ECS781P: Cloud Computing
Lab Instructions for Week 8

Configuring Apps, Git

Dr. Arman Khouzani, Dr. Felix Cuadrado

Feb 26, 2019

In the previous lab session, we used a publicly available REST API which did not require any authentication/authorization. In this lab, we start with a simple example that requires authentication using a direct API-Key, which is the simplest version of authentication. We then see how to not hard-code sensitive information in our code. And finally, the basics of `git`.

Overview

Authentication vs Authorisation

Two closely related but different concepts are authentication and authorisation. Let's disentangle them as it can be a source of confusion later on:

Authentication is ascertaining the identity of an entity;

Authorisation is about establishing access rights (a.k.a. privileges/permissions/capabilities) that is, which actions can be done on what resources by the entity.

To get a clearer idea of why they are different: suppose your friend gives you a ticket to a theatre show. Then you have a means of authorisation to enter the theatre during a certain time and view the show. The ticket is a proof of authorisation, and not authentication.
The concept is rather muddled in everyday life because it is rare to have a pure means of authentication: usually, a proof of identity also provides a proof of rights/privileges, and vice versa. For instance, a driving license, a residence permit, a passport, etc. Another reason for potential confusion is that typically, the authorisation is implicitly or explicitly predicated on authentication: a driving licenses is a proof of permission for driving only if the holder is authenticated. But as in the example of theatre tickets, authorisation can be without authentication too.

Typical Authentication Methods in APIs

HTTP Basic Authentication

A user simply provides a username and password directly in the HTTP header request: no need to handshakes, challenge/response, etc.

The problem is that, unless the process is strictly enforced throughout the entire data cycle to SSL/TLS for security, the authentication is transmitted in open (in the clear) on insecure lines.

API Keys

A uniquely generated value is assigned to each first time user, signifying that the user is known. When the user attempts to re-enter the system, their unique key is used to prove that they are the same known user.

OAuth

OAuth combines Authentication and Authorization to allow more sophisticated *scope* and *validity control*.
Detail of different authentication and authorisation methods is beyond the scope of this module. A quick useful read with practical examples for APIs can be found here:
https://idratherbewriting.com/learnapidoc/docapis_more_about_authorization.html.

1 Example API with simple API-key authentication

Almost any interesting API requires authentication. This is primarily so that they can monetize their service! As a random example, I have chosen an "air quality" REST API for this example:

1. Go to this URL and click on *Sign Up*. After entering your information, you will receive an activation email. After which, you should be able to log in to the developer dashboard page of it: https://developers.breezometer.com/dashboard/.
2. Your API key is there! We will use this API key to send requests in a flask app next.
3. Before that, read the documentation of the API here:
   https://docs.breezometer.com/api-documentation/air-quality-api/v2/#requesting-hourly-history
4. As we know by now: open a `terminal` and change into the directory of our app form last weeks: e.g.
   `cd ~/ECS781P/week5/restful_app`.
5. Activate our virtual environment:
   ```
   source flask_venv/bin/activate
   ```
   where `flask_venv` should be the name of your own python3 environment.
6. Type in `atom`, and add the directory of our app (e.g. with my naming so far:
   `~/ECS781P/week5/restful_app`).
7. Write a flask app (or extend the app from last week, with the following content):

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

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

```
app = Flask(__name__)


air_url_template = 'https://api.breezometer.com/air-quality/v2/historical/
        hourly?lat={lat}&lon={lng}
        &key={API_KEY}
        &start_datetime={start}
        &end_datetime={end}'


MY_API_KEY = 'something!'

@app.route('/airqualitychart',  methods=['GET'])
def airchart():
        my_latitude = request.args.get('lat','51.52369')
        my_longitude = request.args.get('lng','-0.0395857')
        my_start = request.args.get('start','2019-01-27T07:00:00Z')
        my_end = request.args.get('end','2019-01-29T07:00:00Z')
        air_url = air_url_template.format(lat=my_latitude, lng=my_longitude,
                        API_KEY=MY_API_KEY, start=my_start, end=my_end)

        resp = requests.get(air_url)
        if resp.OK:
                resp = requests.get(air_url)
                pprint(resp.json())
        else:
                print(resp.reason)

        return ("Done!")

if __name__=="__main__":
        app.run(port=8080, debug=True)
```

8. Run the app and see if it works (Note: you need to visit `/airqualitychart` url, and you should check what is printed in the terminal!)
9. Fix any issues!

2 Avoiding Exposing Credentials in the Code

As we noticed, we had to pass the correct API-key to get a response. But you should recall that this is bad practice. But how do we set these sensitive values? The "solution" is to assign these variables in a `config` file, which is separate from the app files and load such variables from it. Flask allows multiple `config` files. The convention is that insensitive parameters are put in a `config.py` and put the sensitive information in `instance/config.py`. Later, we should exclude this folder from being shared, e.g. in a repository.
Another problem is that we have been running our app with `DEBUG=True`, which is disastrous in terms of security (find out why!).

1. create a `config.py` file in the same directory as your application file, and add:

```
DEBUG = False
```

2. Now, create a subdirectory `/instance` in your app directory, and create another `config.py` file there, with the following content:

```
DEBUG = True
MY_API_KEY = "your_api_key_here"
```

So now your file directory should look like the following:

```
flask_env/
requirements.txt
myapp.py
config.py
instance/
        config.py
```

3. Now, edit your application file by replacing the `app = Flask(__name__)` with the following lines:

```
app = Flask(__name__, instance_relative_config=True)
app.config.from_object('config')
app.config.from_pyfile('config.py')
```

Note that we have written it in this order so that the variables from `instance/config.py` overrides their values in `config.py`.

4. Now, in order to access any of the parameters set in the `config` files, we can use `app.config['PARAMETER_NAME']`. For instance, in our case, to access the API key, we can use `app.config['MY_API_KEY']`. So modify the main code to use the API key in this manner (you should replace `API_KEY=MY_API_KEY` with `API_KEY=app.config['MY_API_KEY']`, and of course, remove `MY_API_KEY = '...'`!)

3 Getting Started with Git

What is Git?

Git is a fast distributed revision control system (free and open-source). "It is primarily used for source code management in software development, but it can be used to keep track of changes in any set of files. Every Git directory on every computer is a full-fledged repository with complete history and full version tracking abilities, independent of network access or a central server."[1]
Git is an instance of a tool for **Software Configuration Management (SCM**: If something goes wrong, SCM can determine what was changed and who changed it. If a configuration is working well, SCM can determine how to replicate it across many hosts.[2]

Essential Glossary (Speaking "Git")

Repository A collection of files (potentially organized in folders) along with their "history" of creation and edition.

Commits Git stores the history as a compressed collection of interrelated snapshots of the project's contents. In Git each such version is called a **commit**.

Branches The commits are not necessarily all arranged in a single line from oldest to newest; instead, work may simultaneously proceed along parallel lines of development, called **branches**, which may merge and diverge.

[1] https://en.wikipedia.org/wiki/Git
[2] https://en.wikipedia.org/wiki/Software_configuration_management

Heads (Branch heads) A single Git repository can track development on multiple branches. It does this by keeping a list of **heads** which reference the latest commit on each branch.

Master branch A freshly cloned repository contains a single branch head, by default named "**master**", with the working directory initialized to the state of the project referred to by that branch head.

Tags Like heads, **tags**, are also references into the project's history. Tags are expected to always point at the same version of a project, while heads are expected to advance as development progresses.

Setting up with Github

We can have version control completely on a local machine. But in order to easily share your project, collaborate on projects, automatic deployment of your apps and "continuous integration", and many other good reasons, we would like to take advantage of a version control hosting service. The most well-known one is `GitHub`, which we will use in our exercise. It is good to have a public github repository as part of your resume!

1. If you don't have a `github` account, create one, by visiting https://github.com/.
2. After you log in, click on `New` to create a new (empty) repository.
3. Now in your terminal, in the same directory of your app, type in (replacing with your own info!)

```
git init
git config user.name "Arman"
git config user.email arman.khouzani@qmul.ac.uk
```

4. The first file we need to add is a hidden file called `.gitignore`. This file tells git which files/folders to not add for tracking (to ignore). So create a `.gitignore` file (don't forget the starting dot!) with the following content:

```
# your virtual env folder:
flask_venv/

# the instance/ folder containing our secret stuff!
instance/

# anything else you want git to ignore, add here:
```

5. Now we are ready to stage all our files for commitment:

```
git add .
git commit -m "first commit!"
```

6. You can check the status of our staging and commitment using:

```
git status
```

7. Next, let's add a `README.md` (a markdown) file, explaining our app. Create a `README.md` file with the following content:

```
# example REST API with authentication
A Flask app using air quality API that needs authentication!
```

8. This new file is not added to the "index", i.e., it is not tacked. To see this:

   ```
   git status
   ```

   So we need to tell git to track it:

   ```
   git add README.md
   ```

9. We are now ready to commit to this new change:

   ```
   git commit -m "added readme.md file"
   ```

10. To see the history of our commits and the specific changes, you can issue commands like:

    ```
    git log --graph
    git diff README.md
    ```

11. Finally, to `push` our repository:

    ```
    git remote add origin URL_OF_YOUR_NEWLY_CREATED_EMPTY_REPOSITORY_ON_GITHUB
    git push origin master
    ```