Project Introduction

From Wikipedia:

“Representational state transfer (REST) or RESTful web services are one way of providing interoperability between computer systems on the Internet. REST-compliant web services allow requesting systems to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations.”

In other words, REST APIs permit separate services to communicate with each other, using a simple to use system of operations. REST APIs use verbs such as GET, POST, PUT, DELETE to indicate what type of operation must be performed. GET is used to obtain information from an API, POST is used to send information, PUT is used to update pre-existing information, and DELETE does just that – delete.

When using APIs, users sometimes are required to provide some sort of authentication. This is true for APIs that store personal or otherwise sensitive information. A great number of popular APIs use a system called OAuth, but we will be using a simpler mechanism.

Let us use the SmartPlug API as an example. The SmartPlug API functions as a portal to send and obtain different bits of information generated by the SmartPlug. One such piece of information can be something like temperature, humidity, or brightness. In order to send that information to the API, a user must send a POST request to the API with the desired information.

Requests have different fields to fill in with this information: parameters, header, and body. Headers are meant to contain metadata of the state, and nothing else. The token mentioned above is a good example of this. Parameters are used to specify the exact part of the information you are requesting. If you are requesting temperature data but you want just the last 3 minutes of information, you would specify that in the parameters. And finally, the body contains the remainder of all data. When posting a brightness reading, the data for it would go in the body.

Required Downloads:

Latest version of Postman: https://www.getpostman.com/
Works as a Chrome app, or a macOS app.

Required Reading:

JSON Quick tutorial: https://www.tutorialspoint.com/json/json_quick_guide.htm

Quick References:

REST API YouTube video. Explains the fundamental concepts of this topic:
https://www.youtube.com/watch?v=7YcW25PHnAA
In depth Tutorial. Covers the architectural style in depth
http://www.restapitutorial.com/lessons/whatisrest.html

Register for the API

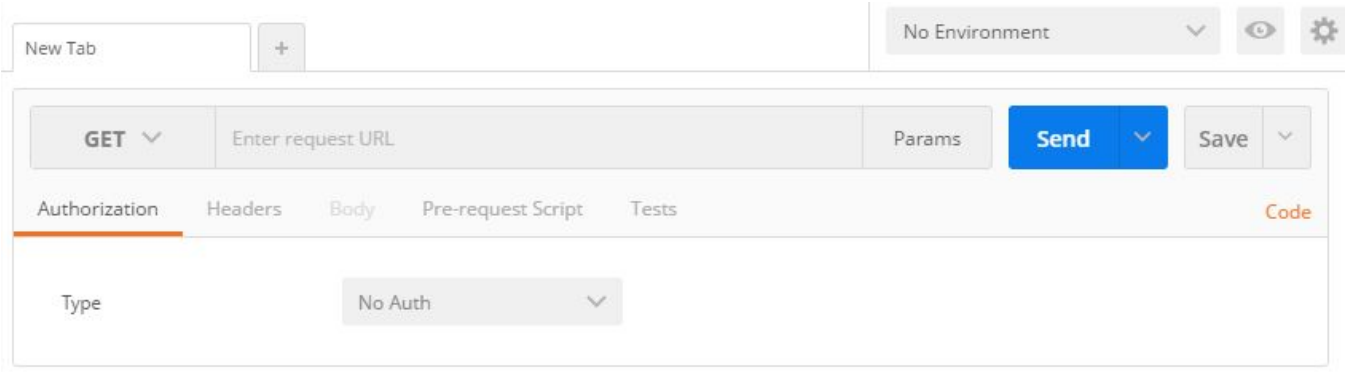
https://goo.gl/forms/DQb4A9rOVu0kAMVy1

Challenge #1: Using Postman

Objective: In order to begin using the SmartPlug API, it is very important to understand all of its components. This challenge focuses on using Postman in conjunction with the SmartPlug API. We will run through the examples necessary for the coding portion of this lab.

1) Login:

a. Open Postman. For the moment being, you are only required to focus on this part of the screen:

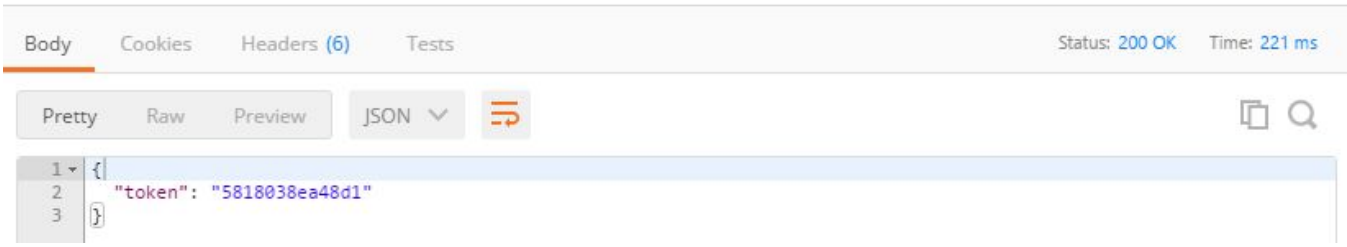


Green: Request type
Red: Parameters
Blue: Headers
Yellow: Body

b. Let's try logging in. Enter:
`http://smartplug.host/api/v1/auth/login`
as the POST request URL. Then under Body, enter two key-value pairs for your credentials, as such:

Please enter your credentials.

a. Send this request, and you will see a response:



Congratulations! You have successfully logged into the SmartPlug API (via Postman that is…)
The token obtained in this step will be necessary for every other single API call. Don't forget this!

b. Now we shall add a device, using our shiny new token. Copy the token provided in the previous step, and then make a POST request to:
`http://smartplug.host/api/v1/devices/`
Use a header key-value pair of:
`Token-Authorization: [your token from the previous step]`
And a body key-value pair of
`device_id: [your device's MAC Address]`
(ifconfig -a, find wlan0)

c. You should see a response like this:

d. Now, let's try to see our current devices. Using the exact same URL and header key-value pair, make a GET request this time. You should see a response with your device names, and a counter with the number of registered devices (1 for now).

e. Finally, we will add a mock brightness reading to the API. We POST to the URL:
`http://smartplug.host/api/v1/devices/[your MAC Address]/light`
With a key-value pair of
`data: 1000`
in the body

f. In order to confirm that the data was successfully posted, make a GET request to the exact same URL.

Challenge #2: Python Requests

Objective: Postman works very well for testing an API, but it is not effective for an automated system such as our Smart Plug. It is significantly easier to automatically do all API calls from within a function, and that is exactly what we will be building for this challenge.

1) Python requests library:

a. http://docs.python-requests.org/en/master/ has an amazing library designed to work with APIs. Think of this library as a python code version of Postman. We will be heavily relying on this framework for our own API system.

b. To install, run on your terminal:
`sudo pip install requests`
This will allow you to use the library in whatever code you desire.

c. Let's create a sample function to automate the token creation. Create a new folder in your directory named `RPi_API_Interface`, and create a file called `RPi_API_Interface.py` with the following code:

```
import requests

def get_token(email, password):
    url = 'http://smartplug.host/api/v1/auth/login'
    data = {'email': email, 'password': password}

    resp = requests.post(url, data=data)
    parsed_json = resp.json()
    return parsed_json

if __name__ == "__main__":
    email = raw_input("Email: ")
    password = raw_input("Password: ")
    print get_token(email, password)
```

d. Let's explain this code, section by section:

i. The first line of the file imports the requests library, so we can use all of its functions.

ii. We define a function **get_token** that takes our email and password.

iii. A dictionary called **data** is created with the necessary information to make a request to our login URL.

iv. We make a request using the imported library, by using the **`requests.post(url, data=data)`** function.

v. The response is stored to a **resp** variable, which we then turn to JSON and return.

vi. The final block of code is simply the 'executable' portion of this python file. **`if __name__ == "__main__":`** is basically asking if the file is being executed (in this case, it is) and if so, proceed to the code below.

vii. **raw_input** lets the user send data to the program through terminal input.

viii. Finally we call our new **get_token** and print its results.

Notes: We will be referencing this file from other files in the future, which is why it is important to check if the current file is the one being executed.

2) Make your own requests:

a. For the final part of this challenge, you are required to make python functions to:
   i. Add a device to the API
   ii. GET all registered devices
   iii. POST a brightness reading
   iv. GET that same brightness reading
b. In order to achieve that, you must use the other requests library functions. All the necessary documentation can be found at http://docs.python-requests.org/en/master/
c. The functions must be named:
   i. add_device (token, device_id)
   ii. get_devices (token)
   iii. add_light (token, device_id, light)
   iv. get_light (token, device_id)
d. The response is required to be in JSON, the same JSON string returned from the API

3) Run your calls:

a. Run add_light once a minute, for an hour. No need to submit anything for this part, we will be able to see it on the API.

Turn in Instructions:

- Submit all your code to github.
- Email a PDF file with all pertinent analysis, your device's MAC Address, and your GitHub repository URL.
- Include Team name in File name (TeamName_Lab4.pdf)
- State which API email+password combination was used in the add_light example

Log-In

POST	/api/v1/auth/login

This method allows a user-agent to log into the REST api for the SmartPlug server.

ARGUMENTS

META Arguments	Type	Required	Default
email	string	yes	
password	string	yes	

RESPONSE

Token : System-generated key for use in subsequent REST API calls

HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```

Get List of Devices

GET	/api/v1/devices

This method retrieves a list of devices registered to the current user.

HEADER

Header Arguments	Type	Required	Default
Token-Authorization	string	yes	

RESPONSE

Returns a json-encode collection of device names:

“count”: n,

"devices":[

{“mac": current_mac_addr, “last_event”: timestamp},

{“mac": current_mac_addr, “last_event”: timestamp}]

HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```

Register Device

POST	/api/v1/devices

This method allows a caller to register a device for their account.

ARGUMENTS

META Arguments	Type	Required	Default
device_id	Unique string	yes	

HEADER

Header Arguments	Type	Required	Default
Token-Authorization	string	yes	

RESPONSE

device_id : Returns the device_id if all went well.

HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```

Get Device Information

GET	/api/v1/devices/{id}

This method retrieves the known information about a named device.

HEADER

Header Arguments	Type	Required	Default
Token-Authorization	string	yes	

RESPONSE

Returns a json-encode collection of device names:

"properties":[...]

HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```

Get Device Temperature Data

GET	/api/v1/devices/{id}/temperature/{count?}

This method retrieves the temperatures recorded by a named device.

HEADER

Header Arguments	Type	Required	Default
Token-Authorization	string	yes	

RESPONSE

Returns a json-encoded collection of temperatures/timestamps for “count” records. Not specifying “count” retrieves all recorded temperatures by the sensor. Specifying a 1 retrieves only the most recent temperature. An example result for a single record:

{"error": "none",
 "data": [{
     "ts": "2016-09-01 10:56:18",
     "temp": "37.5"}]
}

HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```

Add Device Temperature Data

POST	/api/v1/devices/{id}/temperature

This method allows a user to record a temperature reading (“data”) from a given device.

ARGUMENTS

META Arguments	Type	Required	Default
data	String	yes	

HEADER

Header Arguments	Type	Required	Default
Token-Authorization	string	yes	

RESPONSE

event_id : Returns the ID of the temperature record if all went well.

HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```

Get Device Light Data

GET	/api/v1/devices/{id}/light/{count?}

This method retrieves the light recorded by a named device.

HEADER

Header Arguments	Type	Required	Default
Token-Authorization	string	yes	

RESPONSE

Returns a json-encoded collection of light/timestamps for “count” records. Not specifying “count” retrieves all recorded light by the sensor. Specifying a 1 retrieves only the most recent light. An example result for a single record:


```
{"error": "none",
 "data": [{
     "ts": "2016-09-01 10:56:18",
     "light": "37.5"}]
}
```


HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```

Add Device light Data

POST	/api/v1/devices/{id}/light

This method allows a user to record a light reading (“data”) from a given device.

ARGUMENTS

META Arguments	Type	Required	Default
data	String	yes	

HEADER

Header Arguments	Type	Required	Default
Token-Authorization	string	yes	

RESPONSE

event_id : Returns the ID of the light record if all went well.

HTTP Status

Success: (200); Error: (401)

EXAMPLE

```
Put example code here...
```