Programmable Web Project

Exercise 3
Implementing REST APIs in Python
Learning outcomes (I)

• Students understand what a Web API is and learn different Web API architectures.
• Students understand the concept of hypermedia and how it can be used to build Web APIs.
• Students are able to design and implement a Web API following REST architectural style principles using existing web frameworks.
Learning outcomes (II)

• Students are able to write unit and functional tests to inspect their APIs.
• Students are able to document their Web APIs using adequate software tools.
• Students are able to implement simple software applications that make use of the APIs.
Flask-REST

• Library that applies the ROA principles to Flask framework

• Main characteristics
  – Substitutes *view function calls* with *object calls*
    • Implementing classes which extends `flask_restful.Resource`
    • Its main functionality encapsulated in the `flask_restful.Api`
Flask REST framework

• Skeleton

class Message (Resource):

#DELETE
def delete(self,*uritemplatevariables):
    pass

#GET
def get(self,*uritemplatevariables):
    pass

#PUT
def put (self,*uritemplatevariables):
    pass

#POST
def post(self,*uritemplatevariables):
    pass

Methods:
Each one maps to one HTTP method
Flask REST routing(I)

• We need to create an `Api` object which will enclose all Flask-Restful functionality

```python
from flask import Flask
from flask_restful import Api

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

• All new endpoints registered using the `Api.add_resource` method:

```python
api.add_resource(Message, '/forum/api/message/<messageid>')</n```
Flask REST routing (II)

• The method receives extra arguments if the associated uri template contains template variables

```python
api.add_resource(Message,
                 '/forum/api/message/<messageid>/')
```

```python
class Message (Resource):
    def get(self, message_id):
        pass
```
Flask REST routing (III)

• Reverse routing:
  • Mechanism to get dynamically the url associated to a resource.
  • HATEOAS!!! -> Never hardcode the URLs in the response!!!

```
resource_url = api.url_for(resource_class, **urltemplatevariables)
```

• Example:
  • Resource Message; url = /forum/api/messages/<msg-id>
  • If we want to get the URL associated to a Message with id msg-1

```
api.url_for(Message, messageid = 'msg-1')
```

• Generates the url: /forum/api/messages/msg-1
Flask-REST framework.
Request object

`flask.request` is a globally accessible variable which contains HTTP request information.

<table>
<thead>
<tr>
<th>ATTRIBUTE / METHOD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>.data</td>
<td>A string containing the request body</td>
</tr>
<tr>
<td>.args</td>
<td>A dictionary with the URL query parameters</td>
</tr>
<tr>
<td>.headers</td>
<td>A dictionary with HTTP request headers</td>
</tr>
<tr>
<td>.json</td>
<td>Parses the incoming JSON request into python dictionary.</td>
</tr>
</tbody>
</table>
Flask-REST framework. Response object

- Each one of the HTTP methods return a tuple that is transformed into a response object.

```python
return data, status_code, headers
```

- Data: String or Python dictionary.
- The status code as int
- A dictionary the keys are the headers names and values are the header values

- You can return also use the `Response` object:

```python
return Response (data=None, status=None, headers=None, mimetype=None)
```

- Flask tries to serialize response in right format (`Accept header`).
  - Does not support hypermedia type => use `json.dumps()`
Hooks

• Execute code before or after a request is processed
  – E.g. Useful to configure database connections.
  – Implemented as decorator:

```python
@app.before_first_request
@app.before_request
@app.after_request
@app.teardown_request
```

@app.teardown_request. After a request is processed or exception occurred.
• When Flask receives a request, it needs to make a few objects available to the view function.
  - Flask use context: temporally makes objects globally accessible

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_app</td>
<td>Application context</td>
<td>The application instance for the active application.</td>
</tr>
<tr>
<td>g</td>
<td>Application context</td>
<td>An object that the application can use for temporary storage during the handling of a request. This variable is reset with each request.</td>
</tr>
<tr>
<td>request</td>
<td>Request context</td>
<td>The request object, which encapsulates the contents of a HTTP request sent by the client.</td>
</tr>
<tr>
<td>session</td>
<td>Request context</td>
<td>The user session, a dictionary that the application can use to store values that are “remembered” between requests.</td>
</tr>
</tbody>
</table>

Source: Flask Web Development. Miguel Grinberg. O’Reilly
Flask-Restul framework cycle (I)

1. **Fill the request object with the HTTP request data:**
   - Store the headers in `request.headers`
   - Stores the query parameters of the URL in `request.args`
   - Stores the data in `request.stream` or `request.data` (a string)
   - The request body is accessible using the `json` if the HTTP header Content-Type is application/json

2. **Consults the routing mechanism to determine the Resource class which must handle the request.**
Flask-Restulf framework cycle (II)

3. **Execute pre request hooks**

4. **Calls the view method with the same name of the HTTP method.**
   - The method receives keyword arguments with the values of URI template variables.

5. **The application access the request data and process it**
   - The app accesses the database and extracts/modify necessary information.
   - The app generates a python dictionary to store the response data.
6. The app method returns the dictionary containing the data, the status code and the dictionary with the response headers
   - Flask will serialize this dictionary using the adequate resource representation depending on the MIME type contained in the Accept header.

7. Execute post request hooks

8. The HTTP response is returned to the client
Generating hypermedia

- If you find a parser / serializer library for your hypermedia format use it.
  - Working with strings is **insane!!!!!!!!!**

- Otherwise, implement it (e.g):
  - `MasonBuilder(dict)` -> Extend dictionary class to support structure and Mason vocabulary
  - `InventoryBuilder(MasonBuilder)` -> API Resource specific subclass -> Uses application semantics
Schemas

- Mason uses **JSON Schema** schemas to define the structure of the HTTP requests of **PUT** and **POST** methods.
  
  - Define the schema as dictionary

```python
def sensor_schema():
    schema = {
        "type": "object",
        "required": ["name", "model"]
    }
    props = schema["properties"] = {}
    props["name"] = {
        "description": "Sensor's unique name",
        "type": "string"
    }
    props["model"] = {
        "description": "Name of the sensor's model",
        "type": "string"
    }
    return schema
```

- Use `jsonschema.validate` to check if the received requests is valid
Returning Profiles and Link relations

- The profile and link relation documentation should be accessible by the REST clients.

1. If the documentation is stored in your server
   - Store the files in a folder named `static`
   - Use the `flask.send_from_directory`

```python
from flask import send_from_directory

app = Flask(__name__, static_folder="static")

@app.route("/profiles/<resource>/")
def send_profile_html(resource):
    return send_from_directory(app.static_folder, "{}.html".format(resource))
```

2. If the documentation is stored in external server
   - Use the `flask.redirect`

```python
from flask import redirect

@app.route("/profiles/<resource>/")
def send_profile_html(resource):
    return redirect(completeurltotheprofile)
```
Functional testing in Flask (I)

- Flask provides an HTTP test client

```python
from app import app, db

client = app.test_client()

# Sending HTTP requests
resp = client.get(url)
resp = client.put(url, data="string_data", headers={})
resp = client.post(url, data="string_data", headers={})
resp = client.delete(url)

# Accessing the response
body = resp.data
status = resp.status_code
header = resp.headers # Dictionary of headers
```
Functional testing in Flask (II)

• Use pytest

```python
# based on http://flask.pocoo.org/docs/1.0/testing/
@pytest.fixture
def client():
    db_fd, db_fname = tempfile.mkstemp()
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///{}' + db_fname
    app.config['TESTING'] = True
    db.create_all()
    _populate_db()
    yield app.test_client()
    db.session.remove()
    os.close(db_fd)
    os.close(db_fd)
    os.unlink(db_fname)
```

• Use test cases as you did for the database
  – Remember that the Flask HTTP client is passed as argument to the test!!

```python
def test_post_valid_request(self, client):
    valid = _get_sensor_json()
    resp = client.post(self.RESOURCE_URL, json=valid)
    assert resp.status_code == 201
```
Functional testing in Flask (II)

What to test?
- Resources are created, modified and removed correctly
- Structure of the responses are correct
- Headers and status code are correct
- Your API response correctly to incorrect Request (wrong url, wrong format, wrong headers...)
- In hypermedia, check that the controls are correct

Coverage
- Use coverage plugin to check how many lines of your code has been testing.
- Aim for high coverage (>95%)
In the exercise everything in one single file, BUT ...

Recommended structure for your project:

```bash
/your/project/root/
├── MANIFEST.in
├── README.md
├── setup.py
├── sensorhub
 │   ├── __init__.py
 │   ├── api.py
 │   ├── models.py
 │   ├── utils.py
 │   └── resources/
 │       ├── __init__.py
 │       │   └── deployment.py
 │       │   └── location.py
 │       │   └── measurement.py
 │       │       └── sensor.py
 │       └── static/
 │           └── schema/
└── tests
    ├── api_test.py
    ├── db_test.py
    └── utils.py
```

Other recommendations:

- Application factory: separate development, production, testing
- Configuration to `config.py`
- Use blueprints to separate different APIs / different APIs functionalities:
  - e.g. Admin vs User
- Make your project installable
- Use command line interface for administrative tasks: e.g. create / populate the database.

SEE THE FLASK API PROJECT LAYOUT IN LOVELACE
Unit testing

- To access the request, session attributes or url_for function you must be inside the request context.
- Two ways of working with the request context while performing unit-testing
  - with test_client:
    ```python
    with resources.app.test_client() as client:
        resp = client.get(url)
        links = resp.data['links']
        self.assertEquals(links[0]['href'], api.url_for(Messages))
    ```
  - with test_request_context
    ```python
    with resources.app.test_request_context(self.url):
        request.data = mydata
        app.preprocess_request()
        ....
        resp = Response(...)
        resp = app.process_response
    ```
- Much more info at
  http://flask.pocoo.org/docs/testing/?highlight=test_request_context