

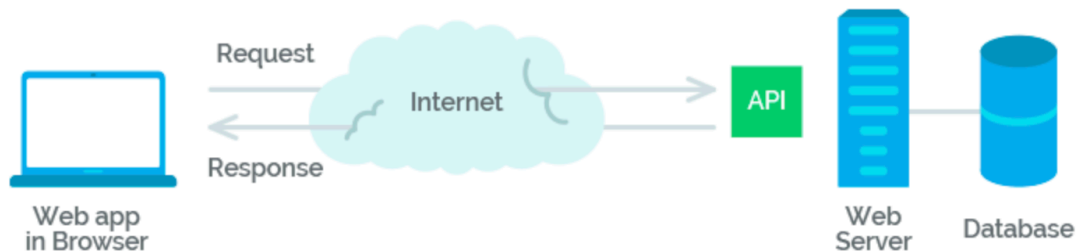
Lab #17
A Flask RESTful API

The lab is due on Thursday, April 18th at 11:59pm. It is worth 3 points. Submit your answers via blackboard.

Part 1: An Introduction to RESTful APIs

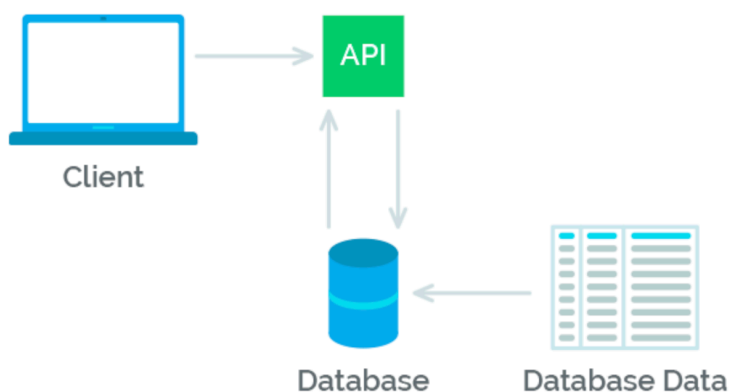
Credit given to <https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api>

An API (application programming interface) is a set of rules and mechanisms by which one application or component interacts with the others. It seems that the name speaks for itself, but let's get deeper into details. API can return data that you need for your application in a convenient format (e.g. JSON or XML).



REST. This abbreviation stands for representational state transfer. The definition can be conveyed with simpler words: data presentation for a client in the format that is convenient for it. There is one of the main points that you need to understand and remember: REST is not a standard or protocol, this is an approach to or architectural style for writing API.

REST API Design



A simple definition of RESTful API can easily explain the notion. REST is an architectural style, and RESTful is the interpretation of it. That is, if your back-end server has REST API and you make client-side requests (from a website/application) to this API, then your client is RESTful.

RESTful API best practices come down to four essential operations:

- receiving data in a convenient format;
- creating new data;
- updating data;
- deleting data.

REST relies heavily on HTTP.

- **HTTP (Hypertext Transfer Protocol)** is the primary means of communicating data on the web. HTTP implements a number of “methods,” which tell which direction data is moving and what should happen to it. The two most common are GET, which pulls data from a server, and POST, which pushes new data to a server.

Each operation uses its own HTTP method:

- GET - getting;
- POST - creation;
- PUT - update (modification);
- DELETE - removal.

All these methods (operations) are generally called CRUD ("create, read, update and delete")

Part 2: Our Goal


The goal for this lab is to create some functionality for a website to interact with the National Park database. We want to create means for a user to create new content, read (display) the data, update values, and delete values. While we are only going to be using HTTP requests for GET and POST (as these can be done in forms), this will provide a crude approximation at a RESTful implementation means for this data. If we were creating a true API for the data, we would provide a variety of methods and means for accessing the data (not necessarily just through web pages), but through other online tools using more HTTP functionality. For our purposes, however, this is still going to be awesome!

Part 3: Continue to Build off of Lab 16 – GET and POST

We want to create a way for a user to have create, read, update and delete operations on the database. The first thing to understand is how the current implementation from Lab 16 uses HTTP Get and Put requests.

The routes in Flask are, by default, performing HTTP GET requests. *A GET request pulls data from the server.* You can verify this by observing the output to the console when the URL requests are made by typing in a route via the web browser:

```
-----
127.0.0.1 - - [15/Apr/2019 22:50:42] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [15/Apr/2019 22:50:52] "GET /about HTTP/1.1" 200 -
■
```



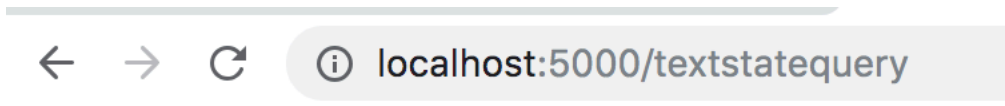
Notice the calls to GET

Consider the following two routes from Lab 16:

```
@app.route('/textstatequery')
def state_form():
    return render_template('textbox.html', fieldname="State")

@app.route('/textstatequery', methods=['POST'])
def state_form_post():
    state = request.form['text']
    return viewstates(state)
```

The first route here, is an implicit GET request. The result of this when the user enters the URL is to call the `render_template` form – calling the `textbox.html` in the `templates` directory



Enter the State

Recall that the `textbox.html` code is rather simple, but it contains a call to a POST form

```
1 <h3>Enter the {{fieldname}}</h3>
2 <form method="POST">
3     <input name="text">
4     <input type="submit">
5 </form>
6
```

The `<input name="text">` will cause a text box to be displayed. Whatever the user types in here will be returned with the identifier *text*

The `<input type="submit">` will cause a button to be displayed. When pressed, this will cause a POST operation to be returned, thus calling the second method for the URL of `/textstatequery`

```
@app.route('/textstatequery', methods=['POST'])
def state_form_post():
    state = request.form['text']
    return viewstates(state)
```

Notice how this takes the text from the textbox, and assigns it to a variable *state*. This is then passed to the function *viewstates*, and the results are displayed on the webpage.

An alternative to having two separate routes with the same name (accepting different GET or POST requests) is to put them in the same route and have the functionality determined by which kind of request is being made. For example, the year query could be written as one route with the following syntax:

```
@app.route('/year', methods=['GET', 'POST'])
def year_form():
    if request.method == 'POST':
        text = request.form['text']
        return viewyears(text)
    elif request.method == 'GET':
        return render_template('textbox.html', fieldname="Year")
```

Challenge #1:

Create a new .html template file that will allow for the user to enter both a year and a state. Write the code that will allow for query to be made so that only National Parks from the state specified that were founded in the year specified will be displayed. Call the route `/queryyearandstate`

Part 4: Creating New Parks!

First, a minor change should be added to the `execute_query` code in the `flaskapp.py` file. Add the line `get_db().commit()`

to allow the database to be properly updated (and not reloaded) before changes are made

```
def execute_query(query, args=()):
    cur = get_db().execute(query, args)
    get_db().commit()
    rows = cur.fetchall()
    cur.close()
    return rows
```

Recall that the C in CRUD stands for Create. Also recall that the SQL syntax for inserting new values into a database is the following

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

The National Park database was created with the following four fields

```
CREATE TABLE natlpark (name text, state text, year integer, area float)
```

Challenge #2:

Create a new .html template file that will allow for the user to enter all four fields. Execute the query to insert the new fields into the database. (You get to create your own national park – be creative!!) After inserting the values, display the updated table with the following call:

```
rows = execute_query("""INSERT INTO natlpark VALUES (?, ?, ?, ?)""", [newname, newstate, newyear, newarea])
rows = execute_query("""SELECT * FROM natlpark""")
return display_html(rows)
```

Call the route `/create`

Part 5: Deleting New Parks!

Let's also implement the D in CRUD by allowing a mistake to be erased – that is deleting a park from the database. Another html template should be used here to allow the user to enter the name of a park to be deleted.

The SQL syntax needed here is

```
DELETE FROM table_name WHERE condition;
```

Challenge #3:

Create a new .html template file that will allow for the user to enter the name field. Execute the query to delete the park that matches the name that was entered. As with Challenge #2, after the code was executed, display the entire table. Call the route /delete

Part 6: Updating Existing Parks

Lastly, let's implement the U in CRUD by allowing a park's area to be expanded or contracted. Another new html template will be necessary here, and the SQL syntax is:

```
UPDATE table_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;
```

Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

Challenge #4:

Create a new .html template file that will allow for the user to enter the name field and a new area. Execute the query to update the park that matches the name that was entered with a new area. As with Challenge #2, after the code was executed, display the entire table. Call the route /update

Part 7: Making it Live!

Follow the instructions to transfer your code to your EC2 instance. If your web server is still running (from Lab 11), this will use the same IP address.

The **Web Server Gateway Interface (WSGI)** is a specification for simple and universal interface between web servers and frameworks for the Python programming language. In short, WSGI is a set of tools that allows the web server to communicate with Python code.

First, ssh to your EC2 instance

```
[ec2-user ~]$ sudo yum -y update
```

Next, install flask and the WSGI libraries

```
[ec2-user ~]$ sudo yum install -y python-pip
[ec2-user ~]$ sudo yum install -y mod24_wsgi
[ec2-user ~]$ sudo yum install -y mod24_wsgi-python27.x86_64
[ec2-user ~]$ sudo pip install flask
[ec2-user ~]$ sudo pip install flask_table
```

Part 8: File Permissions

To allow ec2-user to manipulate files in this directory, you need to modify the ownership and permissions of the directory. There are many ways to accomplish this task; in this tutorial, you add a www group to your instance, and you give that group ownership of the `/var/www` directory and add write permissions for the group. Any members of that group will then be able to add, delete, and modify files for the web server.

Note that these commands in this section will appear cryptic, but the syntax isn't important for our purposes. This is some account set-up that will allow us to create files on our flask server easier (without always having to use SUDO).

1. Add the www group to your instance.

```
[ec2-user ~]$ sudo groupadd www
```

2. Add your user (in this case, ec2-user) to the www group.

```
[ec2-user ~]$ sudo usermod -a -G www ec2-user
```

3. You need to log out and log back in to pick up the new group. You can use the exit command, or close the terminal window.

```
[ec2-user ~]$ exit
```

4. Re-login (via your ssh command)

5. run the **groups** command to verify your membership in the www group. You should see the values of `ec2-user`, `wheel`, and `www` returned.

```
[ec2-user ~]$ groups
ec2-user wheel www
```

6. Change the group ownership of `/var/www` and its contents to the www group.

```
[ec2-user ~]$ sudo chown -R root:www /var/www
```

7. Change the directory permissions of `/var/www` and its subdirectories to add group write permissions and to set the group ID on future subdirectories.

```
[ec2-user ~]$ sudo chmod 2775 /var/www
[ec2-user ~]$ find /var/www -type d -exec sudo chmod 2775 {} \;
```

8. Recursively change the file permissions of `/var/www` and its subdirectories to add group write permissions.

```
[ec2-user ~]$ find /var/www -type f -exec sudo chmod 0664 {} \;
```

Now `ec2-user` (and any future members of the `www` group) can add, delete, and edit files in the Apache document root. Now you are ready to add content, such as a static website or a PHP application.

Part 9. Creating a Flask App on EC2

1. Create a directory called `flaskapp` to hold our files by using the `mkdir` (make directory) command

```
[ec2-user ~]$ mkdir /var/www/html/flaskapp
```

2. Change to this directory

```
[ec2-user ~]$ cd /var/www/html/flaskapp
```

3. Recall that WSGI is **Web Server Gateway Interface** that is the library of functions that provides a way for Python code to interact with our web server. Execute the following four instructions that will place appropriate files to enable WSGI for our server:

```
[ec2-user ~]$ cd /etc/httpd/conf/
[ec2-user ~]$ sudo rm httpd.conf
[ec2-user ~]$ sudo wget http://analytics.drake.edu/~urness/CS178/flaskapp/httpd.conf

[ec2-user ~]$ cd /var/www/html/flaskapp/
[ec2-user ~]$ wget http://analytics.drake.edu/~urness/CS178/flaskapp/flaskapp.wsgi
```

If you are interested, you can examine the files, but they are really just boiler-plate instructions that will enable Flask to work.

Part 10. Copying over the files

1. Exit out of EC2, and `cd` to where your Flask code is on your local machine.
2. Using your DNS value for your EC2 instance, copy your code and templates directory to your EC2 instance using the following command:


```
scp -r -i <path-to-your-pemfile>.pem flaskapp.py natlpark.db  
templates/ <dns>:/var/www/html/flaskapp/
```

For example, my flask code was in a directory one level deeper than my .pem file. (My flask code was in a directory called FlaskApp that was inside my CS178 folder.) Thus, my command looked like this:

```
scp -r -i ../DrakeCS178keypair.pem flaskapp.py natlpark.db  
templates/ ec2-user@ec2-3-95-56-171.compute-  
1.amazonaws.com:/var/www/html/flaskapp/
```

Next, ssh to the EC2 instance.

cd to the directory where the code is located:

```
[ec2-user ~]$ cd /var/www/html/flaskapp/
```

You'll need to update the path to the SQLite database. Execute the following

```
ec2-user ~]$ nano flaskapp.py
```

scroll down until you see

```
DATABASE = 'natlpark.db'
```

Change it to read

```
DATABASE = '/var/www/html/flaskapp/natlpark.db'
```

Type Control-x (to exit)

Type Y (to accept changes)

Type Return (to accept filename)


Finally, in the event that you're still running your web servers, you'll need to re-name your index.html file. Let's rename it to oldindex.html

```
ec2-user ~]$ cd /var/www/html/  
ec2-user ~]$ mv index.html oldindex.html
```

Lastly, restart the web server with the following command:

```
[ec2-user ~]$ sudo service httpd restart
```

At this time, your RESTful API should be up and running at the IP address of your web server!



← → ↻ ⓘ Not Secure 3.95.56.171/viewdb			
Park	State	Year	Area
National Park of American Samoa	American Samoa	1988	8256.67
Arches National Park	Utah	3000	76678.98
Big Bend National Park	Texas	3001	801163.21
Biscayne National Park	Florida	1980	172971.11

For the Points:

1. Answer the following in two or three sentences (each). You can cite the lab and/or lecture, but should find at least one additional external resource to support your answers:
 - What is a RESTful API?
 - What is CRUD?
 - What is HTTP?
 - How are they all related?
2. Describe how far you (or your group) got with this process. If you were able to complete it, submit your URL. If you got stuck, explain where and what you tried to do to solve this problem.