ECS781P: Cloud Computing
Lab Instructions for Week 7

Using REST APIs, Simple Caching

Dr. Arman Khouzani, Dr. Felix Cuadrado

Feb 19, 2019

In the previous lab session, we designed some simple REST API's of our own. Here, we will learn how to use REST services of others in an example application.

Overview

Using a REST service from an application is as simple as:

- constructing the correct HTTP request parameters (constructing the url, setting request headers)
- send the HTTP request (GET, POST, etc);
- receiving and parsing the response.

A python module that facilitates sending HTTP requests is called `requests`. In its simplest form, to send a get request is as simple as invoking `requests.get(url)`, to send a post request is `requests.post(url, data={'key':'value'})`, and so on. Take a quick look at its documentation: Requests: HTTP for Humans.

Preparation

The preparation is as in the last two weeks: we are still working on the ITL machine directly to develop and test our application:

1. open a `terminal` and change into the directory of our app form last weeks: e.g. `cd ~/ECS781P/week5/restful_app`.
2. Activate our virtual environment:

   ```
   source flask_venv/bin/activate
   ```

   where `flask_venv` should be the name of your own python3 environment.

3. As for last weeks, we can use `Atom` as our text editor for convenience. Just type in `atom`. Then using `Add Project Folder`, add the directory of our app (e.g. with my naming so far: `~/ECS781P/week5/restful_app`).

4. Edit the `requirements.txt` file by adding the following entries (these are the names of modules we will be using today):

```
requests
requests-cache
plotly
```

> Note that although we will import `json` module as well, we don't need to `pip install` it, as it is a *built-in module* of python. The `requests` module is for sending our HTTP requests to REST APIs from our application. `requests-cache` is a "transparent persistent cache" for the `requests` module (documentation: here). `plotly` is a graphing library that "makes interactive, publication-quality graphs online" (documentation: here).

5. From the terminal, install these new modules by running:

```
pip install --upgrade -r requirements.txt
```

6. If you encounter any errors, ask for assistance.

Getting Familiar With the API

One of the early steps in using an API is to get familiar with its input and output format and its authentication method (if any). For this lab, we have chosen a free public API with no authentication requirement: one of the UK Police APIs: https://data.police.uk/. In particular, an API that return past "street-level" crime incidents: https://data.police.uk/docs/method/crime-street/.
Let's investigate it using python's `requests` module. Create a `test.py` as follows:

```
import requests
from pprint import pprint

crime_url_template =
 'https://data.police.uk/api/crimes-street/all-crime?lat={lat}&lng={lng}&date={data}'

my_latitude = '51.52369'
my_longitude = '-0.0395857'
my_date = '2018-11'

crime_url = crime_url_template.format(lat = my_latitude,
        lng = my_longitude,
        data = my_date)

resp = requests.get(crime_url)
if resp.ok:
        crimes = resp.json()
else:
        print(resp.reason)

pprint(crimes)
```

Now run it from the terminal (`python test.py`) to see if it works. Incidentally, these are the latitude and longitude of QMUL library!

> Note: when you copy-paste the code, makes sure all characters are proper: in the older version of this document, the hyphens (dashes) were replaced with a wrong character: – make sure you fix them to: - (this should no longer be the case though, but if you notice it still happening, note that there are *four* of them: two in the `crime_url_template`, one in `my_logitude`, and one in `my_date`. Also, make sure there are no spaces in the URL (in general, space characters are not allowed directly in a URL).

Here, the variable which we called `crimes` holds a JSON-formatted response to our GET request. We can "use" it easily in python. For instance, let's say we are interested to find out the statistics of crimes per each category of crime. Here is my attempt (For convenience, I am using another API call to get the categories!):

```
categories_url_template = 'https://data.police.uk/api/crime-categories?date={date}'

resp = requests.get(categories_url_template.format(date = my_date))
if resp.ok:
        categories_json = resp.json()
else:
        print(resp.reasone)

categories = {categ["url"]:categ["name"] for categ in categories_json}

crime_category_stats = dict.fromkeys(categories.keys(), 0)
crime_category_stats.pop("all-crime")


for crime in crimes:
        crime_category_stats[crime["category"]] += 1

pprint(crime_category_stats)
```

Test this out as well! As the next exercise, say we are interested in the statistics of the outcome of crime incidents. Here is a sample code that does the job:

```
crime_outcome_stats = {'None': 0}
for crime in crimes:
        outcome = crime["outcome_status"]
        if not outcome:
                crime_outcome_stats['None'] += 1
        elif outcome['category'] not in crime_outcome_stats.keys():
                crime_outcome_stats.update({outcome['category']:1})
        else:
                crime_outcome_stats[outcome['category']] += 1

print(crime_outcome_stats)
```

These should be self-explanatory. Try this code as well!

Putting it in a web app

So far, we just printed our results for ourself. What if we want to make this service available over the network: we make a web application. Previously, we saw how to create a RESTful

API. The intended user of an API is an application/program. Here, we create an interface that is designed for human consumption directly: a webpage! In particular, we will show these results using pie-charts.

1. From QM+, download the file `week7_app.py` and the folder templates into the same directory of our app.
2. Note how the parameters are passed to our uri. The pattern of our uri is `/crimestat/?lat=&lng=&date=`.
3. Replace the parts for `compute crime_category_stats` and `compute crime_outcome_stats` from above.
4. Save the code and run the app! `python week7_app.py`.

Caching

One of the problems of our app so far is that each time it sends a new REST request to the police API, even if it is the same request. To counter this, we can "cache" the previous requests on our server instead. In python, it is as simple as adding the following:

```
import  requests_cache

requests_cache.install_cache('crime_api_cache', backend='sqlite', expire_after=36000)
```

So you new head of your python file should look like the following:

```
from flask import Flask, render_template, request, jsonify
import plotly.graph_objs as go
from plotly.utils import PlotlyJSONEncoder
import json
import requests
import  requests_cache

requests_cache.install_cache('crime_api_cache', backend='sqlite', expire_after=36000)

app = Flask(__name__)
...
```

1. what is the role of `'crime_api_cache'`?
2. what is the role of `backend='sqlite'`? what are some other options?
3. what does `expire_after=36000` mean? is this a good policy for our case?
4. Run your app again and try loading the webpage with old and new inputs, do you see the difference in loading time? What do you attribute this to?
5. what new file is created in your directory?