ECS781P: Cloud Computing
Lab Instructions for Week 5

RESTful application with Flask

Dr. Arman Khouzani, Dr. Felix Cuadrado

Feb 05, 2019

Overview

In last week's lecture, we got introduced to the concept of REST architecture. In this lab, we will be developing a toy example of a RESTful application using flask. The emphasis is on the application itself, and the cloud deployment is left as an optional step. However, keep in mind that we need to create this RESTful application interfaces because our services will be eventually hosted in the cloud and our client applications need to communicate with them in order to use their service.

Preparation

We will be using the ITL machine to locally test our app. Alternatively, you can use the google shell VM as our local machine, as we did last week. We can use ITL machine because we do not need a root privilege to develop and test our app on the "localhost". To install our own python modules in an isolated environment, we will create a python "virtual environment". When we create a virtual environment, a clean-sheet copy of the python interpreter with just the essential libraries including the python package manager `pip` gets copied in a local directory. After "activating" it, our application will only see this local python and its modules. So there will be no ambiguity about package dependencies: if it needs a module, it will prompt us and will not run. We can install any packages in this local python (virtual environment), using `pip`.
However, these new modules take up some space that may push us over our 5G disk quota! So please take the following steps, to ensure you have enough space. If you are going to use the google shell as your "local machine", you can skip this section!

1. On your host machine (ITL), open a `terminal`.
2. Go to your home directory: `cd ~`

3. Optionally, for convenience, you can change your default Linux prompt to something more informative by executing `export PS1="\u@\h:\w\$ "` to change your shell prompt to show "user name" "host machine": "working directory" and the $ sign and an extra space! (Or to have the current time as well and only the name of the current folder: `export PS1="[\@] \u@\h:\W\$ "`, you can even make it colourful as well, but let's not get too distracted!)

4. From your home directory, run the `du` command for estimating file space usage:

   ```
   du -sch .[!.]* * | sort -h
   ```

   Note that it may take up to a minute to return the results. This command lists all of your files and directories in your current directory (including hidden files and subdirectories, which in Linux are designated by a starting "dot") with their "size" (sorted from smallest to the largest one), ending with a `total` size.

5. Identify the folders that you do not need and you can safely delete: Some examples (at least on my machine) included: `Downloads, Music, Videos, Documents, Pictures, Images, .cache, .thumbnails`. A hidden directory that you can almost certainly safely remove is `.local/share/Trash`: this is your "trash" directory (equivalent of "Recycle-Bin" concept in Windows). This is where files and directories are sent to when they are "deleted" using the file system GUI (Nautilus), and they continue to take up space! Also, can you guess what `.cache` and `.thumbnails` hold?

   I myself deleted my entire `.cache` and `.local` directories! In order to remove a directory, the command to use is `rm -rf <directory>`. For instance, to "empty" our trash:

   ```
   rm -rf .local/share/Trash/*
   ```

   Do this for other safe folders to remove (for instance, I had a 2.5GB `Downloads` directory, and a 1.0GB `.cache` directory – phew!)

   > Note that `rm -rf` can be quite dangerous: `-r` means "recursively", so the entire subdirectory will be irreversibly removed, also `-f` means "forcefully", so you will not be prompted for any warning — use cautiously!

6. Another directory that may have grown too large is the `~/Images`. You may recall from week 2 that these are the image differences of your VMs. Inspect them and delete the ones you do not need.

Setting things up (including the python virtual env)

We will use `flask`, a python's micro-framework. Again: all of the steps in this section can be performed inside your ITL machine, no VMs necessary!

1. Open a terminal. Create a directory for our app somewhere appropriate. For example:

   ```
   mkdir -p ~/ECS781P/week5/restful_app
   cd ~/ECS781P/week5/restful_app
   ```

2. Create our one-file python application, along with a `requirements.txt` file (both are empty for now):

   ```
   touch myapp.py
   touch requirements.txt
   ```

3. Let us now create our *virtual environment* with your preferred name (I chose `flask_venv`) and "activate" it:

   ```
   python3 -m venv flask_venv
   source flask_venv/bin/activate
   ```

   Notice the change in the (beginning of the) prompt that reflects this activation. At then end, you can deactivate this virtual environment by issuing `deactivate`, but don't do it now! You can ensure that your environment is indeed activated properly by issuing: `which python`. It should show the python of your virtual environment. Also to make sure it is python 3, issue `python --version`. It should not say `python 2.x.y`, e.g. `2.7.5`, but rather `python 3.x.y`, e.g. `python 3.4.8`.

4. For writing our code, let us use `Atom` for more convenience. Just type in `atom`.

5. Right-click on the left hand column, and choose the `Add Project Folder`, similar to Fig. 1. If there is no left column, then choose `File > Open Folder ...` option from the menu. Either way, then choose our app directory (directory, and not the file!), which in our example, it is `~/ECS781P/week5/restful_app`. Then you should see that folder added on the left column, with click-able files and sub-directories!

6. In `atom`, click on `myapp.py` file and edit its content to the following: (this is essentially the same hello-world app from last session!) – don't forget to save: `Ctrl+S` in `atom`.

   ```
   from flask import Flask

   app = Flask(__name__)

   @app.route('/')
   def hello():
           return('<h1>Hello World!</h1>')

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

   When developing python code, keep the following in mind:

   - The indentations (the left spaces before a line starts) matter.
   - `__name__` and `__main__` each have "two" underscore characters before and after!
   - In the conditional statement `if __name__ == '__main__':`, there are two equality signs, while in the assignment statement `app = Flask(__name__)`, there is only a single equality sign.

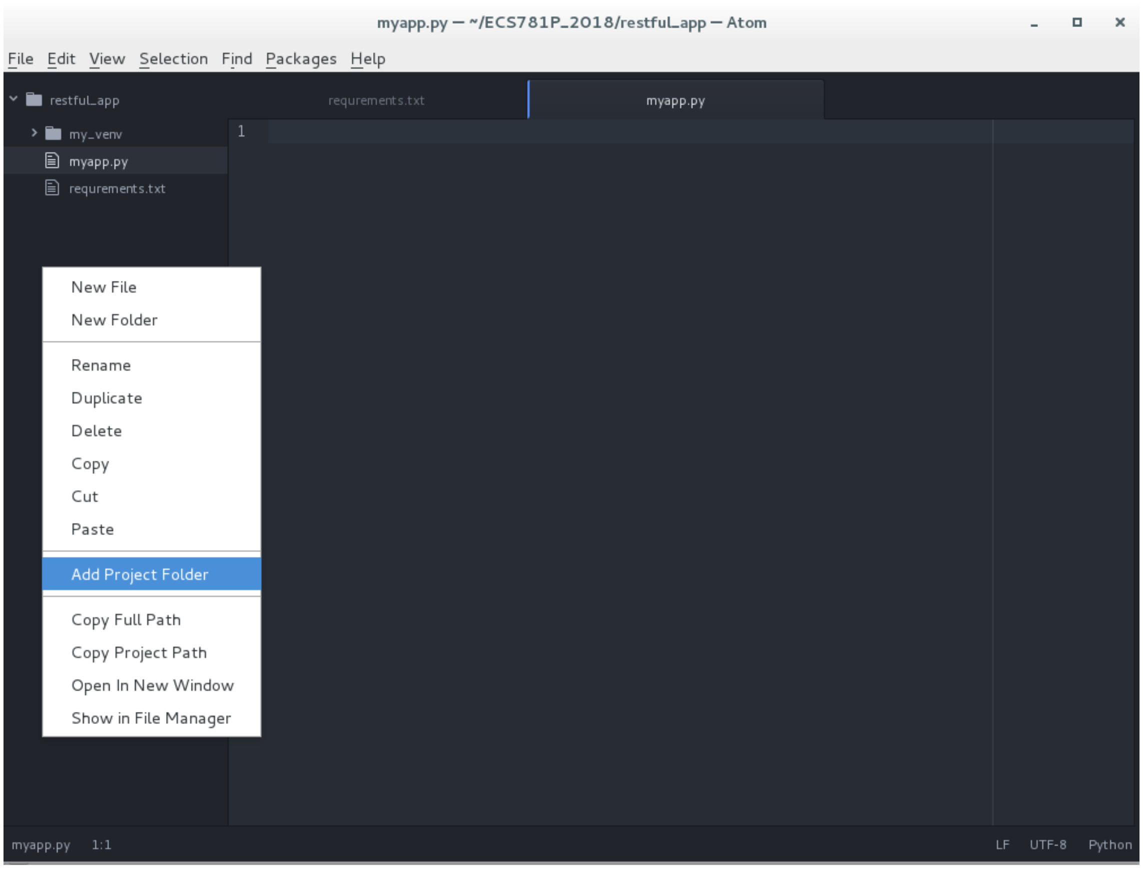

Figure 1: Step 3 of Section .

- Also, in the code, there are two "colon" signs (:), and no semi-colons (;)– Python has no semi-colons!
- Note that there should be no empty lines between `@app.route('/')` line and `def hello():` line. This is because technically, `@app.route('/')` is a "decorator" for the function `hello`, which flask interprets as the function that should be called when a request on that path arrives.

7. Next, open `requirements.txt` file and edit+save its content to the following (on separate lines). Note: this is still using `atom`. In fact, for the rest of today's lab, we are not going to close `atom`!

```
pip>=9.0.1
Flask==0.12.2
```

   The numbers specify the versions of that module (e.g. `pip>=9.0.1` needs we need a `pip` modules that are no older than the `9.0.1` version, etc.)

8. Now, switch to your `terminal`. From your app directory (the same directory that has your `requirements.txt`) run the following:

```
python -m pip install -U -r requirements.txt
```

   This command recursively (`-r`) installs or updates (`-U`) the "requirement" packages (modules) that we have specified one per each line in the provided file, i.e., `requirements.txt`, in our local python environment (so before you run this, make sure that your environment is activated: the beginning of your prompt sign should have the name of your environment in parentheses).

9. Finally, run the hello-world application:

   ```
   python myapp.py
   ```

   You should be able to open the app in your browser (in the prompted link). This is to make sure you have indeed set things up properly. If you see any errors, please seek assistance before moving on to the rest of the lab.

Creating our RESTful Services

We will create a RESTful app. For this week, we will only implement a `GET` method. Next week, we enrich it with other methods (`POST`, `PUT`, and `DELETE`).

GET Method

Recall that the `GET` method is for retrieving information. Our app simply returns the results of some simple queries. To keep things simple, we have avoided using a database. Instead, we are going to store our data directly inside a variable (in memory) and process the queries ourselves. This sample data is information about some singing bands and their albums. We have added to a sample python file that you should download from the QM+ (named: `myapp.py`). Overwrite your `myapp.py` with the provided file. So your code so far should look like the following:

```python
from flask import Flask

all_records =  [
 {
 "name" : "Radiohead",
 "albums" : [
          {
      "title":"The King of Limbs",
                        "songs":[
                                ...
                                ],
                                "description":"..."
                        },
                        {
                                "title":"OK Computer",
                                "songs":[],
                                "description":"..."
                        }
                ]
        },

        {
                "name":"Portishead",
                "albums":[
```

```
                        {
                                "title":"Dummy",
                                "songs":[],
                                "description":"..."
                        },

                        {
                                "title":"Third",
                                "songs":[],
                                "description":"..."
                        }
                ]
        }
]

app = Flask(__name__)

@app.route('/')
def hello():
        return "<h1>Hello, World!</h1>"

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

For our first RESTful service, let us provide our entire dataset to a GET request. First, we need to decide the URL that this resource can be accessed at. Recall that a URL follows the schema of

```
hostname[:portnumber]][/path][?query][#fragment]
```

where the brackets indicate optional components. The host-name and port-number is common between all components of an app, and indicates on which server it is running, and on which port number it is listening to. So we are deciding only on the latter parts: `[/path][?query][#fragment]`. In the simplest format, we can only have the `/path` component. For this lab, w stick with this simple structure. Next, week, we see how to use the `[?query]` to pass small arguments.
Although we can choose any arbitrary path-name for serving a RESTful request, it is good practice to follow easy to understand (and "discover") names and structures. For instance, it is highly recommended to follow a hierarchical (nested) structure, that is, a path line `A/B/` should mean that `B` is a subset (a segment, a refinement, a child, etc.) of resource `A`, etc. We will follow this hierarchical structure for this lab as well. So, let us call the path that returns all of the records simply `records`:

1. We should add this path to our app, and make sure it does the right thing: returns the requested resource. But we also need to make sure the returned format is correct. We will primarily use the JSON format. The required piece to be added to our code (`myapp.py`) is the following:

   ```
   @app.route('/records/', methods=['GET'])
   def get_records():
           return jsonify(all_records)
   ```

Explanation: we are specifying the “path” that the resource can be accessed at as `/records/`, and we are specifying the method of request to be `GET`. Then we say: if a GET request on the `hostname:portnumber/records/]` came through, serve it with the function we named `get_albums()`. What our function does is to return the (serialised) resource (which is all of our data) in th JSON format. To transform the data to a JSON object, we are using a method called `jsonify`, which we need to also import, as detailed next.

2. Add the above snippet to the `myapp.py`, so that your overall code should now look like the following:

```
from flask import Flask, jsonify

all_records = [ ...
]

app = Flask(__name__)

@app.route('/')
def hello():
        return "<h1>Hello, World!</h1>"

@app.route('/records/', methods=['GET'])
def get_records():
        return jsonify(all_records)

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

Note the following:

- The new part should be added somewhere between the `app = Flask(__name__)` part (because otherwise the object named `app` is not assigned yet) and somewhere before the `if __name__ == '__main__':..."` component, which essentially launches our app.
- Also, note the change in the first line:

  ```
  from flask import Flask, jsonify
  ```

  Since we use the method `jsonify`, we need to import it (which is part of the `flask` module).

3. Now, test your app to see if it works. Go to the terminal where you ran your app from (where we activated our virtual environment and we were in the same directory of our app). If the app is not running, then re-launch your app by issuing: `python myapp.py` again. You can check if our first RESTful service is working by sending a GET request to the url that we just added. Either:

- Open our browser and navigate to `localhost:5000/records/`
- From *another* terminal, issue the command:

  ```
  curl -v localhost:5000/records/
  ```

Either way, keep a hawkish eye on two output messages: one, of course, what is returned; two, what are the debugging information that your app is producing (which you can see from the terminal in which your app is running from).

OK, let's make our GET method a bit more exciting! Let us return only the names of the bands that are present in our dataset.

1. Place the following code snippet in the right location in your `myapp.py` code:

```
@app.route('/records/all_bands/', methods=['GET'])
def get_bands():
        response = [item['name'] for item in all_records]
        return jsonify(response)
```

2. Test it to see if it is indeed working as expected (following similar steps as before, except that this time, you would send a GET request (using your browser or `curl` from another terminal) to this url: `localhost:5000/records/all_bands/`

You can imagine that writing a specific function to serve each GET request in this manner can be a very inefficient process. Next, we show how to do this more efficiently, by passing the part of the url as an argument to the function. In the following example, we return all the albums of a specific band, but the name of the band is provided by the user, as part of the url:

```
@app.route('/records/albums_by_band/<bandname>/', methods=['GET'])
def get_album_by_band(bandname):
        response={bandname:'Not Found!'}
        for item in all_records:
                if item["name"]==bandname:
                        response = [x["title"] for x in item["albums"]]
                        break
        return jsonify(response)
```

Again, add this snippet to the right place in the code, and test it!

Extra Practice

Add a new GET url path like `/records/<bandname>/<album>` and returns the description of that `album` of that `bandname` accordingly.

Optional: Cloud Deployment

Once you are done with the lab, you can optionally containerise it and deploy it on GCP similar to what we did last week. Recall that you need to write an appropriate `Dockerfile` as well. Unfortunately, `docker` is not installed on ITL machines, but it is on google cloud shell! In order to upload your app folder to the google Cloud Shell machine, there is a drop-down menu next to the "web-preview" icon, with the first option being "Upload file". Of course you can also use `scp` (secure copy) or even better, through a git repository, but these are what we cover in later labs in detail.