON. See the documentaton of `jsonify` (http://flask.pocoo.org/docs/1.0/api/) for details on how to convert a Python object to JSON.

Run the test below to show that your code is correct.

In [45]:
```python
class Problem1Test(unittest.TestCase):

    # test
    def test_users_get_collection(self):
        r = requests.get(BASE, headers = headers)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertEqual(type(j), list)
        self.assertGreater(len(j), 0)

        # extract the first element of the list
        first = j[0]

        # check all attributes exist
        self.assertIn('id', first)
        self.assertIn('first', first)
        self.assertIn('last', first)
        self.assertIn('email', first)



# Run the unit tests
unittest.main(defaultTest="Problem1Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_get_collection (__main__.Problem1Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.067s

OK
```

Out[45]:
```
<unittest.main.TestProgram at 0x10d85cd90>
```

Problem 2: Retrieve a Single User

10 Points

Add a method to retrieve a single user by ID. That is create a function that will route to

```
GET /users/<id>
```

See the Flask documentation for Routing (http://flask.pocoo.org/docs/1.0/quickstart/#routing) for details on how to bind a parameter to function argument.

This method shall return an HTTP status code of `200` on success and `404` (not found) if the user with the specified ID does not exist. See the unit tests below.

```
In [46]: class Problem2Test(unittest.TestCase):

             def test_users_get_member(self):

                 r = requests.get(BASE + '/0')
                 self.assertEqual(r.status_code, 200)
                 print(r.headers)
                 j = r.json()

                 self.assertIs(type(j), dict)
                 self.assertEqual(j['id'], 0)
                 self.assertIn('first', j)
                 self.assertIn('last', j)
                 self.assertIn('email', j)

             def test_users_wont_get_nonexistent_member(self):

                 r = requests.get(BASE + '/1000')
                 self.assertEqual(r.status_code, 404)

         # Run the unit tests
         unittest.main(defaultTest="Problem2Test", argv=['ignored', '-v'], exit=F
         alse)
```

```
test_users_get_member (__main__.Problem2Test) ... ok
test_users_wont_get_nonexistent_member (__main__.Problem2Test) ...

{'Content-Type': 'application/json', 'Content-Length': '125', 'Server':
'Werkzeug/0.15.4 Python/3.7.0', 'Date': 'Mon, 24 Feb 2020 01:32:16 GM
T'}

ok

----------------------------------------------------------------------
Ran 2 tests in 0.030s

OK
```

```
Out[46]: <unittest.main.TestProgram at 0x10d85cdf0>
```

Problem 3: Create a User

10 Points

Create a user with the following route:

```
POST /users
```

The object to be created will be passed as JSON in the HTTP body. The unit test below shows how. It will be of the form:

```
{
    'first': 'first name',
    'last': 'last name',
    'email': 'email address',
}
```

Use `request.get_json()` to extract the body as JSON from the HTTP request.

All of these parameters are required and your code should enforce this. If validation succeeds, add the new user to the `USERS` list and give it a unique ID.

Return HTTP status code `201` (created) if the operation succeeds and `422` (Unprocessable Entity) if validation fails.

The created user will be returned as JSON if the operation succeeds.

Future versions of your app will enforce validation constraints more rigorously.

In [47]:
```
class Problem3Test(unittest.TestCase):


    def test_users_create(self):
        data = json.dumps({'first': 'Sammy', 'last': 'Davis', 'email':
'sammy@cuny.edu'})

        r = requests.post(BASE, headers = headers, data = data)
        self.assertEqual(r.status_code, 201)

    def test_wont_create_user_without_first_name(self):
        # simple validation (missing parameters)
        data = json.dumps({'last': 'Davis', 'email': 'sammy@cuny.edu'})

        r = requests.post(BASE, headers = headers, data = data)
        self.assertEqual(r.status_code, 422)


# Run the unit tests
unittest.main(defaultTest="Problem3Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_create (__main__.Problem3Test) ... ok
test_wont_create_user_without_first_name (__main__.Problem3Test) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.033s

OK
```

Out[47]:
```
<unittest.main.TestProgram at 0x10d81b9d0>
```

Problem 4: Update a User

10 Points

Update a user with the following route:

```
PATCH/PUT /users/<id>
```

The parameters will be passed in the HTTP body and will be an object with a subset of the user attributes.

Return status code `200` on success, `404` if the user was not found, and `422` if another error occurred.

In [48]:
```
class Problem4Test(unittest.TestCase):

    def test_users_update_member(self):
        data = json.dumps({'first': 'testing'})
        r = requests.patch(BASE + '/0', headers = headers, data = data)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertIs(type(j), dict)
        self.assertEqual(j['id'], 0)
        self.assertEqual(j['first'], 'testing')

        # now retrieve the same object to ensure that it was really upda
ted
        r = requests.get(BASE + '/0', headers = headers, data = data)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertEqual(j['first'], 'testing')


    def test_users_update_member_not_found(self):
        data = json.dumps({'first': 'testing'})
        r = requests.patch(BASE + '/1000', headers = headers, data = dat
a)
        self.assertEqual(r.status_code, 404)



# Run the unit tests
unittest.main(defaultTest="Problem4Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_update_member (__main__.Problem4Test) ... ok
test_users_update_member_not_found (__main__.Problem4Test) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.052s

OK
```

Out[48]:
```
<unittest.main.TestProgram at 0x10d86dc70>
```

Problem 5: Deactivate a User

Deactivate a user with the route

```
POST /users/<id>/deactivate
```

This method will essentially toggle the `active` attribute for the user. Return `200` on success.

This problem shows how to implement non-REST commands.

In [60]:

```
class Problem5Test(unittest.TestCase):

    def test_users_deactivate_member(self):

        r = requests.post(BASE + '/0/deactivate', headers = headers)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertIs(type(j), dict)
        self.assertEqual(j['active'], False)


# Run the unit tests
unittest.main(defaultTest="Problem5Test", argv=['ignored', '-v'], exit=False)
```

```
test_users_deactivate_member (__main__.Problem5Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.020s

OK
```

Out[60]:

```
<unittest.main.TestProgram at 0x10d86d760>
```

```
In [ ]: import os
        from flask import Flask, request, Response, jsonify
        from functools import wraps


        # Custom error handler. Raise this exception
        # to return a status_code, message, and body
        class InvalidUsage(Exception):
            status_code = 400

            def __init__(self, message, status_code=None, payload=None):
                Exception.__init__(self)
                self.message = message
                if status_code is not None:
                    self.status_code = status_code
                self.payload = payload

            def to_dict(self):
                rv = dict(self.payload or ())
                rv['message'] = self.message
                return rv


        print(__name__)

        app = Flask(__name__)

        # set the default error handler
        @app.errorhandler(InvalidUsage)
        def handle_invalid_usage(error):
            response = jsonify(error.to_dict())
            response.status_code = error.status_code
            return response


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

        # dummy users
        USERS = [
            {'id': 0, 'first': 'Joe', 'last': 'Bloggs',
                'email': 'joe@bloggs.com', 'role': 'student', 'active': True},
            {'id': 1, 'first': 'Ben', 'last': 'Bitdiddle',
                'email': 'ben@cuny.edu', 'role': 'student', 'active': True},
            {'id': 2, 'first': 'Alissa P', 'last': 'Hacker',
                'email': 'missalissa@cuny.edu', 'role': 'professor', 'active': T
        rue},
        ]

        # Your code here...
        # E.g.,
        # @app.route("/users", methods=["GET"])

        # Problem 1

        @app.route("/users", methods=["GET"])
```

```python
def get_users():
    return jsonify(USERS)


# Problem 2
@app.route("/users/<int:post_id>", methods=["GET"])
def profile(post_id):
    for user in USERS:
        if post_id == user['id']:
            return jsonify(user)
    return Response(None,404)
# Problem 3
@app.route("/users", methods=["POST"])
def add_user():
    new_user = request.get_json()
    myDict = {}
    if 'first' in new_user.keys():
        myDict['first'] = new_user['first']
        myDict['last'] = new_user['last']
        myDict['email'] = new_user['email']

        #user role true
        #myDict['role'] = new_user['role']
        #myDict['status'] = new_user['status']

        #new user TRUE
        myDict['id'] = len(USERS)

        #response
        USERS.append(myDict)
        response = Response(None, 201)

        return response

    else:
        response = Response(None, 422)
        return response

# Problem 4
@app.route("/users/<int:post_id>", methods=["PUT","PATCH"])
def update_user(post_id):
    for dictionary in USERS:
        if post_id == dictionary['id']:
            user = request.get_json()
            dictionary['first'] = user['first']
            response = Response(None, 200)
            return jsonify(dictionary)

    response = Response(None, 404)
    return response

# Problem 5

@app.route("/users/<int:post_id>/deactivate", methods=["POST"])
def deactivate_user(post_id):
    for dictionary in USERS:
        if post_id == dictionary['id']:
            dictionary['active']=False
            response = Response(None, 200)
            return jsonify(dictionary)
    response = Response(None, 404)
    return response
```